Ladon: High-Performance Multi-BFT Consensus via Dynamic Global Ordering

Anonymous Author(s)

Abstract

Multi-BFT consensus runs multiple leader-based consensus instances in parallel, circumventing the leader bottleneck of a single instance. However, it contains an Achilles' heel: the need to globally order output blocks across instances. Deriving this global ordering is challenging because it must cope with different rates at which blocks are produced by instances. Prior Multi-BFT designs assign each block a global index before creation, leading to poor performance.

We propose *Ladon*, a high-performance Multi-BFT protocol that allows varying instance block rates. Our key idea is to order blocks across instances dynamically, which eliminates blocking on slow instances. We achieve dynamic global ordering by assigning *monotonic ranks* to blocks. We pipeline rank coordination with the consensus process to reduce protocol overhead and combine aggregate signatures with rank information to reduce message complexity. Besides, the dynamic ordering enables blocks to be globally ordered according to their generation, which respects inter-block causality. We implemented and evaluated Ladon by integrating it with both PBFT and HotStuff instances. Our evaluation shows that Ladon-PBFT (resp., Ladon-HotStuff) improves the peak throughput of the prior art by ≈ 8× (resp., 2×) and reduces latency by ≈ 62% (resp., 23%), when deployed with one straggling replica (out of 128 replicas) in a WAN setting, and Ladon achieves similar performance improvements in a LAN setting.

Keywords: Multi-BFT consensus, Straggler nodes, Dynamic global ordering, Performance

1 Introduction

Byzantine Fault Tolerant (BFT) consensus is crucial to establishing a trust foundation for modern decentralized applications. Most existing BFT consensus protocols, like PBFT [7] or HotStuff [38], adopt a leader-based scheme [7, 9, 21], in which the protocol runs in views, and each view has a delegated replica, known as the leader. The leader is responsible for broadcasting proposals (*i.e.*, a batch of client transactions) and coordinating with replicas to reach a consensus on its proposals. However, a leader can become a significant performance bottleneck, especially at scale. The workload of a leader increases linearly with the number of replicas [2, 16, 23, 35, 36], making the leader the dominant factor in the system's throughput and latency.

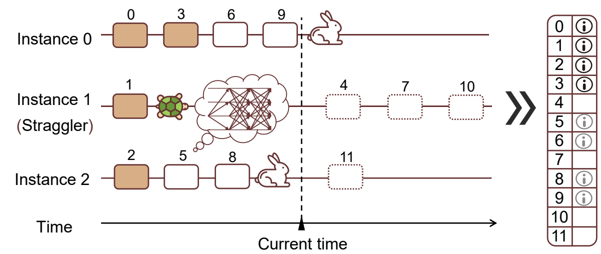


Figure 1. An overview of Multi-BFT paradigm. The shaded (resp., white) blocks refer to the globally confirmed (resp., partially committed) blocks. The dashed blocks refer to the blocks to be produced in the future.

To address the leader bottleneck, Multi-BFT systems [2, 23, 35, 36] have emerged as a promising alternative to build large-scale decentralized applications. Multi-BFT consensus runs multiple leader-based BFT instances in *parallel*, as shown in Fig. 1. A replica can work as a leader for one instance and a backup for others. Like a single BFT system, each BFT instance in Multi-BFT outputs a sequence of committed blocks, which will never be reverted in the partial sequence (as opposed to the global sequence introduced shortly). These blocks are referred to as being partially committed. Then, these blocks are ordered in a global sequence and become globally confirmed, functioning as a single instance system. Such a scheme can balance the workloads of replicas and fully utilize their bandwidth, thereby increasing the overall system throughput.

However, the Achilles' heel of Multi-BFT consensus lies in its global ordering. Existing Multi-BFT protocols [2, 23, 35, 36] follow a *pre-determined* global ordering: a block is assigned a global index that depends solely on two numbers, its instance index and its sequence number in the instance's output. As a concrete example, consider Fig. 1 with three instances outputting four blocks (produced and to be produced in the future). The three blocks from Instance 2 receive global indices of 2, 5, 8, and 11. Replicas execute partially committed blocks with an increasing global index one by one until they see a missing block.

When applied to decentralized systems, this simple global ordering method has performance issues. A slow leader, often called a straggler, can slow down an instance, consequently affecting the entire system's performance. For example, the instance with a straggling leader in Fig. 1 only outputs one block (with a global index of 1). This causes three "holes" in the global log (at positions 4, 7 and 10), and prevents four blocks (5, 6, 8, and 9) from being globally confirmed. This reduces the system's throughput and increases latency, posing a challenge for building high-performance Multi-BFT systems. (Further theoretical analysis and detailed experimental results to illustrate the impact of stragglers are provided in Sec. 2.1). This challenge is prevalent, especially in decentralized systems with open participation and widespread engagement, where variations in replica performance and network conditions make straggling leaders commonplace. Besides, an adversary can easily violate the causality of blocks across instances, allowing it to front-run its transactions ahead of others and gain an unfair advantage in centralized applications such as auctions and exchanges. (See more discussion in Sec. 4.4.)

In this paper, we propose Ladon[1], a high-performance Multi-BFT consensus that considers instances' varying block rates. The insight of our approach is to *dynamically* order partially committed blocks from different instances by their assigned *monotonic ranks* at production. The rank assignment satisfies two properties: 1) *agreement*: all honest replicas have the same *rank* for a partially committed block; and, 2) *monotonicity*: the ranks of subsequent generated blocks are always larger than that of a partially committed block to respect the block causality at best effort. Here, the agreement property ensures that replicas that use a deterministic algorithm to order partially committed blocks by their ranks can get the same ordering sequence, while monotonicity tries to preserve inter-block causality.

The above dynamic global ordering not only decouples the dependencies between the replicas' partial logs to ensure fast generation of the global log, eliminating the straggler impact but also orders blocks by their generation sequence, preserving best-effort causality for inter blocks. For instance, consider Fig. 1 with instances using the monotonic rank for subsequent blocks. The next block of Instance 1 will be assigned the rank of 10 instead of 4, forcing it to be globally ordered after existing partially committed blocks. Moreover, the dynamic ranks enable Instance 1 to quickly synchronize with other instances by one block, which can alleviate the impact of stragglers.

Achieving monotonic ranks is challenging due to the presence of Byzantine behaviors. Replicas need to agree on the ranks of blocks to achieve consensus. To maintain monotonicity, it is crucial that malicious leaders do not use stale ranks or should not exhaust the range of available ranks. A leader has to choose the highest rank from the ranks collected from more than two-thirds of the replicas and increase it by one. To ensure that the leader follows these rules, each block includes a set of collected ranks and the associated proof (*i.e.*, an aggregate signature) of the chosen ranks. However, this basic solution introduces latency and overhead. To optimize this solution, we pipeline the rank information collection with the last round of consensus and further combine aggregate signatures with rank information to reduce the message complexity.

We built end-to-end prototypes of Ladon with PBFT and HotStuff; we call this Ladon-PBFT and Ladon-HotStuff, respectively. Furthermore, we believe Ladon can complement and improve any single-leader BFT protocol. We conduct extensive experiments on AWS to evaluate and compare Ladon with existing Multi-BFT protocols, including ISS [36], RCC [23], Mir [35], and DQBFT [1]. We run experiments over LAN and WAN with $8-128$ replicas, distributed across 4 regions. With one straggler in WAN, Ladon-PBFT (resp. Ladon-HotStuff) achieves 8× (resp. 2×) higher throughput and 62% (resp. 23%) lower latency with 128 replicas as compared to ISS-PBFT (resp. ISS-HotStuff). Over LAN, Ladon demonstrates performance trends similar to those observed in the WAN setting. We show that Ladon always maintains a superior level of causality strength as compared to ISS, RCC, Mir, and DQBFT with stragglers.

2 Existing Multi-BFT Susceptibility

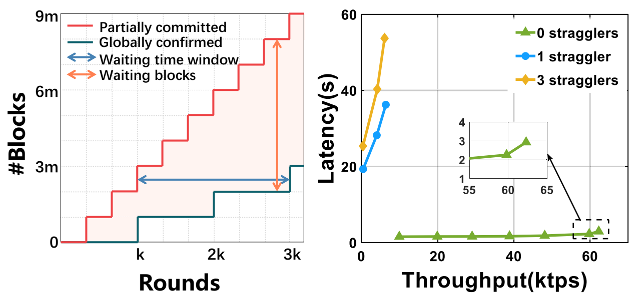
2.1 Performance Degradation

Existing Multi-BFT protocols [2, 23, 35, 36] with the predetermined global ordering perform well when all instances have the same block production rate. With m instances, they can achieve m times higher throughput and similar latency to single-instance systems. By adjusting m, the system can maximize the capacity of each replica, allowing the throughput to approach the physical limit of the underlying network.

However, performance will significantly drop when there are straggling instances caused by faulty or limited-capacity leaders, unstable networks, or malicious behaviors. Consider a simple case where a slow instance with a straggling leader produces blocks every k rounds while the remaining $m-1$ normal instances produce blocks every round. Let R (resp., R') denote the number of partially committed (resp., globally confirmed) blocks per round. We have $R = 1/k + m - 1$ and $R' = m/k$, which implies that the system throughput is about $1/k$ of the ideal scenario. Over time, the accumulation of $R' - R$ blocks every round leads to a continuous delay increase in waiting for global confirmation.

Fig. 2a shows the analytical results for the case above. First, we observe that the number of partially committed blocks that wait to be globally confirmed grows over time. Similarly, the delay for partially committed blocks to become globally confirmed also grows over time. Fig. 2b plots experimental results of throughput and latency (defined in Sec. 6.2) to show the practical impact of stragglers. We run ISS [36], a state-of-the-art Multi-BFT protocol, in which consensus instances are instantiated with PBFT [7]. We set $m = 16$, and show results for 0, 1, and 3 stragglers in WAN. Other settings are the same as Sec. 6.2. With 1 and 3 stragglers,

[1] Ladon is a monster in Greek mythology, the dragon with one hundred heads that guarded the golden apples in the Garden of the Hesperides.



(a) Analytical results **(b)** Experimental results

Figure 2. The analytical and experimental performance of Multi-BFT consensus with/without a straggler. The vertical line in (a) represents the queued partially committed blocks, while the horizontal line represents the delay of the global ordering.

the maximum throughput is reduced by 89.7% and 90.2% of the system's throughput with 0 stragglers, respectively. The latency with 1 and 3 stragglers increases up to 12× and 18× of the system's latency with 0 stragglers, respectively.

2.2 Revisiting Straggler Mitigation

Most existing methods focus on detecting straggling leaders and then replacing them with normal ones. However, we show that this detection-then-replacement approach cannot fix these issues. This motivated us to design a dynamic global ordering mechanism to mitigate the impact of stragglers algorithmically. Stathakopoulou *et al.* [35] propose to use the timeout mechanism to replace leaders of the instances that do not timely output partially committed blocks. Similarly, in RCC [23], a straggling leader will be removed once its instance lags behind other instances by a certain number of blocks. These mechanisms fall short in three ways. First, if there are multiple colluding stragglers in the system, it is difficult to detect them. Second, replicas that perform poorly due to lower capacities will be replaced, resulting in poor participation fairness for decentralized systems. Third, straggling leaders are only one cause of not timely producing partially committed blocks by instances. Network turbulence and dynamically varying replica capacities could also cause an instance to produce blocks slowly for a period of time.

Unlike these methods, DQBFT [1] mitigates the impact of stragglers by adding a special instance to globally order partially committed blocks from other instances. However, this centralized instance can become a performance bottleneck once it suffers from straggling leaders since the leader can easily manipulate the global ordering of blocks.

3 Models and Problem Statement

3.1 System Model

We consider a system with $n = 3f+1$ replicas, denoted by the set $\mathcal{N}$. A subset of at most f replicas is *Byzantine*, denoted as $\mathcal{F}$. Byzantine replicas can behave arbitrarily. The remaining replicas in $\mathcal{N} \setminus \mathcal{F}$ are honest and strictly follow the protocol. All the Byzantine replicas are assumed to be controlled by a single adversary, which is computationally bounded. Thus, the adversary cannot break the cryptographic primitives to forge honest replicas' messages (except with negligible probability). There is a public-key infrastructure (PKI): each replica has a pair of keys for signing messages.

We assume honest replicas are fully and reliably connected: every pair of honest replicas is connected with an authenticated and reliable communication link. We adopt the partial synchrony model of Dwork *et al.* [13], which is widely used in BFT consensus [7, 38]. In the model, there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all message transmissions between two honest replicas arrive within a bound Δ. Hence, the system is running in *synchronous* mode after GST.

Like classical BFT consensus [7, 38], we assume that Byzantine replicas aim to destroy the *safety* and *liveness* properties by deviating from the protocol. Besides, they may strategically delay their operation (e.g., without triggering timeout mechanisms), appearing as stragglers to either compromise performance or violate block causality (Sec. 4.4). It is worth noting that in decentralized applications, due to replicas' heterogeneous capacities, honest replicas may also behave as stragglers.

3.2 Preliminaries

We introduce some building blocks in this work.

Sequenced Broadcast (SB). We follow ISS [36] to use Sequenced Broadcast (SB) for consensus instances. SB is a variant of Byzantine total order broadcast with explicit round numbers and an explicit set of allowed messages. In particular, given a set of messages M and a set of round numbers R, only one sender p (*i.e.*, the leader) can *broadcast* a message (msg, r), where $(msg, r) \in M \times R$. Besides, honest replicas can *deliver* a message msg with round number r. If an honest replica suspects that p is quiet, all correct nodes can deliver a special *nil* value $msg = \perp \notin M$. Otherwise, honest replicas can deliver non-nil messages $m \neq \perp$. There is a failure detector D to detect *quite* sender. SB is implementable with consensus, Byzantine reliable broadcast (BRB), and a Byzantine fault detector [36]. An instance of SB (p, R, M, D) has the following properties:

- **SB-Integrity**: If an honest replica delivers (msg, r) with $msg \neq \perp$ and p is honest, then p broadcast (msg, r).
- **SB-Agreement**: If two honest replicas deliver (msg, r) and (msg', r), then $msg = msg'$.
- **SB-Termination**: If p is honest, then p eventually delivers a message for every round number in R, *i.e.*, $\forall r \in R : \exists msg \in M \cup \{\perp\}$ such that p delivers (msg, r).

Blocks. A block B is a tuple $(txs, index, round, rank)$, where txs denotes a batch of clients' transactions, $index$ denotes the index of the consensus instance, $round$ denotes the proposed round, and $rank$ denotes the assigned monotonic rank. We use $B.x$ to denote the associated parameter x of block B. For example, $B.txs$ is the set of included transactions. If two blocks have the same $index$, we call them intra-instance blocks; otherwise, we refer to them as inter-instance blocks. It's important to note that when a block is globally confirmed (as introduced shortly), replicas can compute a unique global ordering index sn for it. In other words, sn is not a predetermined field of the block. In the following, we still use $B.sn$ for clarity.

3.3 Problem Formulation

We consider a Multi-BFT system consisting of m BFT instances indexed from 0 to $m-1$. Thus, the ith $(1 \le i \le m)$ instance has an index of $i-1$. The system can be divided into two layers, a *partial ordering layer* $\mathcal{P}$ and a *global ordering layer* $\mathcal{G}$.

Partial ordering layer. Clients create and send their transactions to replicas for processing, which constitute the input of the partial ordering layer $\mathcal{P}_{in}$. We assume there is a mechanism (*e.g.*, rotating bucket [35]), which assigns client transactions to different instances to avoid transaction redundancy. Each instance runs an SB protocol, and in each round, a leader packs client transactions into blocks, proposes (*i.e.*, *broadcast* in SB) the blocks, and coordinates all replicas to continuously agree on the blocks. We denote the block produced by instance i in round j as B^i_j. The output of the partial ordering layer is a collection of m sequences of *partially committed* (*i.e.*, *delivered* in SB) blocks produced by all the instances, where the ith sequence from ith instance is denoted by $\langle B^i_1, B^i_2, ..., B^i_{k_i}\rangle$, and k_i is the number of partially committed blocks on the ith instance till now. We denote the entire collection by $\mathcal{P}_{out} = \langle B^i_1, B^i_2, ..., B^i_{k_i}\rangle^{m-1}_{i=0}$.

Global ordering layer. A Multi-BFT system should perform as a single BFT system, and so blocks output by the partial ordering layer across m instances should be ordered into a global sequence. Thus, the input of the global ordering layer is the output of the partial ordering layer, *i.e.*, $\mathcal{G}_{in} = \mathcal{P}_{out} = \langle B^i_1, B^i_2, ..., B^i_{k_i}\rangle^{m-1}_{i=0}$. Following certain ordering rules (which vary according to different designs), these input blocks are ordered into a sequence, and the associated index is denoted as sn. These output blocks are globally committed and executed, *i.e.*, *globally confirmed*, denoted by the set $\mathcal{G}_{out}$. Blocks in $\mathcal{G}_{out}$ satisfy the following properties:

- $\mathcal{G}$**-Agreement**: If two honest replicas globally confirm $B.sn = B'.sn$, then $B = B'$.
- $\mathcal{G}$**-Totality**: If an honest replica globally confirms a block B, then all honest replicas eventually globally confirm the block B.
- $\mathcal{G}$**-Liveness**: If a correct client broadcasts a transaction tx, an honest replica eventually globally confirms a block B that includes tx.

4 Dynamic Global Ordering

4.1 Monotonic Rank

We first present the required properties of monotonic ranks and then discuss how to realize them.

Properties. The global ordering layer assigns partially committed blocks with monotonic ranks (short for rank in the following discussion). These ranks will determine the output block sequence of the system. **M**onotonic **r**anks have two key properties:

- **MR-Agreement:** All honest replicas have the same rank for a partially committed block.
- **MR-Monotonicity:** If a block B' is generated after an intra-instance (or a partially committed inter-instance) block B, then the rank of B' is larger than the rank of B.

These two properties collectively ensure that given a set of partially committed blocks with ranks, honest replicas can independently execute an ordering algorithm to produce a consistent sequence of globally confirmed blocks without any additional communication. Specifically, blocks are ordered by increasing rank, with a tie-breaking of instance indexes. Note that MR-Monotonicity also guarantees causality between any two intra-instance blocks. The reason inter-instance block causality is not guaranteed is that it is impossible to determine causal order between concurrent blocks generated by different instances without a trusted global time [26].

The realization. We first show how to achieve the MR-Agreement property for blocks' ranks without running additional consensus protocols. In particular, a block can be assigned a rank either when it is proposed or after it is output by a consensus instance (*i.e.*, running SB). We observe that for the former approach, no additional procedure is required to achieve the property since the rank is piggybacked with the block that has to go through the consensus process. By contrast, the latter approach requires an extra consensus process. Thus, we use the former approach for efficiency.

Second, to achieve MR-Monotonicity, a leader first collects the highest ranks from at least $2f+1$ replicas. It then increments the highest rank among these by one and assigns this as the rank for its proposed block. However, there exists a potential vulnerability: a malicious leader might attempt to disrupt monotonicity by introducing stale ranks. To counteract this, each collected rank is sourced from blocks that have received sufficient votes and are accompanied by cryptographic certificates, which validate their authenticity (see Sec. 5.2.2). These collected ranks and certificates are then integrated into the proposed block for validation. Consequently, even if a Byzantine leader attempts to manipulate the ranks, the authenticity checks constrain it. We further analyze the impact of possible Byzantine behaviors on system performance in Sec. 4.3.

Overhead analysis. The above approach has two overheads: the rank collection process (one round of communication) and the larger block size. To mitigate the former, we integrate the rank collection into the consensus phases of the prior block. This eliminates extra communication, significantly reducing rank collection latency and minimizing overhead. As for the latter, we observe that the increased block size is negligible given the block payload size. For example, the additional rank information and certificates included in blocks comprise less than 1‰ of the total block size (*i.e.*, 2MB) with 100 replicas. We also use a custom aggregate signature mechanism to further reduce rank information included in blocks (Appendix A). Specifically, a block only needs to include one aggregated signature as a certificate of the collected ranks.

4.2 Global Ordering Algorithm

Algorithm 1 shows the global ordering process running at a replica. The algorithm takes the set $\mathcal{G}_{in}$ of partially committed blocks as its input (which is the output from the partial ordering layer) and outputs the set $\mathcal{G}_{out}$ of globally confirmed blocks (Sec. 3.3).

Algorithm 1 is based on two basic ideas. First, partially committed blocks can be globally ordered by increasing ranks and a tie-breaking to favor block output from consensus instances with smaller indices. For example, given two blocks B and B', block B will be globally ordered before B', when $B.rank < B'.rank$ or $B.rank = B'.rank \wedge B.index < B'.index$. For convenience, we use $B \prec B'$ to denote that block B has a lower global ordering index than block B'. Second, a partially committed block can be globally confirmed if its global index is lower than a certain threshold (which will be defined later).

Now we describe the algorithm. When $\mathcal{G}_{in}$ is updated with new partially committed blocks, the replica runs the global ordering algorithm to decide globally confirmed blocks. The key step is to compute a threshold called the confirmation bar (short for *bar*), by which blocks with lower global ordering indices can be globally confirmed. To compute *bar*, the replica first fetches the last partially confirmed block from each instance, denoted by the set $\mathcal{S}'$ (Line 2). Here, a block is partially confirmed only if all previous blocks in the same instance become partially committed. It then finds the block B^* that has the lowest ordering index among the blocks in $\mathcal{S}'$, *i.e.*, $\forall B' \in \mathcal{S}'$ with $B' \neq B^* : B^* \prec B'$ (Line 3). Thereafter, *bar* can be computed as a tuple of $(rank, index)$ (Line 4):

$$bar := (rank, index) = (B'.rank + 1, B'.index).$$

The threshold *bar* represents the lowest global ordering index that can be owned by subsequently generated blocks. The *bar* is initially set as $(0, 0)$ during initialization.

Algorithm 1 The Global Ordering Algorithm

1: **upon** $\mathcal{G}_{in}$ is updated
2: $\quad \mathcal{S}' \leftarrow \text{GETLASTBLOCK}(\mathcal{G}_{in})$
3: $\quad B^* \leftarrow \text{FINDLOWESTBLOCK}(\mathcal{S}')$
4: $\quad bar = (B^*.rank + 1, B^*.index)$ //compute the bar
5: $\quad \mathcal{S} \leftarrow \mathcal{G}_{in} \setminus \mathcal{G}_{out}$
6: $\quad B_{can} = \text{FINDLOWESTBLOCK}(\mathcal{S})$ //find candidate block
7: $\quad$ **while** $B_{can} \prec bar$ //B_{can} has a lower index than bar
8: $\quad\quad \mathcal{G}_{out} \leftarrow \mathcal{G}_{out} \cup B_{can}$ //globally confirm B_{can}
9: $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \setminus B_{can}$ //update $\mathcal{S}$
10: $\quad\quad B_{can} = \text{FINDLOWESTBLOCK}(\mathcal{S})$ //find next B_{can}
11: $\quad$ **end while**

//Return block with the lowest ordering index
12: **function** FINDLOWESTBLOCK($\mathcal{V}$)
13: $\quad B^* \leftarrow$ first block in $\mathcal{V}$
14: $\quad$ **for** each $B \in \mathcal{V}$ **do**
15: $\quad\quad$ **if** $B \prec B^*$
16: $\quad\quad\quad B^* \leftarrow B$
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: $\quad$ **return** B^*

With *bar*, the replica repetitively checks unconfirmed blocks in $\mathcal{G}_{in}$ and decides which blocks to confirm. Specifically, let $\mathcal{S} = \mathcal{G}_{in} \setminus \mathcal{G}_{out}$ be the set of unconfirmed blocks in $\mathcal{G}_{in}$ (Line 5). The replica finds the block $B_{can} \in \mathcal{S}$ that has the lowest ordering index, which is referred to as the *candidate* block (Line 6). If B_{can} has a lower ordering index than *bar*, B_{can} can be globally confirmed because all future blocks will have higher indices than B_{can}. In particular, B_{can} will be added to the set $\mathcal{G}_{out}$ and removed from the set $\mathcal{S}$ (Lines 8-9). The process repeats itself until no such B_{can} can be found (Line 10).

Fig. 3 provides a concrete example of the global ordering process. Suppose at time t_1, a new partially committed block B_2^2 is added to instance 2, a replica has $\mathcal{G}_{in} = \langle B_1^0, B_2^0, B_3^0, B_1^1, B_2^1, B_1^2, B_2^2 \rangle$ and $\mathcal{G}_{out} = \langle B_1^0, B_2^0, B_1^1, B_1^2 \rangle$. According to the above algorithm, we have $\mathcal{S}' = \langle B_3^0, B_2^1, B_2^2 \rangle$, $B^* = B_2^1$, $bar = (3, 1)$, and $\mathcal{S} = \langle B_3^0, B_2^1, B_2^2 \rangle$. The first candidate block in $\mathcal{S}$ is B_2^1, which has a lower rank than *bar* and so will be globally confirmed. Then, B_2^1 is removed from $\mathcal{S}$, and $\mathcal{S} = \langle B_3^0, B_2^2 \rangle$, the candidate block is B_3^0, which can be globally confirmed because it has the same rank while smaller index than *bar* and so will be globally confirmed. Finally, the set $\mathcal{S}$ contains only one block B_2^2 with a higher rank than *bar*. Thus, B_2^2 will not be globally confirmed, and the search for globally confirmed blocks ends.

The dynamic global ordering effectively mitigates the impact of stragglers on performance. Each instance works akin to a separate relay track in a race. Instead of producing blocks

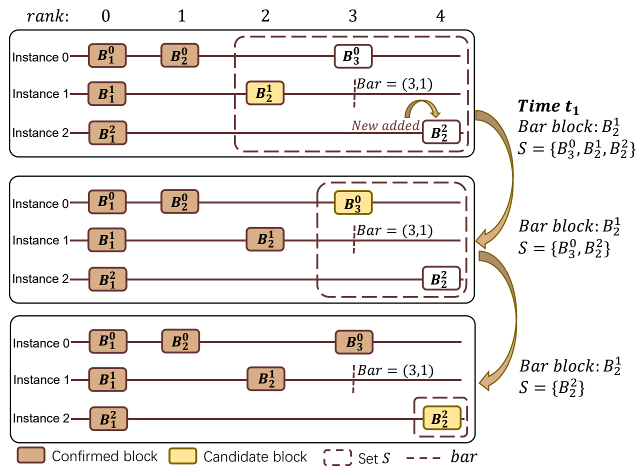


Figure 3. An illustration of the dynamic global ordering process. At time t_1, a new block B_2^2 is partially committed, which makes blocks B_2^1 and B_3^0 globally confirmed.

in a rigid sequential order, the assigned ranks can dynamically adjust the position of each block produced by an instance. This is like allowing slower runners on a relay track to leap ahead in the race to keep up with faster runners on other tracks, improving efficiency and reducing the latency of global ordering.

4.3 Analysis of Byzantine leaders' Impact on Performance

While the dynamic global ordering effectively mitigates stragglers' impact on performance, challenges arise with Byzantine leaders, who may intentionally delay block proposals or strategically minimize ranks for proposed blocks.

Strategic delay of block proposals. The delay in block proposal represents a prevalent Byzantine behavior in BFT consensus mechanisms, posing a challenge due to its indistinguishability from honest conduct. This behavior may be observed even among honest leaders, often arising from a shortage of collected transactions or poor network conditions. This phenomenon underscores a fundamental challenge within BFT systems rather than being specific to Multi-BFT systems.

Minimizing rank for proposed block. Before proposing a block, a leader first collects at least $2f + 1$ highest certified ranks from replicas and then increases the highest of these by one to determine the rank for its new block. However, a Byzantine leader could collect more than $2f + 1$ ranks before proposing a new block, subsequently discarding several high ranks and selecting only the lowest $2f + 1$ ranks.

Now, let's analyze the impact of this strategy. Assume that a Byzantine leader collects at least $2f + 1$ ranks and only selects the lowest $2f + 1$ ranks. Let f' be the actual number of Byzantine replicas (where $f' \leq f$). Then, among the lowest $2f + 1$ ranks, there are at least $2f + 1 - f'$ ranks from honest replicas. So the selected highest rank from these $2f + 1$ ranks is greater or equal to the highest one from the $2f + 1 - f'$ ranks from honest replicas, which is at least the median of all the $n - f'$ ranks from all honest replicas. This implies that the selected highest rank is at least the median of the ranks from honest replicas.

In Ladon-PBFT, when a block is partially committed, at least $2f + 1$ replicas send commit messages. These replicas have all received $2f + 1$ prepare messages, which serve as a quorum certificate for the block's rank. Hence, at least $2f + 1$ replicas (including $f + 1$ honest replicas) have this certified rank. Consequently, the median of the certified ranks among all honest replicas is greater than or equal to the rank of all committed blocks.

To sum up, the selected highest rank by the Byzantine leader is guaranteed to be at least the median of the ranks of all honest replicas, which must be higher than the ranks of all partially committed blocks. Therefore, while the described strategy may introduce some delay or inefficiency, its impact on performance is ultimately mitigated.

4.4 Causality Enhancement

The above dynamic global ordering also respects inter-block causality, which is a highly desirable property for decentralized applications such as auctions and exchanges. It ensures that no one can front-run a partially committed transaction. In sharp contrast, this property is missing in the previous Multi-BFT protocols (with a pre-determined ordering) [2, 23, 35, 36]. An attacker only needs to corrupt one slow leader to front-run partially committed transactions.

To better understand the issues, refer to Fig. 1, where block 4 is proposed after blocks 5, 6, 8, and 9 but is globally ordered and executed before them. Such violations may lead to various attacks including front-running attacks [3, 33], undercutting attacks [20], and incentive-based attacks [6, 14, 15]. For example, in a front-running attack [3, 33], an attacker attempts to place its transaction ahead of some other unexecuted transaction. Consider a cryptocurrency exchange case, in which an attacker sees a large buy order tx_v in block 5, shown in Fig. 1. Then, the attacker creates a similar buy order tx_m in block 4. Since block 4 is placed ahead of block 5 in the global ordering, tx_m is processed before tx_v. As a result, the attacker can buy the cryptocurrency at a lower price, and later sell it back at a higher price to tx_v, profiting at the expense of the original buyer.

Nevertheless, there is still a gap between the above property and an ideal property that no one can front-run a transaction, which is referred to as the client-side causality in prior work [11, 12, 37]. Achieving this strong causality usually requires complicated fair-ordering mechanisms [8], costly cryptographic techniques (*e.g.*, commitment [11] and threshold cryptography [12]) or Trusted hardware (*e.g.*, TEEs [37]). We leave it as future work to use these techniques in Ladon.

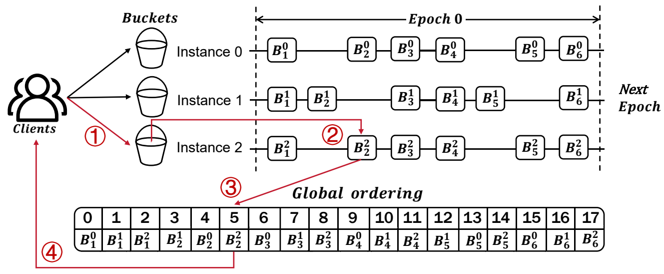


Figure 4. An overview of LADON's design. Client transactions are enqueued into rotating buckets and are then consumed by concurrent BFT protocol instances. Each instance packs transactions into blocks that are eventually totally ordered. Instances execute in epochs.

5 LADON Design

We first provide an overview of LADON in Sec. 5.1 and then introduce the detailed protocol with PBFT consensus instances in Sec. 5.2, and then provide refinements to reduce the message complexity in Sec. ??. We sketch the correctness analysis of LADON in Sec. 5.4. Besides, we also provide analysis of message complexity, a running example, and the design of instantiating LADON with the HotStuff protocol. Due to space constraints, we leave the detailed description in Appendix A, B and D.

5.1 Overview

Fig. 4 provides an overview of the core four components in LADON: rotating buckets, epoch pacemaker, consensus instance, and global ordering. Among them, rotating buckets and the epoch pacemaker bear resemblance to existing paradigms [36], while instance consensus and global ordering are tailored to realize the dynamic global ordering.

Rotating buckets. LADON adopts rotating buckets from ISS [36]. These are used to prevent multiple leaders from simultaneously including the same transaction in a block. Client transactions are divided into disjoint buckets, and these are rotationally assigned to consensus instances when an epoch changes. Bucket rotation mitigates censoring attacks, in which a malicious leader refuses to include transactions from certain clients.

Epoch pacemaker. The epoch pacemaker ensures LADON proceeds in epochs. At the beginning of an epoch, LADON has to configure the number of consensus instances, and the associated leader for each instance, and initialize systems parameters. At the end of an epoch, LADON creates checkpoints of the current epoch across all instances, partially committing the block with the maximum rank of the current epoch. The liveness property inherent to each instance ensures that eventually, every instance will partially commit the block with the maximum rank for the given epoch. Once this condition is satisfied, LADON transitions to the next epoch. More details of the Epoch pacemaker are given in Sec. 5.2.1.

Consensus instances. In each epoch, multiple consensus instances run in parallel to handle transactions from rotating buckets. LADON uses off-the-shelf BFT protocols such as PBFT and HotStuff. Each consensus instance contains (1) a mechanism for normal-case operation, and (2) a view-change mechanism. Sec. 5.2.2 further details these two mechanisms.

Global ordering. The blocks produced by consensus instances are globally ordered by the global ordering algorithm, as detailed in Sec. 4.2. Upon global confirmation of a block, the transactions are sequentially executed, with the results subsequently relayed back to the respective clients.

We now review the flow of client transactions in LADON. These four steps match the numbered red arrows in Fig. 4.

① A client creates a transaction tx, and sends it to some relay replicas. The transaction tx is assigned into one bucket, and forwarded to associated leaders in the current epoch.

② When the leader receives the transactions, it will first pack the transactions in a block. Then, it runs consensus instances with other replicas to partially commit the block.

③ Replica runs the global ordering algorithm to globally confirm output blocks from instances.

④ Once the block is globally confirmed, the replica sends a reply to the client. Upon receiving more than $f+1$ identical replies, the client acknowledges the response as accurate.

5.2 Protocol Description

5.2.1 Epoch Pacemaker. LADON proceeds in epochs. We start with epoch 0, and an empty undelivered block set. All buckets are initially empty.

Epoch initialization. At the start of epoch e, LADON will do the following: (1) calculate the range of $rank$s and instance index, (2) calculate the set of replicas that will act as leaders in epoch e based on the leader selection policy, (3) create a new consensus instance for each leader, (4) assign buckets and indices to the created instances, (5) start the instances (see details in Sec. 5.2.2).

Here, we omit the details of the leader selection policy, bucket assignment policy, and index assignment policy, because they are the same as the policies in ISS [36].

The range of $rank$s for epoch e is denoted as $[minRank(e), maxRank(e)]$. Given an epoch e, the length $l(e)$ is a customizable parameter. It characterizes the number of $rank$s in an epoch, such that $l(e) = maxRank(e) - minRank(e) + 1$. For epoch $e = 0$, $minRank(0) = 0$, and $maxRank(0) = l(0) - 1$. For epoch $e \neq 0$, $minRank(e) = maxRank(e-1) + 1$.

Epoch advancement. LADON advances from epoch e to epoch $e+1$ when all the instances reach $maxRank(e)$, *i.e.*, the blocks with $maxRank(e)$ in all instances have been partially committed. Only then does the replica start processing messages related to epoch $e+1$. To prevent transaction duplication across epochs, LADON requires replicas to globally confirm all blocks in epoch e before proposing blocks for epoch $e + 1$. To this end, LADON adopts a CHECKPOINT mechanism, in which replicas broadcast a checkpoint message in the current epoch before moving to the next epoch. Upon receiving a quorum of $2f + 1$ valid checkpoint messages, a replica creates a stable checkpoint for the current epoch, which is an aggregation of the checkpoint messages. When a replica starts receiving messages for a future epoch $e + 1$, it fetches the missing log entries of epoch e along with their corresponding stable checkpoint, which prove the integrity of the data.

5.2.2 Consensus Instance.

We now describe LADON using PBFT [7] for consensus instances (called LADON-PBFT).

Data structure. Messages are tuples of the form $\langle type, v, n, d, i, rank\rangle_\sigma$, where $type \in$ {PRE-PREPARE, PREPARE, COMMIT, RANK}, v indicates the view in which the message is being sent, n is the round number, d is the digest of the client's transaction, i is the instance index, $rank$ is the rank of the message, $\langle msg\rangle_\sigma$ is the signature of message msg. We use $ppremsg, premsg, commsg, rankmsg$ as the shorthand notation for pre-prepare, prepare, commit, and rank messages, respectively. The instance index is added to mark which instance the message belongs to, since we run multiple instances of consensus in parallel. The parameter $rank$ is the MR, which is used for the global ordering of the blocks. An aggregation of $2f + 1$ signatures of a message msg is called a Quorum Certificate (QC) for it. When we say a replica sends a signature, we mean that it sends the signed message together with the original message and the signer's identity.

Normal-case operation. Algorithm 2 shows the operation of LADON protocol in the normal case without faults. Within an instance, the protocol moves through a succession of views with one replica being the leader and the others being backups in a view. The protocol runs in rounds within a view. An instance starts at view 0 and round 1 and a unique instance index i. The leader starts a three-phase protocol (PRE-PREPARE,PREPARE,COMMIT) to propose batches of transactions to the backups. After finishing the three phases, replicas commit the batch with corresponding parameters. We generally use a VERIFY function to check the validity of a message, such as the validity of the signature and whether the parameters match the current view and round.

1) PRE-PREPARE. This phase is only for the leader. When a leader runs RUNINSTANCE, it directly proposes a batch for round $n = 1$ without any waiting. We assume a special $rankSet$ for round 1, which only contains a $rankmsg$ of the leader. Note that when $n = 1$, the leader doesn't need to wait for $rankmsg$, but let $rankSet[n] \leftarrow \langle$RANK$, v, n - 1, \perp, i, curRank.rank\rangle_\sigma$. In the following rounds $n \neq 1$, upon receiving $2f+1$ $rankmsg$ (including one from itself) for round n in round $n - 1$, the leader proposes a batch for the new round. When a leader proposes for round n, it forms a set $rankSet$ of the $rankmsg$ for round n (Line 2), and picks the maximum rank value with its QC from $rankSet$, denoted as $rank_m$ and QC (Line 3). Then, the leader cuts a batch txs of transactions (Line 4), and calculates the digest of txs (Line 5). A rank number $rank = rank_m + 1$ is assigned to txs, which should not exceed the $maxRank(e)$ of the current epoch (Line 6). The leader multicasts a pre-prepare message (Line

Algorithm 2 The LADON-PBFT Algorithm for Instance i at View v, Round n and Epoch e

▷ PRE-PREPARE phase (only for leader)
1: **upon** receive $2f + 1$ $rankmsg$ **do**
2: $rankSet \leftarrow 2f + 1$ $rankmsg$
3: $rank_m, QC \leftarrow$GETRANK$(rankSet)$
4: $txs \leftarrow$ CUTBATCH$(ins.bucketSet)$
5: $d \leftarrow hash(txs)$
6: $rank \leftarrow min\{rank_m + 1, maxRank(e)\}$
7: $ppremsg \leftarrow \langle$PRE-PREPARE$, v, n, d, i, rank\rangle_\sigma$
8: multicast $\langle ppremsg, txs, QC, rankSet\rangle$
9: **if** $rank = maxRank(e)$
10: stop propose
11: **end if**
12:
▷ PREPARE phase
13: **upon** receive $ppremsg$ **do**
14: **if** VERIFY$(ppremsg)$
15: $premsg \leftarrow \langle$PREPARE$, v, n, d, i, rank\rangle_\sigma$
16: multicast $premsg$
17: **end if**
18:
▷ COMMIT phase
19: **upon** receive $2f + 1$ $premsg$ **do**
20: **if** VERIFY$(premsg)$
21: $commsg \leftarrow \langle$COMMIT $, v, n, d, i, rank\rangle_\sigma$
22: multicast $commsg$
23: **if** $commsg.rank > curRank.rank$
24: $curRank.rank \leftarrow commsg.rank$
25: $curRank.QC \leftarrow$ AGG$(premsg)$
26: **end if**
27: $rankmsg \leftarrow \langle$RANK$, v, n, \perp, i, curRank.rank\rangle_\sigma$
28: send $\langle rankmsg, curRank.QC\rangle$ to leader
29: **end if**
30:
▷ Finally
31: **upon** receive $2f + 1$ $commsg$ **do**
32: **if** VERIFY$(commsg)$
33: $B \leftarrow \langle txs, i, n, rank\rangle$
34: $\mathcal{G}_{in} \leftarrow \mathcal{G}_{in} \cup B$ //Commit B
35: **end if**
36:
37: **upon** receive $rankmsg$ **do**
38: **if** VERIFY$(rankmsg)$ $\wedge rankmsg.rank > curRank.rank$
39: $curRank.rank \leftarrow rankmsg.rank$
40: $curRank.QC \leftarrow rankmsg.QC$
41: **end if**

8) with txs, QC, and $rankSet$to all the backups, where QC is proof for the validity of $rank_m$. The set $rankSet$ is used to prove the leader follows the rank calculation policy. After proposing a batch with the $maxRank(e)$, the leader stops proposing (Lines 9-10).

2) PREPARE. A backup accepts the pre-prepare message after the following validity checks:

- The pre-prepared message meets the acceptance conditions in the original PBFT protocol.
- $rankSet$ contains $2f+1$ ($n \neq 1$) or 1 ($n = 1$) signed $rankmsg$ from different replicas on current view and previous round (*i.e.*, $rankmsg.v = v, rankmsg.n = n - 1$).
- If $rank_m$ is the highest rank in $rankSet$ and $rank_m \neq minRank$, QC is a valid aggregate signature of $2f + 1$ signatures for $rank_m$.
- If $rank_m + 1 \leq maxRank(e)$, $rank = rank_m + 1$; Otherwise, $rank = maxRank(e)$.

The backup then enters the prepare phase by multicasting a $\langle \text{PREPARE}, v, n, d, i, rank\rangle_\sigma$ message to all other replicas (Lines 15-16). Otherwise, it does nothing.

3) COMMIT. Upon receiving $2f+1$ valid prepare messages from different replicas (Lines 19-20), a replica multicasts a commit message to other replicas (Line 22). If the $rank$ carried in the commit message is greater than the current highest rank that the replica knows $curRank.rank$, the replica updates its $curRank$ by setting $curRank.rank$ to $commsg.rank$ (Line 24), and generates a QC for it by aggregating the $2f + 1$ $premsg$ (Line 25). Then, a backup sends a $rankmsg$ together with the QC for $rank$ to the leader to report its $curRank$ (Lines 27-28).

Finally, upon receiving $2f + 1$ valid commit messages, a replica commits a block B with its corresponding parameters (Lines 31-34). All the committed blocks will be globally confirmed and delivered to clients.

4) Respond to clients. When a replica commits a block, it checks whether any undelivered block can be globally confirmed (see Sec. 4.2). If so, it assigns the block a global index sn and delivers it back to clients.

View-change mechanism. If the leader fails, an instance uses the view-change protocol of PBFT to make progress. A replica starts a timer for round $n+1$ when it commits a batch in round n and stops the timer when it commits a batch in round $n + 1$. If the timer expires in view v, the replica sends a view-change message to the new leader. After receiving $2f + 1$ valid view-change messages, it multicasts a new-view message to move the instance to view $v + 1$. Thereafter, the protocol proceeds as described in the previous section. The specific description of the view-change protocol can be found in [7].

5.3 Message Complexity Refinement

We note that the leader must broadcast at least $2f + 1$ rank messages to allow backups to authenticate the accuracy of the leader's $rank$ calculation. This prevents Byzantine leaders from arbitrarily selecting a $rank$ for the new proposal. However, this results in a communication complexity of $O(n^2)$ in the pre-prepare phase (which is $O(n)$ in PBFT). We provide an optimization of LADON-PBFT using aggregate signature schemes to reduce its message complexity. The optimized protocol is referred to as LADON-opt.

High-level Description. Our key idea is to aggregate the $2f + 1$ rank messages into one by using the aggregate signature scheme. Recall that standard multi-signatures or threshold signatures require that the same message be signed [19]. This is, however, not the case for us, because different replicas may have different $rank$ values. Instead, rather than encoding the $rank$ value information in the message directly, we encode it in the private keys.

We modify the rules for generating rank messages as follows. For each replica, we generate K private keys. In round n, when replica r creates a rank message, it computes the difference between the highest rank that it knows(denoted as $rank_r$) and the rank of the current round (denoted as $rank$), *i.e.*, $k \leftarrow (rank_r - rank)$. The replica then signs the message using its kth signature key, *i.e.*, $rankmsg \leftarrow \langle \text{RANK}\ v, n, \perp, i, rank\rangle_{\sigma_{r_k}}$. This scheme allows each replica to sign the same message. Upon receiving a rank message, the leader can recover the $rank_r$ intended to be transmitted by a replica r by computing the sum of $rank$ and k. Note that the rank difference could be beyond the number of private keys. We use the Kth private key to sign the replica's rank difference when the difference is beyond this bound. The K can be adjusted according to stragglers in the real deployment.

Detailed protocol. We present an optimized version of LADON for round $n \neq 1$. There are three modifications.

1) PRE-PREPARE phase. Upon receiving $2f + 1$ rank messages, the leader aggregates the partial signatures into a single signature σ, which the replicas can efficiently verify using the matching public keys. The $rankSet$ is set to the signature σ instead of a set of $2f+1$ rank messages, which reduces the communication complexity from $O(n^2)$ to $O(n)$. The leader obtains the maximum k from the $2f+1$ rank messages, denoted as k_m, and lets $ppremesg.rank = k_m + rankmsg.rank + 1$.

2) PREPARE phase. The backups will check the validity of σ: (a) it is a valid aggregate signature of $2f + 1$ signatures from different replicas; and, (b) $ppremesg.rank = k_m + rankmsg.rank + 1$, where k_m is the maximum k in σ.

3) COMMIT phase. A replica calculates the difference between the highest rank it knows and the $rank$ of the current round as k, *i.e.*, $k \leftarrow curRank.rank - commsg.rank$. If $k < K$, the replica then sends the $rankmsg \leftarrow \langle \text{RANK}\ v, n, \perp, i, commsg.rank\rangle_{\sigma_{r_k}}$ together with $curRank.QC$ to the leader. Otherwise, the replica signs the $rankmsg$ with the Kth private key. Upon receiving a rank message, a replica updates its $curRank$ if $rankmsg.rank + rankmsg.k > curRank.rank$.

5.4 Correctness Analysis Overview

In this section, we provide a brief security analysis of LADON. Due to space constraints, we leave detailed proofs to Appendix C. We show that LADON satisfies *totality*, *agreement*, and *liveness* properties.

Proof sketch. To establish totality, we first prove that all partially committed blocks will eventually be globally confirmed. If a replica globally confirms a block B, it must have partially committed it. According to SB-Agreement and SB-Termination, all honest replicas will partially commit B. Therefore, all honest replicas will globally confirm B. To show agreement, we use induction and proof-by-contradiction. The proof relies on two key observations. First, each block is assigned a unique global ordering index, ensuring a one-to-one mapping between blocks and global ordering indexes. Second, if two honest replicas globally confirm different blocks with the same global ordering index, it directly contradicts the established protocol rules, highlighting that blocks with the same global ordering index must be identical. This straightforward reasoning establishes the uniqueness of the global ordering index and ensures the agreement property. Regarding liveness, our bucket rotation mechanism ensures that all transactions will eventually be assigned to an honest leader for processing. Then we prove that a transaction proposed by an honest leader will be eventually partially committed and then globally confirmed.

6 Evaluation

In this section, we evaluate the performance and causality of LADON across different scenarios. We compare LADON against four state-of-the-art Multi-BFT protocols: ISS [36], RCC [23], Mir [35], and DQBFT [1]. We implemented LADON in Go[2] and used the Go BLS library for aggregate signatures. The evaluation results are illustrated using ChiPlot[3]. We build two end-to-end prototypes of LADON-PBFT and LADON-HotStuff. Due to space constraints, we present the results of LADON-PBFT in this section and leave the results of LADON-HotStuff to Appendix D. For brevity, we refer to LADON-PBFT as LADON in this section. Our experiments aim to answer the following research questions:

- **Q1:** How does LADON perform with varying replicas in WAN and LAN as compared to ISS, RCC, Mir and DQBFT? (Sec. 6.2)
- **Q2:** How does LADON perform under faults? (Sec. 6.3)
- **Q3:** How does the causal strength of LADON compare to that of ISS, Mir, RCC, and DQBFT? (Sec. 6.4)

6.1 Experimental Setup

Deployment settings. We deploy our protocols on AWS EC2 machines with one c5a.2xlarge instance per replica. All processes run on dedicated virtual machines with 8vCPUs and 16GB RAM running Ubuntu Linux 22.04. We conduct extensive experiments of LADON in LAN and WAN environments. For LAN, each machine is equipped with one private network interface with a bandwidth of 1Gbps. For WAN, machines span 4 data centers across France, America, Australia, and Tokyo on Amazon cloud. Replicas are equally distributed across the four regions. Each machine is equipped with a public and a private network interface. We limit the bandwidths of both to 1 Gbps. For WAN experiments, we use the public interface for client transactions and the private interface for server communications. We use NTP for clock synchronization across the servers. We conduct 5 experimental runs for each condition and plot the average value.

System settings. To ensure a fair comparison, we employed the same system configuration across all the protocols (LADON, ISS, RCC, Mir, and DQBFT). Each transaction carries a 500-byte payload, which is the same as the average transaction size in Bitcoin [31]. Each replica can work as a leader for one instance, and as backup replicas for other instances, *i.e.*, $m = n$. We follow ISS by limiting the total block rate (number of blocks proposed by all leaders each second) to 16 $blocks/s$ in WAN and 32 $blocks/s$ in LAN. This prevents the leader from trying to propose too many batches in parallel, which triggers a view change timeout. However, this constraint leads to a higher end-to-end latency as we increase the number of nodes. We allow a large batch size of 4096 transactions. The epoch length is fixed at $l(e) = 64$ for both protocols.

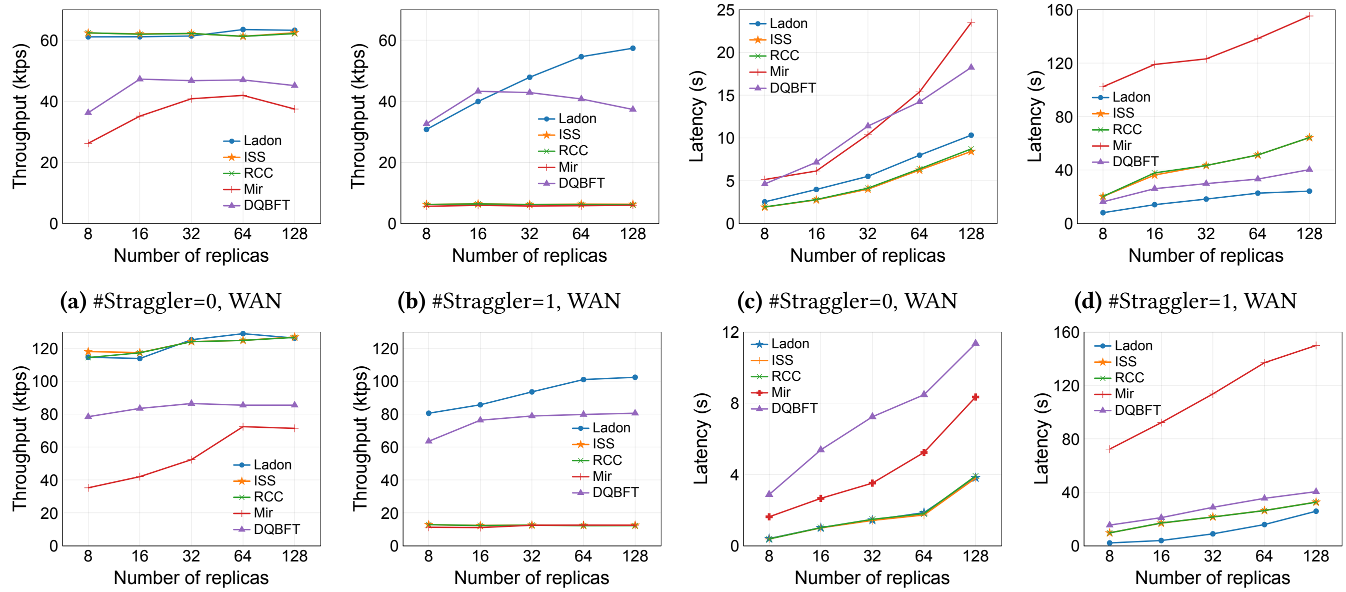
While we acknowledge that the above setting may not be exhaustive or optimal, conducting exhaustive experiments to identify the optimal configuration exceeds the scope of this work. However, our chosen parameters enable us to demonstrate that LADON outperforms other protocols in the presence of stragglers while introducing minimal overhead. This is a key contribution of our work. Here, we focus on discussing the most critical configuration parameters.

Straggler settings. We simulate two types of stragglers. The first type, evaluated in Sec. 6.2 and Sec. 6.4, are honest stragglers. They follow the ISS protocol, delaying proposals for a set period without triggering timeouts and not including transactions in their blocks. Stragglers are randomly selected, with proposal rates (number of blocks proposed by a leader each second) fixed to $1/k$ of normal leaders, where k is a parameter. The second type, evaluated in Sec. 6.3, are Byzantine stragglers. These behave like honest stragglers and also manipulate rank selection by collecting more than $2f + 1$ ranks, discarding the higher ranks, and selecting the lowest $2f + 1$ ranks before proposing a new block.

6.2 Failure-Free Performance

We evaluate the performance of LADON and its counterparts under two conditions: with one honest straggler and without stragglers in both WAN and LAN environments. In this

[2] https://github.com/eurosys2024ladon/ladon
[3] https://www.chiplot.online/



(a) #Straggler=0, WAN **(b)** #Straggler=1, WAN **(c)** #Straggler=0, WAN **(d)** #Straggler=1, WAN

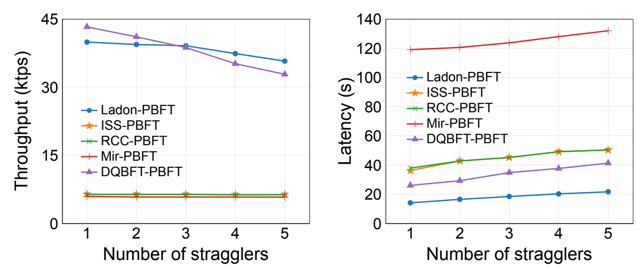
(e) #Straggler=0, LAN **(f)** #Straggler=1, LAN **(g)** #Straggler=0, LAN **(h)** #Straggler=1, LAN

Figure 5. Throughput and latency of Ladon, ISS, RCC, Mir, and DQBFT in WAN (a) – (d), and LAN (e) – (h).

section, we adopt $k = 10$ for stragglers. We measure the peak throughput in kilo-transactions per second (ktps) before reaching saturation along with the associated latency in second (s) in Sec. 6.2.1 and Sec. 6.2.2, and analyze the CPU and bandwidth usage in Sec. 6.2.3. We define the throughput and latency as follows: 1) throughput: the number of transactions delivered to clients per second, and 2) latency: the average end-to-end delay from the time that clients submit transactions until they receive $f + 1$ responses.

6.2.1 Performance in WAN. Fig. 5a shows the throughput of each protocol without stragglers with varying numbers of replicas. Ladon demonstrates a comparable throughput with ISS and RCC, with a minimal difference of approximately 1% on 128 replicas, which demonstrates that Ladon only incurs minimal overhead. Furthermore, we notice that Ladon consistently outperforms Mir and DQBFT. Fig. 5b shows the throughput with one honest straggler. The plot illustrates the superior throughput of Ladon, with 9.1×, 9.4×, and 9.6× of ISS, RCC, and Mir, respectively, on 128 replicas. This is because dynamic global ordering mitigates the performance degradation caused by stragglers that are prevalent in pre-determined global ordering schemes. DQBFT, another dynamic ordering protocol, shows comparable throughput with Ladon initially, but its throughput declines as the number of replicas increases.

Comparing Fig. 5a and Fig. 5b, it is evident that the throughput of pre-determined global ordering protocols (*i.e.*, ISS, RCC, and Mir) significantly drops by 89.9%, 90.1%, and 84.1% on 128 replicas, respectively, in the presence of one straggler. In contrast, dynamic global ordering protocols (*i.e.*, Ladon and DQBFT) are less affected by stragglers, with throughput drops only by 9.3% and 17.3% on 128 replicas, respectively.



(a) Throughput **(b)** Latency

Figure 6. Throughput and latency of Ladon-PBFT and other protocols with a varying number of stragglers.

Fig. 5c and Fig. 5d show the latency for each protocol without stragglers and with one straggler. With one straggler, the latency of Ladon and DQBFT is 2.3× and 2.2× of that with no stragglers on 128 replicas, respectively. In contrast, the latency of ISS, RCC, and Mir with one straggler increases significantly compared to that with no stragglers, achieving 7.6×, 7.4× and 6.6× on 128 replicas, respectively. Without stragglers, Ladon's latency is 22.6% and 18.5% higher compared to ISS and RCC on 128 replicas while much lower than Mir and DQBFT. With one straggler, Ladon shows the lowest latency. The latency of all protocols increases as the number of replicas grows. This phenomenon arises from our decision to maintain a fixed total block rate. Consequently, with more replicas, the time interval for each replica to propose a block becomes longer.

Table 1. CPU and bandwidth usage of Ladon and ISS. Maximum CPU usage is 800%.

Protocols & Settings	Total block rate	CPU	Bandwidth
ISS-0-stragglers	16 blocks/s	319%	85MB/s
	32 blocks/s	566%	160MB/s
ISS-1-straggler	16 blocks/s	132%	25MB/s
	32 blocks/s	292%	77MB/s
Ladon-0-stragglers	16 blocks/s	350%	99MB/s
	32 blocks/s	591%	175MB/s
Ladon-1-straggler	16 blocks/s	195%	54MB/s
	32 blocks/s	432%	121MB/s

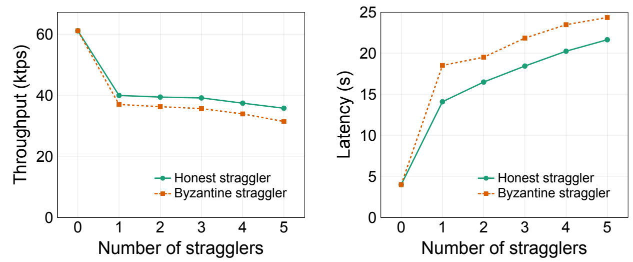
Fig. 6 evaluates the performance of Ladon-PBFT, ISS-PBFT, RCC-PBFT, Mir-PBFT, and DQBFT-PBFT with a varying number of stragglers. We use 16 replicas in these experiments, and vary the number of stragglers from 1 to 5. Fig. 6a shows that the throughput of Ladon-PBFT, ISS-PBFT, RCC-PBFT, Mir-PBFT, DQBFT-PBFT drop by 10%, 1%, 1%, 2% and 24%, from one straggler to 5 stragglers, respectively. In Fig. 6b, the latency increases slightly for all protocols. From Fig. 6, we observe the robustness of these protocols against the rise in straggler count. The throughput and latency largely remain steady despite the increasing number of stragglers. This is because the system performance is limited by the slowest straggler, as discussed in Sec. 2.1.

6.2.2 Performance in LAN. We evaluate the throughput and latency of Ladon and other protocols without stragglers and with one honest straggler in a LAN environment. The results are shown in Fig. 5e–Fig. 5h. All protocols exhibit trends similar to our WAN results (Fig. 5a–Fig. 5d), with higher throughput and reduced latency. Ladon shows comparable performance with ISS and RCC without stragglers and always outperforms other protocols with one straggler.

6.2.3 CPU and Bandwidth Analysis. To delve deeper into the performance bottlenecks of the protocols, we conducted an assessment of the CPU and bandwidth usage for both ISS and Ladon. The summarized results for 32 replicas in a WAN (16 blocks/s) and LAN (32 blocks/s) are presented in Table 1. Our findings indicate that neither ISS nor Ladon is constrained by CPU resources, as the maximum CPU usage observed is 800%, given that each instance is equipped with 8 vCPUs. Without stragglers, Ladon exhibits comparable bandwidth usage to ISS. With one straggler, Ladon generally experiences higher network bandwidth consumption and CPU usage compared to ISS.

6.3 Performance Under Faults

In this section, we study the performance of Ladon under Byzantine stragglers and crash faults in a WAN of 16 replicas.



(a) Throughput **(b)** Latency

Figure 7. Throughput and latency of Ladon with honest and Byzantine stragglers.

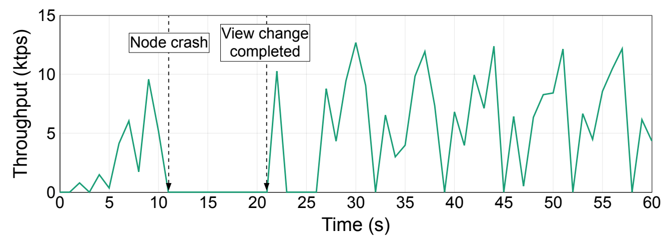


Figure 8. Ladon's throughput average (over 1s intervals) over time with one crash fault. The crash is at 11s, and the view change is completed at 21s.

6.3.1 Byzantine Stragglers. We study the impact on throughput and latency of Byzantine stragglers, with $f = 1$ up to the maximum tolerated number of $f = 5$ stragglers. Fig. 7 shows the impact of an increasing number of stragglers. Ladon with Byzantine stragglers reaches ≈ 90% of the throughput with honest stragglers. The latency increases by 12.5% with 5 Byzantine stragglers compared to the latency with 5 honest stragglers. These results indicate that the impact of Byzantine stragglers on the system's performance is only slightly more pronounced than that of honest stragglers. This is because the manipulation of ranks by Byzantine stragglers is limited, as discussed in Sec. 4.3.

6.3.2 Crash Failures. We study how crash faults affect the throughput of Ladon. The PBFT view change timeout is set at 10 seconds. Fig. 8 shows the throughput over time. A crash occurs in the first epoch at 11 seconds, causing the throughput to drop to 0. The view change process is initiated to handle the crash fault and is completed at 21 seconds, at which point the throughput begins to recover. This delays the epoch change. A new epoch starts at 26 seconds. Each subsequent short drop to 0 in throughput corresponds to an epoch change.

6.4 Causality Evaluation

We first define a metric, Inter-block Causal Strength (*CS*) and then use it to evaluate the causality of Ladon.

Inter-block Causal Strength *CS*. Assume that a series of n blocks $\{B_1, B_2, ..., B_n\}$ has been globally confirmed. For any

Table 2. Causal Strength (*CS*) of different protocols for different numbers of stragglers and proposal rates.

Stragglers settings	# Stragglers					Proposal rate (blocks/s)				
	1	2	3	4	5	0.5	0.4	0.3	0.2	0.1
Mir	0.154	0.042	0.012	0.004	0.002	0.241	0.204	0.174	0.148	0.154
ISS	1.04×10^{-5}	3.42×10^{-7}	6.79×10^{-10}	1.75×10^{-13}	1.83×10^{-16}	0.078	4.73×10^{-3}	1.36×10^{-4}	7.28×10^{-5}	1.04×10^{-5}
RCC	8.45×10^{-6}	2.48×10^{-7}	5.44×10^{-10}	9.87×10^{-14}	1.18×10^{-16}	0.076	2.49×10^{-3}	9.88×10^{-5}	5.07×10^{-5}	8.45×10^{-6}
DQBFT	1.15×10^{-5}	2.17×10^{-7}	9.74×10^{-10}	1.02×10^{-13}	6.85×10^{-17}	0.044	7.51×10^{-3}	9.15×10^{-4}	4.35×10^{-4}	1.15×10^{-5}
LADON	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

$i < j$, if B_i is generated after B_j is committed by $f+1$ replicas, we say a causality violation has occurred. The number of causality violations is denoted as N, and $CS = e^{-N/n}$.

The Inter-block Causal Strength (*CS*) is defined as a measure to evaluate the strength of causality of a system, with a value in (0, 1]. The closer the CS score is to 1, the stronger the system's causal property. A CS score of 1 implies that no one can front-run a partially committed transaction as discussed in Sec. 4.4.

Results. We evaluate the *CS* by varying the number of stragglers with a fixed proposal rate of 0.1 blocks/s and varying the proposal rate of one straggler. We conducted experiments in a WAN environment with 16 replicas. We now show results of LADON-PBFT (short for LADON in the following), which are similar to those of LADON-HotStuff.

Table 2 shows that LADON always exhibits strong causality across all straggler settings. By contrast, Mir, ISS, RCC, and DQBFT display a diminishing *CS* with an increasing number of stragglers, reflecting a weakening of the system's causal property. The *CS* of ISS, RCC, and DQBFT progressively decreases with the proposal rate. This trend suggests that these protocols are susceptible to the presence of stragglers, which weaken their causal properties.

We attribute the difference in causal strengths in Table 2 to the different ordering mechanisms. In the predetermined ordering, slow instances with straggling leaders might receive a global ordering for blocks earlier than other blocks in faster instances. This practice leads to causal violations, as it does not ensure that the global order aligns with the actual generation sequence, which is discussed in Sec. 4.4. Although DQBFT centralizes the global ordering, it fails to consider the causality between blocks. By contrast, LADON employs a dynamic ordering mechanism that respects the causality inherent in block generation. This design ensures that the global order of blocks corresponds to their actual generation sequence.

7 Related Work

Existing leader-based BFT protocols, such as Zyzzyva [28], PBFT [7], and HotStuff [38], suffer from the leader bottleneck. Many approaches to scaling leader-based BFT consensus have been proposed. These can be divided into three classes: parallelizing consensus, reducing committee size, and optimizing message transmission.

Parallelizing consensus. The idea in these approaches is that every replica acts as the leader to propose blocks, making all replicas behave equally. A representative method of this approach is Multi-BFT consensus [23, 35, 36], which runs several consensus instances in parallel to handle transactions. Stathakopoulou *et al.* [35] propose Mir-BFT, in which a set of leaders run the BFT protocol in parallel. Each leader maintains a partial log, and all instances are eventually multiplexed into a global log. To prevent malicious leaders, an epoch change is triggered if one of the leaders is suspected of failing. Byzantine leaders can exploit this by repeatedly ending epochs early to reduce throughput. Later, ISS [36] improved on Mir-BFT, by allowing replicas to deliver $\perp$ messages and instances to make progress independently. This improved its performance in the presence of crash faults. RCC [23] is another Multi-BFT protocol that operates in three steps: concurrent Byzantine commit algorithm (BCA), ordering, and execution. RCC adopts a wait-free mechanism to deal with leader failures, which does not interfere with other BCA instances. DQBFT [1] adopts a special order instance to globally order the output transactions from other parallel instances. Nonetheless, the system performance undergoes a significant decline if the ordering instance has a straggling leader. Additionally, the centralization of the ordering process makes it a prime target for attacks, resulting in elevated security vulnerabilities. Multi-BFT consensus is simple and has high performance in ideal settings. However, as analyzed in Sec. 2, Multi-BFT systems suffer from severe performance issues.

Similar to Multi-BFT consensus, DAG protocols [10, 25, 34] also parallelize consensus instances to improve the system scalability. However, in DAG protocols, each block has to contain at least $2f + 1$ references of predecessor blocks, instead of one reference in Multi-BFT consensus. We deliberately refrain from comparing our work with DAG-based systems due to fundamental differences in network assumptions and architectures, aligning with the standard practice in Multi-BFT research.

Reducing committee size. Another approach to improve BFT performance is to reduce the number of consensus participants, avoiding the leader bottleneck in large-scale settings. The representative solution is to randomly select a small group of replicas as the subcommittee, who are responsible for validating and ordering transactions. This solution has been developed by Algorand [18]. The sharding approach takes one step further and divides replicas into multiple disjoint subcommittees. Subcommittees run BFT protocols in parallel to process clients' transactions, improving on the efficiency of a single subcommittee. Many BFT sharding protocols, such as Elastico [30], OmniLedger [27] and RapidChain [39], have been proposed. However, subcommittees lower the system's tolerance to Byzantine replicas (*e.g.*, tolerating 25% Byzantine replicas in Algorand rather than 33%). The designs of these systems, especially state synchronization between subcommittees, are also more complex. Reducing committee size can significantly improve the scalability of BFT systems, however, it also weakens their fault tolerance and increases complexity.

Optimizing message transmission. Substantial work has also gone into improving network utilization. This line of work can be categorized according to the message transmission topology: structured and unstructured. A representative solution using unstructured transmission topology is gossip [5], in which a replica sends its messages to some randomly sampled replicas. Gossip has been used in Tendermint [4], Gosig [29], and Stratus [17], which can remove the leader bottleneck in large-scale settings. By contrast, in a structured topology, each replica sends its messages to a fixed set of replicas. For example, in Kauri [32], replicas disseminate and aggregate messages over trees, and in Hermes [24] the leader sends blocks to a committee, which helps to relay the block and vote messages. These approaches work well at scale (e.g., thousands of replicas) and have a high overhead at smaller scales, which are the focus of our work. Despite improving performance at scale, these approaches have high overhead in smaller deployments.

8 Conclusion

We propose Ladon, a Multi-BFT protocol that mitigates the impact of stragglers on performance. We propose dynamic global ordering to assign a global ordering index to blocks according to the real-time status of all instances, which differs from prior work using pre-determined global ordering. We decouple the dependencies between the various partial logs to the maximum extent to ensure a fast construction of the global log. We also design monotonic ranks, pipeline their formulation with the consensus process, and adopt aggregate signatures to reduce rank data. We build Ladon-PBFT and Ladon-HotStuff prototypes, conducting comprehensive experiments on Amazon AWS to compare them with existing Multi-BFT protocols. Our evaluation shows that Ladon has significant advantages over ISS, RCC, Mir, and DQBFT in the presence of stragglers.

References

[1] Balaji Arun and Binoy Ravindran. Scalable Byzantine fault tolerance via partial decentralization. In *VLDB*, 2022.

[2] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. FnF-BFT: Exploring performance limits of BFT protocols. In *arXiv preprint arXiv:2009.02235*, 2020.

[3] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. SoK: Mitigation of front-running in decentralized finance. In *Cryptology ePrint Archive*, 2021.

[4] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. In *M.Sc. Thesis, University of Guelph, Canada*, 2016.

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

[6] Miles Carlsten, Harry Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the instability of Bitcoin without the block reward. In *CCS*, 2016.

[7] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[8] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, Mirco Marchetti, et al. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[9] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[10] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *EuroSys*, 2022.

[11] Sisi Duan, Michael K Reiter, and Haibin Zhang. Secure causal atomic broadcast, revisited. In *DSN*, 2017.

[12] Sisi Duan, Michael K Reiter, and Haibin Zhang. BEAT: Asynchronous BFT made practical. In *CCS*, 2018.

[13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. In *JACM*, 1988.

[14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Commun.*, 2018.

[15] Chen Feng and Jianyu Niu. Selfish mining in Ethereum. In *ICDCS*, 2019.

[16] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. Dissecting the performance of chained-BFT. In *ICDCS*, 2021.

[17] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Devouring the leader bottleneck in BFT consensus. In *arXiv preprint arXiv:2203.05158*, 2022.

[18] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *SOSP*, 2017.

[19] Abraham I Giridharan N, Howard H. No-commit proofs: Defeating livelock in BFT. In *Cryptology ePrint Archive*, 2021.

[20] Tiantian Gong, Mohsen Minaei, Wenhai Sun, and Aniket Kate. Towards overcoming the undercutting problem. In *FC*, 2022.

[21] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *EuroSys*, 2010.

[22] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous BFT protocols. In *CCS*, 2020.

[23] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: Resilient concurrent consensus for high-throughput secure transaction processing. In *ICDE*, 2021.

[24] Mohammad M. Jalalzai, Chen Feng, Costas Busch, Golden G. Richard, and Jianyu Niu. The hermes BFT for blockchains. In *TDSC*, 2022.

[25] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *PODC*, 2021.
[26] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for Byzantine consensus. In *CRYPTO*, 2020.
[27] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *S&P*, 2018.
[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *OSR*, 2007.
[29] Peilun Li, Guosai Wang, Xiaoqi Chen, Fan Long, and Wei Xu. Gosig: a scalable and high-performance Byzantine consensus for consortium blockchains. In *SoCC*, 2020.
[30] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *CCS*, 2016.
[31] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. In *Working Paper*, 2008.
[32] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *SOSP*, 2021.
[33] Malay Kumar Patra, Subhendu Rakshit, and Neeraj Kumar Sharma. On the security of Byzantine fault tolerant consensus protocols against front-running attacks. In *ICDCN*, 2017.
[34] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *CCS*, 2022.
[35] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-BFT: Scalable and robust BFT for decentralized networks. In *JSys*, 2022.
[36] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *EuroSys*, 2022.
[37] Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. Adding fairness to order: Preventing front-running attacks in BFT protocols using TEEs. In *SRDS*, 2021.
[38] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.
[39] M Zamani, R Jurdak, Y Liu, B O'Flynn, and P Dutta. Rapidchain: A fast and scalable blockchain protocol. In *PerCom*, 2018.

A Message Complexity Analysis

We focus on the message complexity of Ladon-PBFT and Ladon-opt in the normal case operations because the introduced dynamic global ordering does not affect the view-change mechanism. The message complexity measures the expected number of messages that honest replicas generate during the protocol execution [22]. Specifically, we take PBFT as a baseline to illustrate the overhead of dynamic global ordering in Ladon.

1) PBFT. In the Pre-prepare phase, an honest leader broadcasts a message to all backup replicas, resulting in a complexity of $O(n)$. In the Prepare and Commit phases, which necessitate all-to-all communications among honest nodes, the complexity escalates to $O(n^2)$. Therefore, PBFT exhibits a message complexity of $O(n^2)$.

2) Ladon-PBFT. We analyze the message complexity of a consensus instance using PBFT in Ladon-PBFT. In the Pre-prepare phase, messages include a list containing $2f+1$ rank information, resulting in a $O(n^2)$ complexity. The Prepare phase aligns with PBFT, also reflecting $O(n^2)$ complexity. However, in the Commit phase, each backup communicates its highest rank to the leader, resulting in an additional all-to-one communication with a complexity of $O(n)$. Consequently, the Commit phase manifests a combined complexity of $O(n^2+n) \approx O(n^2)$.

3) Ladon-opt. In the Pre-prepare phase, the $2f+1$ rank information, previously $O(n)$, is condensed into a singular $O(1)$ complexity, thereby reducing the overall complexity to $O(n)$. The Prepare and Commit phases in the Ladon-opt remain consistent with Ladon-PBFT.

To sum up, Ladon-PBFT increases the message complexity of the Pre-prepare phase from $O(n)$ to $O(n^2)$. But, both Ladon-PBFT and Ladon-opt do not increase the overall message complexity of the protocol.

Authenticator complexity. In addition to message complexity, recent work [38] introduces a concept called authenticator complexity (also known as signature complexity) to quantify the number of cryptographic operations (such as verifications) done by replicas. Since in Ladon, every message contains at least one signature, the authenticator complexity also reflects the message complexity. In the Pre-prepare phase of PBFT, each backup replica verifies $O(1)$ signatures, whereas, in Ladon-PBFT, it verifies an additional $O(n)$ signatures. By contrast, the Ladon-opt employs aggregate signatures to optimize this to $O(1)$. Likewise, during the Commit phase, the leader verifies an extra $O(n)$ signatures in Ladon. Thus, considering the verification of $O(n)$ signatures in commit messages, the Commit phase manifests a combined complexity of $O(2n) \approx O(n)$.

B Illustrative Example of Protocol Behaviors

Here we provide a running example to illustrate the Ladon protocol behaviors under normal case and with a Byzantine leader.

Consider the example shown in Fig. 9, where we have four replicas: r_0, r_1, r_2, and r_3. In this scenario, r_0, r_1, and r_2 are the leaders of Instance 0, 1, and 2, respectively. With $f=1$, there are $3f+1=4$ replicas in total. During the commit phase of B_1^2, each replica r_i sends its highest known rank, $rank_i$, to r_2. Suppose the ranks received by r_2 are $\{3,2,2,2\}$. At time t_1, r_2 is about to generate a new block for Instance 2.

Case 1: If r_2 is honest, it will select the highest rank from the set and add one, resulting in $B_2^2.rank=4$.

Case 2: If r_2 is detected as a Byzantine leader, a view-change protocol will be initiated to replace r_2 with another replica as the leader of Instance 2, assumed here to be r_3, which is honest. Upon taking over, the new leader r_3 will collect the rank set $\{3,2,2\}$ from r_0, r_1, and itself, respectively. r_3 will then select the highest rank from the set and add one, resulting in $B_2^2.rank=4$.

Case 3: If r_2 is Byzantine and wishes to avoid detection, it can discard the highest rank ($rank_0=3$ from r_0), as it only needs $2f+1$ ranks to proceed. By selecting the highest rank from the remaining set $\{2,2,2\}$ and adding one, it would set $B_2^2.rank=3$.

C Proof of Correctness

We now provide proof and analysis of Ladon-PBFT protocol properties including the correctness of monotonic ranks and security properties of Ladon.

C.1 Correctness of Monotonic Rank

We prove that monotonic ranks in Ladon satisfy the two properties: MR-Agreement and MR-Monotonicity.

Lemma 1. *If an honest replica partially commits a block B then all honest replicas eventually partially commit B.*

Proof. If an honest replica r partially commits a block B, by SB-Termination, all honest replicas will partially commit a block for $B.round$. By SB-Agreement, if an honest replica r' partially commits a block b' for $B.round$, then $B'=B$. Thus, all honest replicas eventually partially commit B. ◻

Theorem 1 (MR-Agreement)**.** *All honest replicas have the same rank for a partially committed block.*

Proof. When a leader proposes a block B, it determines block parameters and formats B as $B=\langle txs, index, round, rank\rangle$. If an honest replica r partially commits the block B, all other honest replicas eventually partially commit B (Lemma 1). As $B.rank$ is one of the parameters of block B, all honest replicas have the same rank for block B. The proof is done. ◻

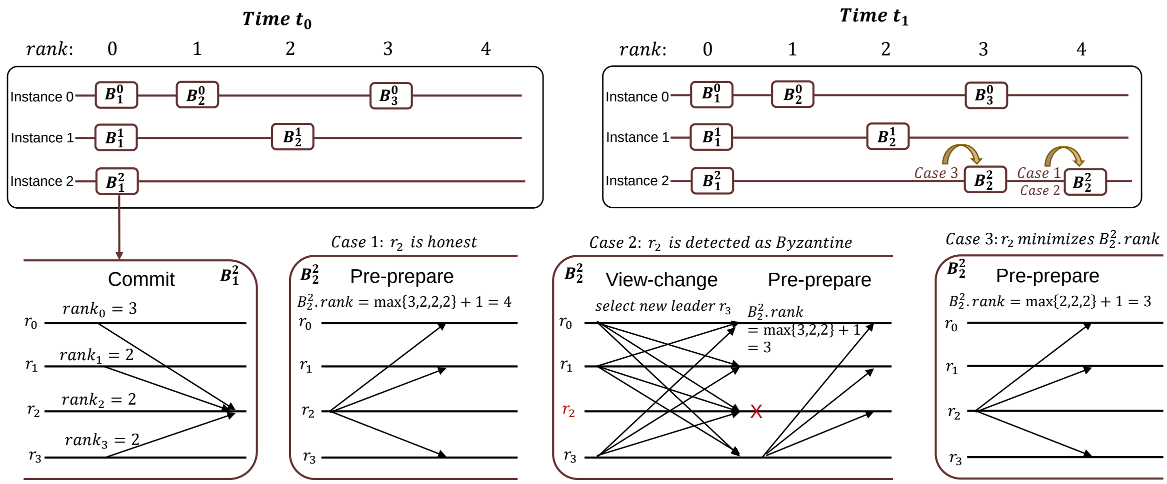


Figure 9. Illustrative example of LADON with four replicas and three instances under three cases: (1) with an honest leader, (2) with a detectable Byzantine leader, and (3) with an undetectable Byzantine leader who minimizes the rank for the newly proposed block.

Theorem 2 (MR-Monotonicity). *If a block B' is generated after an intra-instance (or a partially committed inter-instance) block B, then the rank of B' is larger than the rank of B.*

Proof. When block B is partially committed inter-instance block, $B.rank$ is also agreed by a quorum Q of at least $2f+1$ replicas by the MR-Agreement property. Later, when a block B' is generated, its rank is set according to the highest ranks collected from a quorum Q' of at least $2f+1$ replicas. Since there are $n = 3f+1$ replicas in total, the intersection $Q \cap Q'$ ensures that at least one honest replica in Q will report its highest rank $rank_m$, and $rank_m \geq B.rank$. By Algorithm 1, $B'.rank = rank_m + 1$, which is larger than $B.rank$, and satisfies the monotonicity property.

When block B is an intra-instance block, according to Algorithm 2, the leader will collect $2f+1$ ranks for B' at the commit phase of B, at which point the prepare phase has been finished and the leader has update its highest rank $curRank.rank$ if it is smaller than $B.rank$ (Lines 23-26). When propose B', the leader will select the highest rank $rank_m$ from the $2f+1$ collected ranks, which contain the highest rank from itself. So we have $rank_m \geq B.rank$, $B'.rank = rank_m + 1 > B.rank$., which satisfies the monotonicity property. □

C.2 System Security Properties

Lemma 2. *The ranks for blocks generated by the same BFT instance is strictly increasing, i.e., $B^i_{j+1}.rank > B^i_j.rank$.*

Proof. Assuming block B^i_j is in epoch e. By Algorithm 2, at the commit phase of B^i_j, an honest replica will update its highest rank value, denoted by $curRank.rank$, if $B^i_j.rank > curRank.rank$ (Lines 23-25). Then the replica will send its highest rank value $curRank.rank \geq B^i_j.rank$ to the leader (Line 27). Upon proposing B^i_{j+1}, the leader will collect a set of $2f+1$ $rank$ values from different replicas (Line 1), denoted as $rankSet$, among which at least $f+1$ $rank$s are from those honest replicas. The highest rank that the leader gets from the $rankSet$ will be $rank_m \geq B^i_j.rank$. By Line 6, if $rank_m + 1 \leq maxRank(e)$, $B^i_{j+1}.rank = rank_m + 1 > B^i_j.rank$; otherwise $B^i_{j+1}.rank = maxRank(e)$.

When $B^i_{j+1}.rank = maxRank(e)$, we discuss two different scenarios. If $B^i_j.rank = maxRank(e)$, the leader will stop proposing for epoch e after it proposes B^i_j, and B^i_{j+1} will be in epoch $e+1$. We have the $minRank(e+1) = maxRank(e) + 1$, *i.e.*, the minimum rank of a block in the e+) epoch is one more than the maximum rank of the blocks in the epoch e, thus $B^i_{j+1}.rank > B^i_j.rank$. If $B^i_j.rank < maxRank(e)$, $B^i_{j+1}.rank = maxRank(e) > B^i_j.rank$.

In summary, $B^i_{j+1}.rank > B^i_j.rank$. □

Lemma 3. *If an honest replica partially commits a block B, it will eventually globally confirm B.*

Proof. If an honest replica r partially commits a block B in epoch e, it will be added to the input of the partial ordering layer $\mathcal{G}_{in}$ (Algorithm 2, Line 34), which triggers Algorithm 1.

In Algorithm 1, a replica r first fetches the last partially confirmed block from each instance, denoted by the set $\mathcal{S}'$ (Line 2), and finds the block with the lowest ordering index among the blocks in $\mathcal{S}'$, denoted by B^* (Line 3). Then r computes the bar as a tuple of $(B^*.rank + 1, B^*.index)$ (Line 4). Thereafter, r will transverse all the blocks that have been partially committed by not globally confirmed yet, and global confirms all the blocks that have a lower ordering index than the bar (Lines 5-11). Thus, B will be globally confirmed if it has a lower ordering index than the bar.

Now we prove that B will eventually have a lower ordering index than the *bar*. If the leader of an instance is Byzantine, it will be detected by backups, and a view-change protocol is triggered to change the leader. Since there are more than 2/3 honest replicas, there will eventually be an honest leader for the instance. If the leader of an instance is honest, it will continuously output partially committed blocks with increasing rank values (Lemma 2). Since B^* represents the block with the lowest ordering index among the last partially confirmed blocks from each instance, eventually $B^*.rank \geq B.rank$. Thus the *rank* of the *bar* will be greater than $B.rank$, which means B will have a lower ordering index than the *bar* (In Ladon, blocks are ordered by increasing *rank* values).

In summary, block B will eventually be globally confirmed by r. □

Theorem 3 (Totality). *If an honest replica globally confirms a block B, then all honest replicas eventually globally confirm the block B.*

Proof. If an honest replica globally confirms B, it must have partially committed B. By Lemma 1, all honest replicas eventually partially commit B. By Lemma 3, all honest replicas eventually globally confirm B. □

In the following text, we build on the concept of a global ordering index sn. In Algorithm 1, global ordering refers to the act of appending a block to $\mathcal{G}_{out}$, which functions as a global log. The global ordering index sn indicates the position of the block within the log, starting at 0. With the addition of each block, the global ordering index sn grows by one.

Lemma 4. *An honest replica uniquely maps a block B to a single global ordering index sn.*

Proof. We first prove a block B will be assigned a single global ordering index. In Algorithm 1, the set $\mathcal{S}$ represents the set of blocks waiting to be globally ordered (Line 5). A block will be moved out of $\mathcal{S}$ once it is globally ordered (Line 9). Thus, one block will only have one global ordering index.

Next, we prove that a single global ordering index corresponds to only one block. Assuming an honest replica partially commits two blocks B and B'. In Algorithm 1, if B and B' are both in the set $\mathcal{S}$ (Line 5), only one of them can become the candidate block during a single round of searching (Line 6) and subsequently be appended to the global log (Line 8). The latter block will be assigned a greater global ordering index. If B' is added to $\mathcal{S}$ after B has been globally confirmed, B' will be appended to the global log at a later point and consequently receive a greater global ordering index than B. Therefore, B and B' will have distinct global ordering indexes.

This concludes the proof, demonstrating that a single global ordering index uniquely corresponds to a specific block in the global log of an honest replica. □

Theorem 4 (Agreement). *If two honest replicas globally confirm $B.sn = B'.sn$, then $B = B'$.*

Proof. We prove this statement by induction, considering the global block sequence from global ordering index 0 to sn for two honest replicas r and r', denoted as $L(sn)$ and $L'(sn)$, respectively. Here, $L(-1)$ signifies the initial state of the block sequence with no blocks.

First, we establish the base case $L(-1) = L'(-1) = \emptyset$. Then we provide the inductive step. Assuming $L(sn-1) = L'(sn-1)$, we aim to show that $L(sn) = L'(sn)$.

Assume r and r' globally confirm two different blocks B and B', respectively, with the same global ordering index sn. Since replica r globally confirms B with sn, we have 1) r' will globally confirm B (Theorem 3); 2) $B \notin L(sn-1)$ (Lemma 4). Thus $B \notin L'(sn-1)$. Now, if r' globally confirms B with the global ordering index sn', by Lemma 4, we know $sn' \neq sn$, hence $sn' > sn$. Similarly, if r globally confirms B' with the global ordering index sn'', we have $sn'' > sn$.

According to the global ordering rules, for replica r', if $sn'' > sn$, $B.rank > B'.rank$ or $B.rank = B'.rank$ and $B.index > B'.index$; for replica r, if $sn' > sn$, $B'.rank > B.rank$ or $B'.rank = B.rank$ and $B'.index > B.index$. However, this leads to a contradiction to the assumption that $B \neq B'$. This concludes the proof, demonstrating that if two honest replicas globally confirm $B.sn = B'.sn$, then $B = B'$ holds true. □

Lemma 5. *If a correct client broadcasts a transaction tx, some honest replica eventually proposes B with $tx \in B.txs$.*

Proof. Assuming tx is assigned to buckets $\hat{b}$, and $\hat{b}$ is assigned to instance i. If the leader of the instance i refuses to propose tx, tx will be left in $\hat{b}$. According to the bucket rotation policy [36], $\hat{b}$ will be reassigned to different instances an infinite number of times. If the leader of an instance is Byzantine, it will be detected by backups, and a view-change protocol is triggered to change the leader. Since there are more than 2/3 honest replicas, there will eventually be an honest leader for the instance. Thus, $\hat{b}$ will be eventually assigned to an instance with an honest leader, who will propose a block containing tx. □

Theorem 5 (Liveness). *If a correct client broadcasts a transaction tx, an honest replica eventually globally confirms a block B that includes tx.*

Proof. If a correct client broadcasts a transaction tx, by Lemma 5, some honest replica eventually proposes B with $tx \in B.txs$. By SB-Termination, the replica eventually partially commits B. By Lemma 3, the replica eventually globally confirms B. □

D Ladon with Chained HotStuff

We now describe Ladon with the chained HotStuff [38] consensus instances (Ladon-HotStuff).

D.1 Data Structures

Tree and branches. Each replica stores a tree of pending batches of transactions submitted by clients as its local data structure. Each tree node contains a proposed batch, metadata associated with the protocol, and a parent link. The branch led by a given node is the path from the node all the way to the tree root by visiting parent links. Two branches are conflicting if neither one is an extension of the other. Two nodes are conflicting if the branches led by them are conflicting.

Messages. The general messages have a format of the tuple $\langle\langle type, view, index, rank, node, parentQC\rangle_\sigma, rank_m, rankQC\rangle$, where $type \in \{$GENERIC, VOTE, NEW-VIEW$\}$, *node* contains a proposed node (the leaf node of a proposed branch), *parentQC* is used to carry the *QC* for the parent node, $rank_m$ and *rankQC* are the highest rank known by the message sender and the QC for it, respectively. $\langle msg\rangle_\sigma$ is the signature of message *msg*. We use *genmsg*, *votemsg*, and *nvmsg* as shorthand for the generic messages, vote messages, and new-view messages, respectively.

D.2 Algorithm Description

As shown in Algorithm 3, the Chained HotStuff consensus protocol simplifies the algorithm into a two-phase process: proposal and voting.

Proposal phase. Upon receiving $2f + 1$ *votemsg* from the previous view (Line 1), a leader generates the QC for the last node using GENERATEQC (Line 3) and proposes a new node extending the last node (Line 5). The leader sets the *rank* of the new node as the highest rank it knows plus one. If the *rank* is greater than the $maxRank(e)$ of the current epoch, it is set to $maxRank(e)$ (Line 6). Then, the leader generates a *genmsg* and broadcasts it to all replicas (Lines 7-18). If the *rank* of the new proposal is equal to $maxRank(e)$, the leader stops proposing (Lines 9-10)[4].

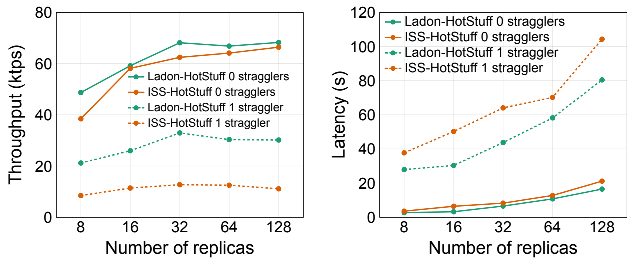
Voting phase. Upon receiving the *genmsg* from the leader, a replica validates it and updates its *currentRank* if the $rank_m$ is greater than the highest rank it knows (Lines 15-17). The replica then votes for it if it extends the highest node in the replica's view by sending a *votemsg* to the leader (Lines 25-26). Finally, the replica will check the commit rule to decide whether there is a node that could be committed.

Commit rule. Chained HotStuff introduces a specific commit rule based on its chain structure: a node is committed when its 3-chain successor has received votes from a quorum of replicas. In the context of the Chained HotStuff commit rule, a 3-chain successor to a node refers to the node that is three positions ahead in the chain. For instance, if you have a chain of nodes $A \to B \to C \to D \to E$, node B is the

[4]To ensure that all blocks can be delivered, we extend each partial log with 3 dummy blocks which are not added to the global log

Algorithm 3 The LADON-HotStuff Algorithm for Instance i at View v and Epoch e

▷ as a leader
1: **upon** receive $2f + 1$ *votemsg* **do**
2: $voteSet \leftarrow 2f + 1$ *votemsg*
3: $QC \leftarrow$ GENERATEQC$(voteSet)$
4: $txs \leftarrow$ CUTBATCH$(ins.bucketSet)$
5: $proposal \leftarrow$ CREATELEAF$(QC.node, txs)$
6: $rank \leftarrow min\{curRank.rank + 1, maxRank(e)\}$
7: $genmsg \leftarrow \langle\langle$GENERIC$, v, i, rank, proposal, QC\rangle_\sigma, curRank.rank, curRank.QC\rangle$
8: broadcast $\langle genmsg, voteSet\rangle$
9: **if** $rank = maxRank(e)$
10: stop propose
11: **end if**
12:
▷ as a replica
13: **upon** receive $\langle genmsg, voteSet\rangle$ from leader **do**
14: **if** VERIFY$(\langle genmsg, voteSet\rangle)$
15: **if** $genmsg.rank_m > curRank.rank$
16: $curRank.rank \leftarrow genmsg.rank_m$
17: $curRank.QC \leftarrow genmsg.rankQC$
18: **end if**
19: $B^* \leftarrow genmsg.node$
20: $B'' \leftarrow B^*.QC.node$
21: $B' \leftarrow B''.QC.node$
22: $B \leftarrow B'.QC.node$
23: **end if**
24: **if** CHECKNODE
25: $votemsg \leftarrow \langle\langle genmsg\rangle_\sigma, curRank.rank, curRank.QC\rangle$
26: send *votemsg* to leader
27: **end if**
▷ start PRECOMMIT phase on B^*'s parent
28: **if** $B^*.parent = B''$
29: $genericQC \leftarrow B^*.QC$
30: **end if**
▷ start COMMIT phase on B^*'s parent
31: **if** $(B^*.parent = B'') \land (B''.parent = B')$
32: $lockedQC \leftarrow B''.QC$
33: **end if**
▷ start DECIDE phase on B^*'s parent
34: **if** $(B^*.parent = B'') \land (B''.parent = B') \land B'.parent{=}B$
35: $\mathcal{G}_{in} \leftarrow \mathcal{G}_{in} \cup B$ //Commit B
36: **end if**
37:
▷ as a leader
38: **upon** receive *votemsg* **do**
39: **if** VERIFY$(votemsg) \land votemsg.rank_m > curRank.rank$
40: $curRank.rank \leftarrow votemsg.rank_m$
41: $curRank.QC \leftarrow votemsg.rankQC$
42: **end if**



(a) Throughput **(b)** Latency

Figure 10. Throughput and latency of Ladon-HotStuff and ISS-HotStuff with a varying number of replicas.

3-chain predecessor of node E. For example, if a node B in the chain has a 3-chain successor E and E receives a quorum of votes, then B is committed.

View change. Similar to PBFT, HotStuff instantiates a view change when the current leader fails to propose a node within a certain period (this duration is typically predefined), or if the leader is suspected of being faulty. Replicas use a new-view message to carry the highest *genericQC* and send it to the leader.

D.3 Performance Evaluation

Fig. 10 shows the throughput and latency of Ladon-HotStuff and ISS-HotStuff with one honest straggler and without stragglers. The total block rate is set as $16 blocks/s$. Fig. 10a shows the throughput of Ladon-HotStuff without stragglers is comparable with that of ISS-HotStuff, while the throughput of Ladon-HotStuff with one straggler is 2.7× compared to that of ISS-HotStuff on 128 replicas. In Fig. 10b, as the number of replicas scales up, the latency of both protocols increases. This is because we limit the total block rate of the whole system. The latency of Ladon-HotStuff without stragglers is comparable with that of ISS-HotStuff, while the latency of Ladon-HotStuff with one straggler is 22.9% lower than that of ISS-HotStuff on 128 replicas.

We note that the performance of Ladon-HotStuff is more affected by stragglers compared to Ladon-PBFT. This is due to the use of chained HotStuff, where the commitment of a block is pipelined with the subsequent blocks. Consequently, blocks in a slow instance are committed much more slowly than those in PBFT, leading to reduced throughput and increased latency.