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

Hanzheng Lyu*
hzlyu@student.ubc.ca
University of British Columbia (Okanagan campus)

Shaokang Xie*
12011206@mail.sustech.edu.cn
Southern University of Science and Technology

Jianyu Niu†
niujy@sustech.edu.cn
Southern University of Science and Technology

Chen Feng
chen.feng@ubc.ca
University of British Columbia (Okanagan campus)

Yinqian Zhang†
yinqianz@acm.org
Southern University of Science and Technology

Ivan Beschastnikh
bestchai@cs.ubc.ca
University of British Columbia (Vancouver campus)

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. Ladon's 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 protocols. 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.

*Both authors contributed equally to this research. This work was done in part while Hanzheng Lyu was a visiting student at SUSTech.

†Jianyu Niu and Yinqian Zhang are affiliated with Research Institute of Trustworthy Autonomous Systems and Department of Computer Science and Engineering of SUSTech. Corresponding author: Jianyu Niu.

EuroSys'25, March 30–April 03, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03.
https://doi.org/10.1145/3689031.3696102

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

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

ACM Reference Format:
Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Chen Feng, Yinqian Zhang, and Ivan Beschastnikh. 2025. Ladon: High-Performance Multi-BFT Consensus via Dynamic Global Ordering. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696102

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 [8] or HotStuff [39], adopt a leader-based scheme [8, 10, 23], 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, the leader can become a significant performance bottleneck, especially at scale. The leader's workload increases linearly with the number of replicas [2, 18, 24, 36, 37], making the leader the dominant factor in the system's throughput and latency.

To address the leader bottleneck, Multi-BFT systems [2, 24, 36, 37] have emerged as a promising alternative. Multi-BFT consensus runs multiple leader-based BFT instances in *parallel*, as shown in Fig. 1. A replica may participate as a leader in one BFT instance and as a backup in 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

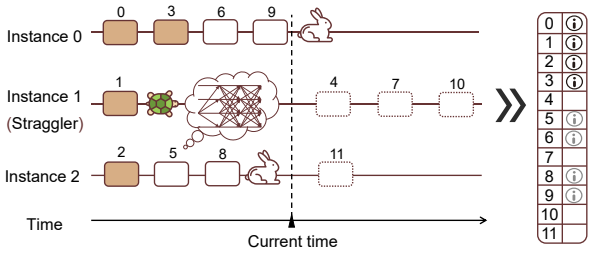


Figure 1. An overview of Multi-BFT paradigm. The shaded (resp., white) blocks refer to the globally confirmed (resp., partially committed) blocks. The dashed blocks refer to the blocks to be produced in the future.

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, 24, 36, 37] 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). For example, 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.

In a decentralized system, this simple global ordering method has performance issues. A slow leader, often called a straggler, can slow down not just one instance, but the entire system. For example, Instance 1 in Fig. 1 has a straggling leader that 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. (We provide further theoretical analysis and detailed experimental results to illustrate the impact of stragglers in Sec. 2.1). This challenge is prevalent in decentralized systems where variations in replica performance and network conditions make straggling leaders commonplace. Even worse, an adversary can violate the causality of blocks across instances, allowing it to front-run [3, 15] its transactions ahead of others to gain an unfair advantage in applications like auctions and exchanges. (See more discussion in Sec. 4.3.)

In this paper, we propose Ladon[1], a high-performance Multi-BFT consensus protocol that considers instances' varying block rates. Our insight 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 subsequently generated blocks are always larger than the rank of a partially committed block to respect block causality. Here, the agreement property ensures that replicas that use a deterministic algorithm to order partially committed blocks by their ranks can achieve the same ordering sequence, while monotonicity preserves inter-block causality.

The above dynamic global ordering decouples the dependency between the replicas' partial logs to ensure fast generation of the global log, eliminating the straggler impact. It also orders blocks by their generation sequence, preserving inter-block causality. 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. The dynamic ranks enable Instance 1 to quickly synchronize with other instances, alleviating 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 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 message complexity.

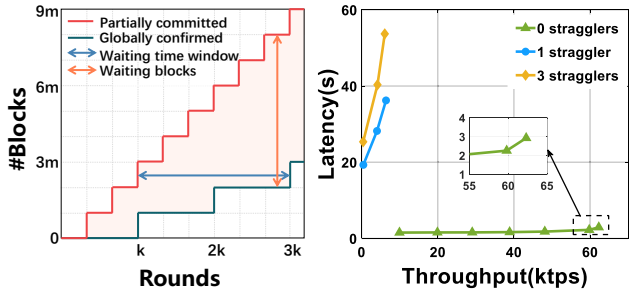
We built end-to-end prototypes of Ladon with PBFT and HotStuff, denoted as Ladon-PBFT and Ladon-HotStuff, respectively. Furthermore, we believe Ladon can compose with any single-leader BFT protocol. We conduct extensive experiments on AWS to evaluate and compare Ladon with existing Multi-BFT protocols, including ISS [37], RCC [24], Mir [36], 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.

2 Existing Multi-BFT Susceptibility

2.1 Performance Degradation

Existing Multi-BFT protocols [2, 24, 36, 37] with the predetermined global ordering perform well when all instances have the same block production rate. With m instances, they

[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.

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 [37], a state-of-the-art Multi-BFT protocol, in which consensus instances are instantiated with PBFT [8]. 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, 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 detect-and-replace 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.* [36] propose to use the timeout mechanism to replace leaders of the instances that do not timely output partially committed blocks. Similarly, in RCC [24], 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 slow down 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 itself become a performance bottleneck with a straggling leader. And, the leader in this special instance can also maliciously 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.* [14], which is widely used in BFT consensus [8, 39]. 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 [8, 39], we assume that Byzantine replicas aim to destroy the *safety* and *liveness* properties by deviating from the protocol. They may strategically delay their operation (e.g., without triggering timeout mechanisms), appearing as stragglers to either compromise performance or violate block causality (Sec. 4.3). It is worth noting that in decentralized applications, due to replicas' heterogeneous capacities, honest replicas may also behave as stragglers.

3.2 Preliminaries

We now introduce some building blocks for our work.

Sequenced Broadcast (SB). We follow ISS [37] 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 $(msg, r) \in M \times R$. 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 a *quiet* sender. SB is implementable with consensus, Byzantine reliable broadcast (BRB), and a Byzantine fault detector [37]. 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.

Aggregated signature scheme. The aggregated signature is a variant of the digital signature that supports aggregation [4]. That is, given a set of users R, each with a signature σ_r on the message m_r, the generator of the aggregated signature can aggregate these signatures into a unique short signature: $\text{AGG}(\{\sigma_r\}_{r \in R}) \rightarrow \sigma$. Given an aggregation signature, the identity of the aggregation signer r and the original message m_r of the signature can be extracted. The verifier can also verify that r signed message m_r by using the verify function: $\text{VERIFYAGG}((pk_r, m_r)_{r \in R}, \sigma) \rightarrow 0/1$.

3.3 Problem Formulation

We consider a Multi-BFT system consisting of m BFT instances indexed from 0 to $m-1$. Thus, the ith $(1 \leq i \leq m)$ 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 [36]), 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 in 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 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. MR-Monotonicity guarantees block causality such that if a block is partially committed before another block is generated, then the partially committed block is globally confirmed before the latter. While a trusted global time could further strengthen block causality [27], the current system still provides a robust level of causality, even without such global time coordination.

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. 4.4.

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. 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. Specifically, a block only needs to include one aggregated signature as a certificate of the collected ranks.

Algorithm 1 The Global Ordering Algorithm

```
1:  upon G_in is updated
2:      S' ← getLastBlock(G_in)
3:      B* ← findLowestBlock(S')
4:      bar = (B*.rank + 1, B*.index) //compute the bar
5:      S ← G_in \ G_out
6:      B_can = findLowestBlock(S) //find candidate block
7:      while B_can ≺ bar //B_can has a lower index than bar
8:          G_out ← G_out ∪ B_can //globally confirm B_can
9:          S ← S \ B_can //update S
10:         B_can = findLowestBlock(S) //find next B_can
11:     end while

    //Return block with the lowest ordering index
12: function findLowestBlock(V)
13:     B* ← first block in V
14:     for each B ∈ V do
15:         if B ≺ B*
16:             B* ← B
17:         end if
18:     end for
19:     return B*
```

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

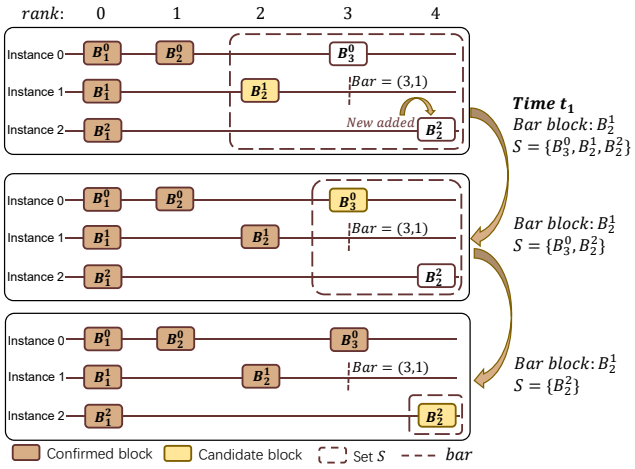


Figure 3. An illustration of the dynamic global ordering process. At time t_1, a new block B_2^2 is partially committed, which makes blocks B_2^1 and B_3^0 globally confirmed.

$\mathcal{S}'$, *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 initialized with $(0, 0)$.

With *bar* defined, 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 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 but a smaller index than *bar* and so it will be globally confirmed. Finally, the set $\mathcal{S}$ contains only 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 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 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 previous Multi-BFT protocols (with a pre-determined ordering) [2, 24, 36, 37]. To better understand the issues in existing Multi-BFT protocols, 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, 15], undercutting attacks [22], and incentive-based attacks [7, 16, 17]. For example, consider a front-running attack [3, 15] of cryptocurrency exchange, 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 client-side causality in prior work [12, 13, 38]. Achieving this strong causality usually requires complicated fair-ordering mechanisms [9], costly cryptographic techniques (*e.g.*, commitment [12] and threshold cryptography [13]) or Trusted hardware (*e.g.*, TEEs [38]). We leave it as future work to use these techniques in Ladon.

4.4 Analysis of Byzantine leaders' Impact

While the dynamic global ordering effectively mitigates stragglers' impact on performance as well as enhancing the inter-block causality, challenges arise with Byzantine leaders, who may intentionally delay block proposals or strategically minimize ranks of proposed blocks.

Strategic delay of block proposals. A Byzantine leader can strategically delay the proposal of a new block, impacting system latency. Additionally, such delays can be exploited to gain a front-running advantage or more fees from proposing transactions in some applications like blockchains. This challenge is inherent to BFT systems (including single instance and Multi-BFT systems), as delayed block proposals are a common Byzantine behavior that is hard to differentiate from honest actions in consensus protocols. The impact

of this strategy is constrained by a timeout mechanism. In each round, the leader is given a limited window of time to propose a block. If the leader fails to do so within this period, a timeout triggers, preventing excessive delays and allowing the system to move forward, typically by initiating a view change to replace the leader.

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 (see detailed example in Appendix B of [32]).

To analyze this, consider 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 (see detailed description in Sec. 5.2.2), 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. Therefore, even with rank manipulation, the leader's proposed rank will not be lower than the ranks of all partially committed blocks.

Conclusion. Both Byzantine strategies will have a limited impact on system performance and transaction causality, as their effects are mitigated by built-in timeout mechanisms and rank certification.

5 Ladon Design

We overview 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 message complexity in Sec. 5.3. We sketch the correctness analysis of Ladon in Sec. 5.4. Additionally, we provide an analysis of message complexity, examples of protocol behavior, and the design for composing Ladon with HotStuff; these are detailed in Appendices A, B, and D in [32] due to space constraints.

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 [37], while instance consensus and global ordering are tailored to realize the dynamic global ordering.

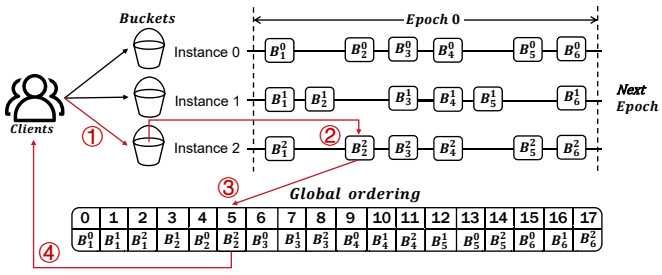


Figure 4. An overview of Ladon's design. Client transactions are enqueued into rotating buckets and are then consumed by concurrent BFT protocol instances. Each instance packs transactions into blocks that are eventually totally ordered. Instances execute in epochs.

Rotating buckets. Ladon adopts rotating buckets from ISS [37]. 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 assigned round-robin 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. We detail the Epoch pacemaker 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 the associated leader for 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 Replicas run 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 [37].

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 [8] for consensus instances (called Ladon-PBFT).

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

▷ pre-prepare phase (only for leader)
upon receive $2f + 1$ $rankmsg$ **do**
  $rankSet \leftarrow 2f + 1$ $rankmsg$
  $rank_m, QC \leftarrow$ getRank($rankSet$)
  $txs \leftarrow$ cutBatch($ins.bucketSet$)
  $d \leftarrow hash(txs)$
  $rank \leftarrow min\{rank_m + 1, maxRank(e)\}$
  $ppremsg \leftarrow \langle$pre-prepare$, v, n, d, i, rank\rangle_\sigma$
  multicast $\langle ppremsg, txs, QC, rankSet\rangle$
  if $rank = maxRank(e)$
    stop propose
  end if

▷ prepare phase
upon receive $ppremsg$ **do**
  if verify($ppremsg$)
    $premsg \leftarrow \langle$prepare$, v, n, d, i, rank\rangle_\sigma$
    multicast $premsg$
  end if

▷ commit phase
upon receive $2f + 1$ $premsg$ **do**
  if verify($premsg$)
    $commsg \leftarrow \langle$commit$, v, n, d, i, rank\rangle_\sigma$
    multicast $commsg$
    if $commsg.rank > curRank.rank$
      $curRank.rank \leftarrow commsg.rank$
      $curRank.QC \leftarrow$ agg($premsg$)
    end if
    $rankmsg \leftarrow \langle$rank$, v, n, \perp, i, curRank.rank\rangle_\sigma$
    send $\langle rankmsg, curRank.QC\rangle$ to leader
  end if

▷ Finally
upon receive $2f + 1$ $commsg$ **do**
  if verify($commsg$)
    $B \leftarrow \langle txs, i, n, rank\rangle$
    $\mathcal{G}_{in} \leftarrow \mathcal{G}_{in} \cup B$ //Commit B
  end if

upon receive $rankmsg$ **do**
  if verify($rankmsg$) $\wedge rankmsg.rank > curRank.rank$
    $curRank.rank \leftarrow rankmsg.rank$
    $curRank.QC \leftarrow rankmsg.QC$
  end if

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. In round $n \neq 1$, 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 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). Note that when $n = 1$, the leader doesn't need to wait for *rankmsg*, but let $rankSet[n] \leftarrow \langle \text{RANK}, v, n-1, \perp, i, curRank.rank \rangle_\sigma$.

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 in 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 PBFT view-change protocol to make progress [8]. 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, the new leader multicasts a new-view message to move the instance to view $v + 1$. Thereafter, the protocol proceeds as we described it.

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 [21]. 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 a real deployment.

Detailed protocol. We present an optimized version of normal-case operation for Ladon-PBFT in round $n \neq 1$. There are three modifications.

1) Pre-prepare. 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. 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. 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 in [32]. 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

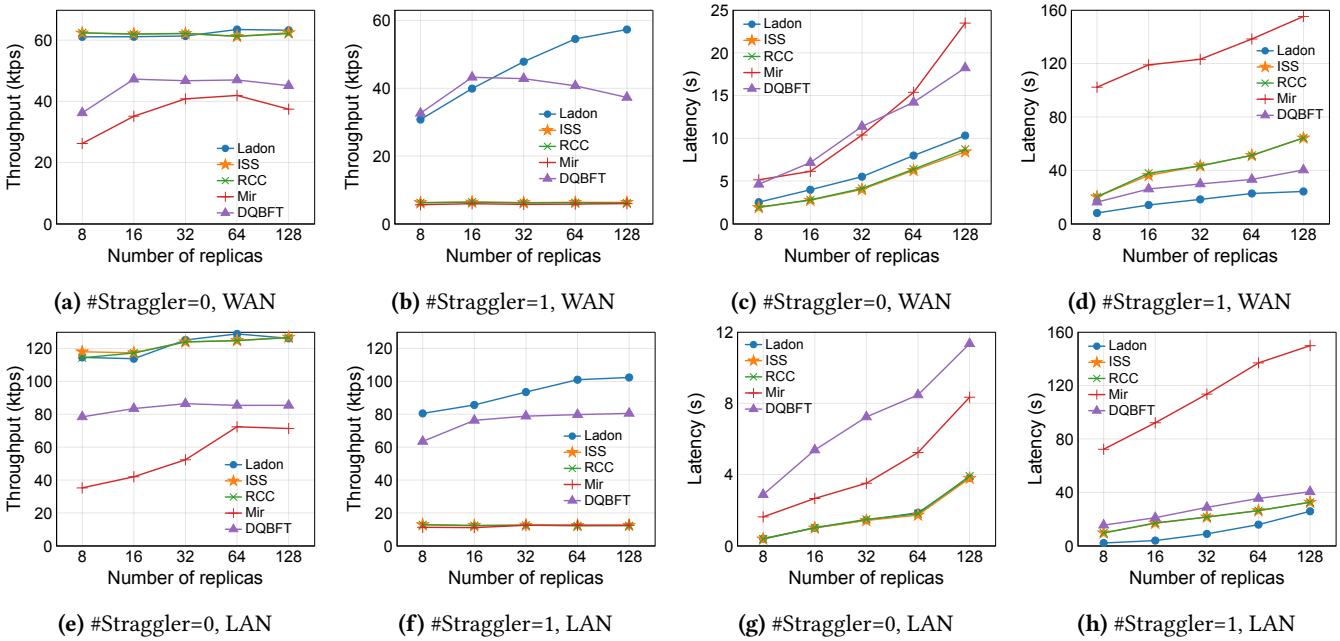
In this section, we evaluate the performance and causality of Ladon across different scenarios. We compare Ladon against four state-of-the-art Multi-BFT protocols: ISS [37], RCC [24], Mir [36], and DQBFT [1]. We implemented Ladon in Go[2] and used the Go BLS library for aggregate signatures. The evaluation 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 in [32]. 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 number of replicas in WAN and LAN environments 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 all systems 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 AWS data centers across France, America, Australia, and Tokyo. We distribute the replicas evenly 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 BFT consensus. We use NTP for clock synchronization across the servers. We conduct 5 experimental runs for each condition and plot the average value.

[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).

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 [33]. Each replica operates 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.

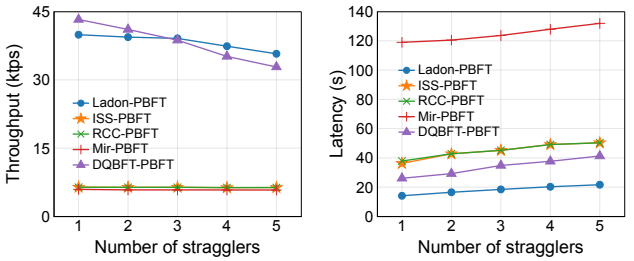
While we acknowledge that the above setting may not be exhaustive or optimal, conducting exhaustive experiments to identify the optimal configuration exceeds the scope of this work. However, our chosen parameters enable us to demonstrate that Ladon outperforms other protocols in the presence of stragglers 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 using 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. We also evaluate the performance of Ladon and its counterparts with a varying number of honest stragglers in WAN. In this 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×,



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

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

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.

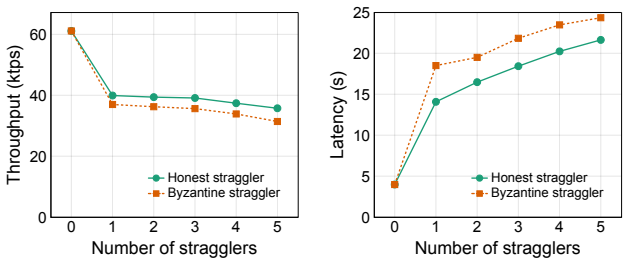
Comparing Fig. 5a and Fig. 5b, it is evident that the throughput of pre-determined global ordering protocols (*i.e.*, ISS, RCC, and Mir) significantly drops by 89.9%, 90.1%, and 84.1% on 128 replicas, respectively, in the presence of one straggler. In contrast, dynamic global ordering protocols (*i.e.*,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

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.

Fig. 6 evaluates the performance of each protocol with a varying number of stragglers in WAN. 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, ISS, RCC, Mir, DQBFT 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.

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

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



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

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

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 similar performance trends to those observed in the WAN environment (Fig. 5a–Fig. 5d), with higher throughput and reduced latency. Specifically, 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 possible 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.

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

Stragglers	# Stragglers					Proposal rate (blocks/s)				
settings	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

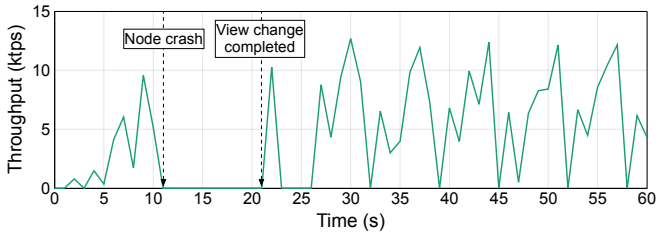


Figure 8. Ladon's throughput average (over 1s intervals) over time with one crash fault. The crash is at 11s, and the view change is completed at 21s.

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. 4.4.

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 average 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 $i < j$, 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 block as discussed in Sec. 4.3.

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

Table 2 shows that Ladon always exhibits strong inter-block 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. Traditional BFT protocols utilize predetermined ordering in which even 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. 4.3. 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 [29], PBFT [8], and HotStuff [39], 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 [24, 36, 37], which runs several consensus instances in parallel to handle transactions. Stathakopoulou *et al.* [36] 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 [37] 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 [24] 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, and the centralization of the ordering process makes it a prime target for attacks. 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 [11, 26, 35] 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 of this approach 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 [20]. 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 [31], OmniLedger [28] and RapidChain [40], 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 [6], in which a replica sends its messages to some randomly sampled replicas. Gossip has been used in Tendermint [5], Gosig [30], and Stratus [19], 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 [34], replicas disseminate and aggregate messages over trees, and in Hermes [25] 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.

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 distribution and collection 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.

Acknowledgement

We thank our paper's shepherd, Bernard Wong, and the anonymous reviewers for their feedback. We thank Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić for their work [37]. Their research has provided an essential foundation for the experiments and analyses presented in this paper. This work is supported in part by the NSFC under Grant 62302204; in part by the Shenzhen Science and Technology Program under Grants RCBS20221008093248075 and JSGG20220831095603007; and, in part by the NSERC Discovery Grants RGPIN-2014-04870 and RGPIN-2016-05310 as well as the CSCP Contract 4248090-1-NS-5001-22170.

References

[1] Balaji Arun and Binoy Ravindran. Scalable Byzantine Fault Tolerance via Partial Decentralization. In *PVLDB*, 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 *FC*, 2023.

[4] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *EUROCRYPT*, 2003.

[5] Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. In *M.Sc. Thesis, University of Guelph, Canada*, 2016.

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

[7] Miles Carlsten, Harry Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the Instability of Bitcoin Without the Block Reward. In *CCS*, 2016.

[8] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, 1999.

[9] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, Mirco Marchetti, et al. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, 2009.

[10] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, 2006.

[11] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. In *EuroSys*, 2022.

[12] Sisi Duan, Michael K Reiter, and Haibin Zhang. Secure Causal Atomic Broadcast, Revisited. In *DSN*, 2017.

[13] Sisi Duan, Michael K Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS*, 2018.

[14] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. In *JACM*, 1988.

[15] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. SoK: Transparent Dishonesty: Front-running Attacks on Blockchain. In *FC*, 2020.

[16] Ittay Eyal and Emin Gün Sirer. Majority is Not Enough: Bitcoin Mining is Vulnerable. In *Commun.*, 2018.

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

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

[19] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Scaling Blockchain Consensus via a Robust Shared Mempool. In *ICDE*, 2023.

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

[21] Abraham I Giridharan N, Howard H. No-Commit Proofs: Defeating Livelock in BFT. In *Cryptology ePrint Archive*, 2021.

[22] Tiantian Gong, Mohsen Minaei, Wenhai Sun, and Aniket Kate. Towards Overcoming the Undercutting Problem. In *FC*, 2022.

[23] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. In *EuroSys*, 2010.

[24] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *ICDE*, 2021.

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

[26] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All You Need is DAG. In *PODC*, 2021.

[27] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-Fairness for Byzantine Consensus. In *CRYPTO*, 2020.

[28] 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.

[29] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, 2007.

[30] 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.

[31] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A Secure Sharding Protocol For Open Blockchains. In *CCS*, 2016.

[32] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Chen Feng, Yinqian Zhang, and Ivan Beschastnikh. Ladon: High-Performance Multi-BFT Consensus via Dynamic Global Ordering (Extended Version). In *arXiv preprint arXiv:2409.10954*, 2024.

[33] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. In *Working Paper*, 2008.

[34] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *SOSP*, 2021.

[35] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. In *CCS*, 2022.

[36] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-BFT: Scalable and Robust BFT for Decentralized Networks. In *JSys*, 2022.

[37] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State Machine Replication Scalability Made Simple. In *EuroSys*, 2022.

[38] 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.

[39] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*, 2019.

[40] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling Blockchain via Full Sharding. In *CCS*, 2018.

A Artifact Appendix

This appendix provides a detailed guide on how to reproduce the results presented in our paper. Our experimental setup and architecture are largely based on the ISS framework, as detailed in [37]. Below, we describe how our system is structured and the steps necessary to replicate our experiments.

A.1 Abstract

Our implementation builds upon the ISS modular framework [37] to create a state-machine replication service. The primary function of this project is to maintain a totally ordered log of client requests. Similar to ISS, our system utilizes multiple instances of an ordering protocol, which are coordinated to produce a final, globally ordered log. The log is a sequence of entries. Each entry in the log is defined by a sequence number and contains a batch of client requests. All client requests are divided into subsets called buckets based on their hashes. There is a manager module that assigns

each bucket to a specific instance. Each ordering protocol instance then forms batches of requests using its assigned bucket. To prevent request duplication, the manager ensures that no two instances are assigned the same bucket.

A.2 Description & Requirements

A.2.1 How to access. The code used to produce the results of the experiments is publicly available in Github repository[4][5], which have 5 branches: `research-ladon`, `research-dqbft`, `research-iss`, `research-mirbft` and `research-rcc`. The code of protocol `x` can be found in the `research-x` branch.

A.2.2 Hardware dependencies. We performed our evaluation 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.

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. We use the public interface for client transactions and the private interface for server communications.

A.2.3 Software dependencies. Go 1.21+, Python 3.10

A.2.4 Benchmarks. None.

A.3 Set-up

Detailed setup instructions are available in the repository's README file. The deployment process is automated via scripts located in the deployment directory. These scripts enable you to deploy a network of nodes and clients, run experiments, analyze the results, and generate summary data.

For deployments on AWS Cloud, you need to set up an AWS account and register an SSH key with it first. The repository includes scripts to initialize the AWS CLI, streamlining the deployment process.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): Ladon significantly enhances the performance of ISS in the presence of stragglers, achieving a 8.1× increase in throughput and reducing latency by 62.3% in WAN with one honest straggler in 128 replicas. These improvements are demonstrated in the experiment (E1) described in Sec. 6.2.1, with results shown in Fig. 5b and Fig. 5d.
- (C2): Ladon achieves a 7.2× increase in throughput and reducing latency by 20.8% in LAN with one honest straggler in 128 replicas. These improvements are demonstrated in the experiment (E2), with results shown in Fig. 5f and Fig. 5h of Sec. 6.2.2.
- (C3): Ladon is robust under crash faults: the impact of the crash fault is limited since the throughput will recover after the view change. This is demonstrated in the experiment (E3) with results illustrated in Fig. 8 of Sec. 6.3.

A.4.2 Experiments. We first outline the general steps required to perform Ladon's experiments.
[Preparation] To set up for the experiments, follow these steps:

- **AWS Account and Configuration**: Ensure you have an active AWS account. Configure the *awscli* environment on the controller machine to interact with AWS services. This involves installing *awscli* and setting up your AWS credentials and default region.
- **Region Login and Template Creation**: Log in to each AWS region where you intend to conduct the experiments. Create an AWS EC2 launch template, which simplifies the process of launching new instances with pre-defined configurations.
- **Config Adjustment**: Modify the `deployment/scripts/cloud-deploy/deploy-cloud-WAN.sh` script to match your specific requirements and configurations. This script manages instance deployment and other operational tasks.

[Execution] With the preparation complete, navigate to the `Deployment` Directory, and configuration Adjustments. Before running each experiment, you should need to update the parameters in the `deployment/scripts/experiment-configuration/generate-config.sh` file. This file contains the configuration settings needed for the experiment. Then proceed with the following steps to execute the experiments:

- **Launch Instances**: Execute `bash scripts/cloud-deploy/deploy-cloud-WAN.sh -i -r -k`. This command initializes new instances based on the configuration specified in `generate-config.sh`, retrieves their basic information, and updates the necessary key settings.
- **Conduct Experiment**: Run `bash scripts/cloud-deploy/deploy-cloud-WAN.sh -d`. This command performs the experiment according to the configurations defined in `generate-config.sh`.
- **Shutdown Instances**: Use `bash scripts/cloud-deploy/deploy-cloud-WAN.sh -sd` to terminate all running instances in AWS EC2.

Alternatively, to perform all the above steps in one go, execute: `bash scripts/cloud-deploy/deploy-cloud-WAN.sh -i -r -k -d -sd`.

[4]https://github.com/eurosys2024ladon/ladon.git
[5]Persistent ID: 10.5281/zenodo.13714937

Table 3. Configuration files and corresponding figures

Branches	Configuration files	Figures
All branches	generate-config-8/16/32/64/128peer-1straggler-wan.sh	Fig. 5b, Fig. 5d
All branches	generate-config-8/16/32/64/128peer-0straggler-wan.sh	Fig. 5a, Fig. 5c,
All branches	generate-config-8/16/32/64/128peer-1straggler-lan.sh	Fig. 5f, Fig. 5h
All branches	generate-config-8/16/32/64/128peer-0straggler-lan.sh	Fig. 5e, Fig. 5g,
research-ladon	generate-config-different-straggler.sh	Fig. 7
research-ladon	generate-config-different-byzantinestraggler.sh	Fig. 7
research-ladon	generate-crash-faults.sh	Fig. 8

[Results] Upon completion of the experiments, the results will be available in the deployment-data directory. This directory contains a comprehensive set of detailed statistics, including throughput, latency, and other performance metrics. Review these files to analyze and interpret the outcomes of your experiments.

Then, we provide detailed configuration files and settings that can be used to reproduce the main claims of the paper. Automated scripts have been written to run the experiments presented in Figures 5, 6, and 7. You can execute them using the command: `bash ./scripts/cloud-deploy/deploy-multiple-figX-ladon.sh`. Additionally, the configuration files are listed in Table 3.

Experiment (E1): [Throughput and latency with one straggler in WAN] [1 human-hour + 3 compute-hour]: This experiment is designed to evaluate the system's peak performance in a WAN environment with one straggler, focusing on throughput and latency. This experiment involves running various scenarios with different configurations, including varying replica counts (8, 16, 32, 64, 128), and comparing the results with other protocols (Fig. 5b and Fig. 5d).

The corresponding configuration files are located in the `deployment/scripts/experiment-configuration` directory and follow the naming convention `generate-config-peernumber-strategy.sh`, for example, `generate-config-16peer-1straggler.sh`.

Experiment (E2): [Throughput and latency with one straggler in LAN] [1 human-hour + 3 compute-hour]: This experiment is similar to Experiment (E1), evaluating the system's peak performance in a LAN environment with one straggler (Fig. 5f and Fig. 5h). The configuration files are similarly named and located in the `deployment/scripts/experiment-configuration` directory.

Experiment (E3): [Throughput with crash faults in WAN] [1 human-hour + 1 compute-hour]: This experiment evaluates the system's robustness in the presence of crash faults (Fig. 8). The corresponding configuration file is located in the `deployment/scripts/experiment-configuration` directory and is named `generate-crash-faults.sh`. Two fault strategies are tested: one where faulty processes crash immediately after the epoch begins, and another where they crash at the end of the epoch, after all blocks have been proposed.