Cerberus: High-Performance and Secure Multi-BFT Consensus via Dynamic Global Ordering

Hanzheng Lyu[1,2], Shaokang Xie[2], Jianyu Niu[2], Ivan Beschastnikh[1], Yinqian Zhang[2], Chen Feng[1]
[1]University of British Columbia
[2]Southern University of Science and Technology
[1]{hzlyu@student.ubc.ca, chen.feng@ubc.ca, bestchai@cs.ubc.ca}
[2]{xiesk2020@mail.sustech.edu.cn, niujy@sustech.edu.cn, yinqianz@acm.org}

ABSTRACT

In Multi-BFT consensus, multiple leader-based consensus instances, *e.g.*, PBFT or HotStuff, run in parallel, circumventing the leader bottleneck of a single instance. Despite the improved performance, the Multi-BFT consensus has 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 different instances, and simultaneously, it must respect inter-block causality constraints for security. In prior Multi-BFT designs, each block is assigned a global index before creation, leading to poor performance and causality violations.

We propose Cerberus, a high-performance and secure Multi-BFT protocol that considers varying block rates of instances while ensuring block causality. Our key idea is to dynamically order consensus results across instances. First, the dynamic ordering decouples the dependencies between the replicas' partial logs to ensure fast generation of the global log. This eliminates blocking on slow instances. Second, blocks are ordered by their generation sequence, which respects inter-block causality. To this end, we assign *monotonic ranks* to the output blocks of instances for global ordering. We further pipeline these monotonic ranks with the consensus process and aggregate signature to reduce protocol overhead. We implemented and evaluated Cerberus by initializing consensus instances with PBFT. Our evaluation shows that Cerberus improves the peak throughput of ISS, a state-of-the-art Multi-BFT design, by 6x and reduces transaction latency of ISS by 94%, when deployed with one straggling replica out of 32 replicas in WAN.

PVLDB Reference Format:
Hanzheng Lyu[1,2], Shaokang Xie[2], Jianyu Niu[2], Ivan Beschastnikh[1], Yinqian Zhang[2], Chen Feng[1]. Cerberus: High-Performance and Secure Multi-BFT Consensus
via Dynamic Global Ordering. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and / or other artifacts have been made available at https://github.com/JeffXiesk/cerberus.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Byzantine Fault Tolerant (BFT) consensus is crucial in establishing a strong decentralized trust foundation for modern decentralized applications. BFT consensus enables a group of replicas to agree on the same sequence of transactions in the presence of Byzantine faults (*i.e.*, arbitrary replica behavior). Most existing BFT consensus protocols adopt a leader-based scheme [10, 12, 23]. In such a scheme, the protocol run in views and each view has a delegated replica, known as the leader. The leader is responsible for broadcasting proposals (a batch of client transactions) and coordinating with replicas to reach consensus on the proposals and their ordering. 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, 18, 24, 40, 42], making the coordinating leader the dominant factor in the system's throughput and latency.

To address the leader bottleneck, Multi-BFT systems [2, 24, 40–42] have emerged as a promising alternative. In a Multi-BFT setting, multiple leader-based BFT instances, such as off-the-shelf BFT consensus protocols like PBFT [8] or HotStuff [45], run in *parallel*, as shown in Figure 1. A replica may act as both the leader for a BFT instance and a backup for other instances. Like a single BFT system, each BFT instance in Multi-BFT outputs a sequence of confirmed blocks, called the partially confirmed blocks. To behave as a single instance system, these partially confirmed blocks are globally ordered and then confirmed. Intuitively, such a scheme can balance the workloads of replicas, 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, 40–42] 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 Figure 1 with three instances each outputting (*i.e.*, partially confirm) three blocks and one instance outputting one block. The three blocks confirmed by Instance 0 receive global indices of 0, 3, and 6. Replicas execute partially confirmed blocks with increasing global index one by one until they see a missing block. The pre-determined global indices enable blocks across instances to be arranged into a consistent sequence of blocks because a partially confirmed block will never be reverted once produced.

This simple global ordering method has performance and security issues. *First*, a slow instance, often called a straggler, can slow down the entire system. For example, the straggler in Figure 1 only outputs one block (with a global index of 1). This causes two "holes" in the global log (at positions 4 and 7), and prevents three blocks

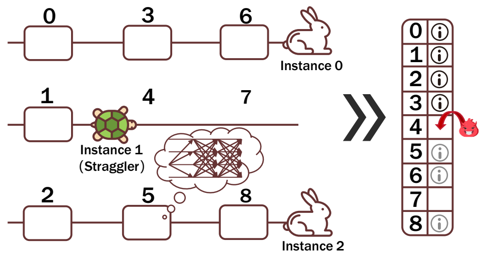


Figure 1: An overview of Multi-BFT paradigm. There are three BFT instances running in parallel, and blocks output from each instance (*i.e.*, partially confirmed) have to be globally ordered and become globally confirmed. The black icons (resp., gray icons) refer to the globally confirmed (resp., partially confirmed) blocks.

(5, 6, and 8) from being globally confirmed. This reduces the system's throughput and increases latency. This has been confirmed by experimental results from prior work [42]: even one straggler among 32 replicas can reduce a system's maximum throughput by 85% and increase latency by 14x! (We provide theoretical analysis and detailed experimental results in Section 2.1.)

Second, the pre-determined global ordering violates the causality of blocks across instances. For example, in Figure 1, block 4 is generated after blocks 5 and 6, but it will be globally confirmed and executed before them. This causality violation can lead to attacks, such as *front-running* [3, 16, 37] and *incentive* attacks [17, 34]. These attacks can have severe consequences in decentralized applications like DeFi [3], online voting system [39], and decentralized exchanges [1]. For example, Torres *et al.* [43] found that front-running attacks have caused a total loss of over 18.41M USD on the Ethereum blockchain, and reports [11] show that replicas can obtain 686M USD by changing the sequence of transactions (known as the Miner Extractable Value, MEV).

In this paper, we propose Cerberus, a high-performance and secure Multi-BFT consensus that considers varying block rates of instances while ensuring block causality. The key insight in our approach is to *dynamically* order partially confirmed blocks from different instances. To this end, each block is assigned a *monotonic rank*, which satisfies two properties: 1) *agreement*: all honest replicas have the same *rank* for a partially confirmed block; and, 2) *monotonicity*: the ranks of later proposed blocks are always larger than that of a partially confirmed block. Furthermore, partially confirmed blocks can be globally ordered by monotonic ranks together with a tie-breaking rule. This 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 inter-block causality.

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 first pipeline the rank information collection with the last round of consensus. Second, we use aggregate signatures creatively to reduce the message size. Specifically, each replica is assigned a set of private keys, where each key denotes the difference between the highest rank that the replica knows and the rank used in the proposed blocks. With these optimizations, the overhead of using monotonic ranks is only 10.3% compared to pre-determined global ordering.

In summary, we make the following contributions:

- We identify the performance degradation caused by stragglers and causality violations in existing Multi-BFT protocols using pre-determined global ordering.
- We propose Cerberus, a Multi-BFT system that dynamically orders blocks to solve the issues identified above. Cerberus builds on dynamically monotonic ranks and pipelines their formulation with the consensus process. Furthermore, Cerberus adopts the aggregate signature to reduce the size of the rank information.
- We build an end-to-end prototype of Cerberus and conduct extensive experiments on Amazon AWS to evaluate its performance. We run experiments over LAN and WAN with $8-64$ replicas, distributed across 4 regions. With one straggler in the LAN setting, Cerberus achieves 6x higher throughput and 92% lower latency than ISS [42] with one straggler in 32 replicas. In the WAN setting, Cerberus achieves 5x higher throughput and 97% lower latency with one straggler in 16 replicas. We also show that Cerberus is more secure to front-running attacks than ISS.

Roadmap. Section 2 presents the susceptibilities of existing Multi-BFT consensus protocols, and Section 3 provides the system model, preliminaries, and problem statement. A high-level description of the key components are provided in Section 4, and a detailed design of Cerberus is given in Section 5. Evaluation is provided in Sec. 6, and the related work is discussed in Section 7. Finally, Section 8 concludes the paper.

2 EXISTING MULTI-BFT SUSCEPTIBILITY

In this section, we first introduce the performance limitations and causality violations of existing Multi-BFT systems, and we then revisit prior solutions to address these issues.

2.1 Performance Degradation

The pre-determined global ordering of existing Multi-BFT protocols [2, 24, 40–42] performs well when all instances have the same block production rate. In an ideal scenario where instances simultaneously output partially confirmed blocks with the same sequence number, these blocks become globally confirmed immediately. Therefore, compared with a single instance system, Multi-BFT protocols with m parallel instances can achieve m times larger throughput with almost the same latency. By adjusting m, the system can maximize the capacity of each replica, allowing the throughput to approach the physical limit of the underlying network.

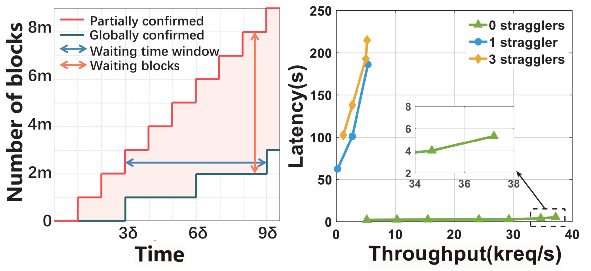
However, the performance will significantly drop when there are stragglers or slow instances. Stragglers in real-world deployments can be caused by faulty or capacity-limited leaders, an unstable network, or malicious behavior. Consider a simple case where a slow instance produces blocks every k rounds while the remaining $m-1$ normal instances produce blocks every round. (That is, the speed of a normal instance is k times that of the straggler.) Let R (resp., R') denote the number of globally confirmed (resp., partially confirmed) blocks per round. As shown in Figure 1, the straggler's block creates a "hole' in the global index, which prevents blocks with the same sequence number from other instances to be globally confirmed. In other words, once the "hole" is filled, m blocks will be globally confirmed. Therefore, we have $R = m/k$, which implies that the system throughput is only m times larger than that of the straggler instance (*i.e.*, $1/k$). As well, we compute $R' = m/k+(m-1)(k-1)/k$, where $(m-1)(k-1)/k$ is the number of newly added partially confirmed blocks, waiting to be globally confirmed, per round. Now, over time, more and more partially confirmed blocks from normal instances will wait to be globally confirmed, which will lead to continuous growth in delay.

Figure 2a shows the analytical results of the simple case above, in which we have $m = 4$ and $k = 3$. First, we observe that the number of partially confirmed blocks that wait for being globally confirmed grows larger and larger as time increases. Meanwhile, the delay for partially confirmed blocks to become globally confirmed also grows longer and longer as time increases. Note that, in this case, we only consider one straggler, and extending to more stragglers will only make it slightly worse. This is because the effect of stragglers on system performance is a kind of bucket effect: the system performance is mainly limited by the slowest instance.

Figure 2b presents experimental results of throughput and latency to show the practical impact of the straggler. We run the Multi-BFT protocol, ISS [42], in which consensus instances are instantiated with PBFT [10]. We set $m = 16$, and the number of stragglers is chosen from $\{0, 1, 3\}$. The results show that the maximum throughput with one straggler and 3 stragglers reduces to 85.6% and 83.8% of that without stragglers, respectively. The average end-to-end transaction latency with 1 straggler and 3 stragglers increases up to 34x and 40x of that without stragglers.

2.2 Causality Violation

Existing Multi-BFT protocols have severe security issues when deployed in decentralized applications. This is because their predetermined global ordering does not respect blocks' causality, by which an adversary can manipulate the ordering of transactions. The causality of events in distributed systems has been formally defined [36]. It captures the cause-and-effect relationships between events occurring in different replicas, assuming that any action a replica takes may be affected by any message it has previously received. Similarly, in a Multi-BFT system, causality can be informally understood as the binary relation between blocks, assuming that any block a replica proposes may be affected by any block it has previously committed. (See formal definition in Section 3.3.) To understand the causality violations in existing protocols, see Figure 1, in which block 4 is proposed after blocks 5 and 6, but it will be globally confirmed and executed before blocks 5 and 6.



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

Figure 2: The analytical and experimental performance results of Multi-BFT consensus with/without straggler. The vertical line in the left figure represents the queued blocks waiting for being globally confirmed, while the horizontal line represents the delay of the transition.

This violation may lead to several ordering manipulation attacks, *e.g.*, front-running attack [3, 35] and undercutting attack [22]). For example, in a front-running attack [3, 35], an attacker attempts to place its own transaction ahead of the other unexecuted transaction to obtain profits from applications. For example, consider a cryptocurrency exchange case, shown in Figure 1, in which an attacker sees a large buy order tx_v in block 5. Then, the attacker creates a similar buy order tx_m in block 4, and consequently, 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 for tx_v, obtaining profits at the expense of the original buyer.

The order manipulation also incentivizes an adversary to gain more profits if the Multi-BFT consensus is adopted in real applications with reward mechanisms [17, 34]. In particular, in an incentive-based attack [7], an attacker wins more transaction fees by overriding transactions included in previous blocks. For example, if there is a transaction tx with a high transaction fee, and tx has been placed in block 5 in Figure 1, the leader of the instance 1 may still put tx in block 4, which would be executed before the block 5. Thus, leader 1 rather than leader 2 will win the associated transaction fee, which incentivizes selfish replicas to launch this attack in some real applications.

2.3 Revisiting Straggler Mitigation

We revisit existing methods to reduce the impact of stragglers on system performance. In general, these methods focus on detecting straggling leaders and then replacing them with normal ones. For example, Stathakopoulou *et al.* [41] proposes to use the timeout mechanism, in which replicas set a timeout for the block with global index n as soon as the block with index $n-1$ is partially committed. If the timeout of enough replicas is triggered, the associated leader will be regarded as a malicious straggler and then be replaced. Similarly, in RCC [24], a straggling leader will be removed once its instance lags behind other instances by a certain number of blocks.

The straggling leader detection and replacement falls short in three aspects. First, if there are multiple colluding stragglers in the system, it is difficult to detect them using the above detection mechanism. 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 factor of straggler instances, and network turbulence, and replicas' varying runtime capacities could also make an instance produce blocks slowly in a period.

Summary. Existing Multi-BFT protocols suffer from performance issues and casualty violations. The straggling leader detection and replacement cannot fix these issues. This motivated us to design a *high-performance* and *secure* multi-BFT system, which algorithmically mitigates the impact of stragglers.

3 MODELS AND PROBLEM STATEMENT

3.1 System Model

We consider a system with $n = 3f + 1$ replicas, denoted by the set $\mathcal{N}$. Replicas process transactions from a set of clients. 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. We assume all the Byzantine replicas are assumed to be controlled by a single adversary, which is computationally bounded. In other words, the adversary cannot break the cryptographic primitives to forge honest replicas' messages (except with negligible probability). There is a public-key infrastructure (PKI) among replicas: 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.* [15], which is widely used in BFT consensus [10, 45]. 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.

3.2 Preliminaries

We introduce some used building blocks in this work.

Byzantine atomic broadcast (BAB). A Byzantine atomic broadcast is a communication primitive that ensures the total ordering of messages in distributed systems and maintains the consistency of replicated information despite Byzantine failures [14]. The interface of BAB is defined by two primitives: *propose* and *confirm*. The following properties are usually required from a BAB protocol:

- **BAB-Validity**: If an honest replica proposes a transaction tx, then it eventually confirms tx.
- **BAB-Agreement**: If an honest replica partially confirms a transaction tx, then all honest replicas eventually confirms tx.
- **BAB-Integrity**: For any transaction tx, every honest replica delivers tx at most once, and only if it was previously proposed by an honest replica.
- **BAB-Total order**: If an honest replica r confirms tx before tx', then every other honest replica must confirm tx before tx'.

Aggregate Signature Schemes. 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: $\textsc{agg}(\{\sigma_r\}_{r\in R}) \rightarrow \sigma$. Given the aggregation signature, the identity of the aggregation signer r and the original message m_r of the signature can be generated, and the verifier can be sure that it is the user r signed on the message m_r by the verify function: $\textsc{verifyAgg}((pk_r, m_r)_{r\in R}, \sigma) \rightarrow 0/1$.

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 that produce the block, $round$ denotes the index of 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. In this work, we focus on the block level, and therefore we ignore the internal details of application-specific transactions within a batch.

3.3 Problem Formulation

We consider a Multi-BFT system consisting of m BFT instances. Specifically, a Multi-BFT 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 [41]), which assigns clients' transactions to different instances to avoid transaction redundancy. Each instance runs multiple rounds of BAB protocols, and in each round, a leader packs clients' transactions in blocks, proposes the blocks, and coordinates all replicas to continuously agree on the same blocks. The block produced at instance i in round j can be denoted as B^i_j. The output of the partial ordering layer is the sequenced *partially confirmed* blocks produced by all the instances $\mathcal{P}_{out} = \langle B^i_1, B^i_2, ..., B^i_k \rangle^{m-1}_{i=0}$.

Global ordering layer. A Multi-BFT system should perform as a single BFT system, and so blocks output by partial ordering layer across m instances should be ordered in a global sequence. Thus, the input of the global ordering layer is the output of the partial ordering layer, *i.e.*, $\mathcal{G}_{in} = \mathcal{P}_{out} = \langle B^i_1, B^i_2, ..., B^i_k \rangle^{m-1}_{i=0}$. By following certain determined rules, these input blocks are ordered and outputted in a sequence, and the associated index is denoted as sn. These output blocks are *globally confirmed*, denoted by the set $\mathcal{G}_{out}$, which further 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 B, then all honest replicas eventually globally confirm B.
- **$\mathcal{G}$-Liveness**: If a correct client broadcasts a transaction tx, honest replica eventually globally confirms a block B that includes it.

Except for satisfying the above properties as existing work [42], a high-performance and secure Multi-BFT system also has to guarantee causality of blocks and robustness, which are defined as:

- **$\mathcal{G}$-Causality:** If a block B is partially confirmed by any honest replicas before the creation of block B', then block B is globally confirmed before block B' (*i.e.*, $B.sn < B'.sn$).

  $\mathcal{G}$-Robustness: The throughput and end-to-end transaction latency of a Multi-BFT system with stragglers approach to these in a system without stragglers.

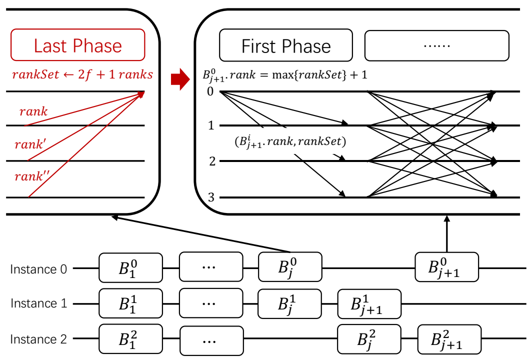


Figure 3: The realization of monotonic rank. The monotonic rank collection process is integrated with the last phase of the previous consensus round. Then, the leader includes the latest ranks in its blocks.

We note that the inter-block causality at the same instances can be ensured by BAB because of their strictly increasing round number.

4 DYNAMIC GLOBAL ORDERING OVERVIEW

We provide a high-level description of monotonic rank and its usage in realizing dynamic global ordering.

4.1 Monotonic Rank

We first abstract the required properties of monotonic ranks for dynamic global ordering and then introduce our realization.

Properties of monotonic ranks. The key component of the global ordering layer is to assign partially confirmed blocks with monotonic ranks (short for rank in the following discussion). These 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 confirmed block.
- **MR-Monotonicity:** The ranks of later generated blocks are larger than those of a partially confirmed block.

The MR-Agreement property guarantees that given a set of partially confirmed blocks with ranks, honest replicas can run certain determined ordering algorithms to output the same sequence of globally confirmed blocks. The MR-Monotonicity property guarantees inter-block causality.

The realization. A block can be assigned a rank either when it is proposed or when it is output by a consensus instance (*i.e.*, partially confirmed). We observe that for the former, no additional procedure is required to achieve the agreement property since the rank is piggybacked with the block that has to go through the consensus process. By contrast, the latter requires some complicated consensus process for agreeing on the same rank. Therefore, we chose the former approach.

When including a rank for a block, a malicious leader may either exhaust the range of ranks to make the algorithm fail or lies to use stale ranks to violate the dynamic monotonicity and drag down the whole system performance. To prohibit these misbehaviors, each leader has to prove his best try to use the highest rank for the block. Specifically, before proposing a block, a leader first collects more than $2f+1$ replicas' highest ranks, and then increases the highest rank by one as the rank for its block. The collected ranks are also included in blocks for validity check.

The above rank collection and inclusion in the block introduce high protocol overhead. To reduce the overhead, we first observe that the consensus processes of blocks are repeated, and so rank collection can be integrated with the phases of the previous block. This cuts off the additional rank collection latency. Second, we find that using aggregate signatures can reduce the size of ranks included in blocks.

Figure 3 depicts the high-level realization of ranks. Generally, a BFT protocol contains multiple phases of message exchanges. At the last phase of block B^0_j's consensus process, each replica sends its highest ranks to the leader. When proposing the block B^0_{j+1}, the leader includes a set of $2f+1$ ranks in the block, and meanwhile, choose the highest rank plus one as B^0_{j+1}'s rank.

When a block B is partially confirmed, this block's rank will be known by at least $f+1$ honest replicas. Thus, when a new block B' is created, at least one honest replica among the above $f+1$ replicas will report its highest rank that is no less than B's rank to the proposer of B'. Therefore, B''s rank is larger than B's rank, which satisfies the second property.

4.2 Global Ordering Algorithm

The input of the global ordering algorithm running at a replica is the set of partially confirmed blocks from the partial ordering layer, *i.e.*, $\mathcal{G}_{in} = \langle B^i_1, B^i_2, ..., B^i_k \rangle^{m-1}_{i=0}$ (Section 3.3). In particular, these blocks are ordered by increasing the value of their ranks and a tie-breaking to favor consensus instances with smaller indexes. For example, given two blocks B and B', block B will be globally ordered after 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 their relationship.

The key step of global ordering is to decide when partially confirmed blocks become globally confirmed. In particular, we focus on blocks in the set $\mathcal{S} = \mathcal{G}_{in} \setminus \mathcal{G}_{out}$ we first find the *candidate block* $B'' \in \mathcal{S}$, that satisfies $\forall B' \in \mathcal{S} \wedge B' \neq B'', B'' \prec B'$. Next, we decide whether block B can be outputted as a globally confirmed block. We fetch the last partially conformed block from each instance and use $\mathcal{S}'$ to denote the set. We find the *bar block* $B^* \in \mathcal{S}'$, that satisfies $\forall B' \in \mathcal{S} \wedge B^* \neq B', B^* \prec B'$. Block B'' is globally confirmed if it satisfies one of the following conditions: 1) $B'' = B^*$, 2) $B''.rank < (B^*.rank + 1)$, and 3) $B''.rank = (B^*.rank + 1) \wedge B''.index < B^*.index$.

Figure 4 provides a concrete example of the global ordering process. Suppose a replica has $\mathcal{G}_{out} = \emptyset$ and $\mathcal{G}_{in} = \langle B^0_1, B^0_2, B^0_3, B^1_1, B^1_2, B^1_3, B^2_1 \rangle$ at the time t_1. According to the above algorithm, the bar block is B^2_1, and the first candidate block is B^0_1. Since B^0_1 satisfies the second condition, it can be globally confirmed and added to $\mathcal{G}_{out}$. Repeating the above process, we can subsequently appended blocks B^1_1, B^2_1, B^0_2, and B^1_2 to $\mathcal{G}_{out}$. At the time t_2, a new partially confirmed block B^2_2 is added to instance 2. Besides, since more than $f+1$ honest replicas have seen block B^0_3 with $rank = 3$, thus B^2_2 will have

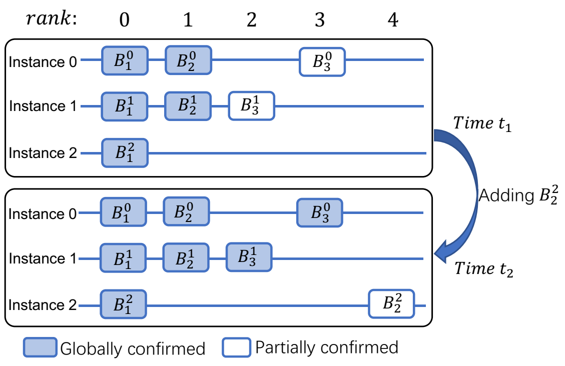


Figure 4: A simple illustration of the dynamic global ordering process.

$B_2^2.rank = 4$. Then, $\mathcal{S}$ is updated to $\{B_3^0, B_3^1, B_2^2\}$., and the bar block is updated to B_3^1. It is easy to see that the candidate block B_3^1 satifies the first condition and can be added to $\mathcal{G}_{out}$. Then B_3^0 becomes the candidate block and is added to $\mathcal{G}_{out}$.

5 CERBERUS

We first provide an overview of CERBERUS (Section 5.1) and then give the detailed system description. Specifically, we use PBFT to initialize the consensus instances in CERBERUS. We provide a basic version, called CERBERUS-BA in Section 5.2, and an alternative with optimized message size, called CERBERUS-OPT in Section 5.3. We present a correctness analysis of CERBERUS in Section 5.4.

5.1 Overview

Figure 5 provides an overview of CERBERUS, which has three components: rotating buckets, epoch pacemaker, and instance consensus.

CERBERUS adopts the rotating buckets from ISS [42]. Rotating buckets are used to prevent multiple leaders from simultaneously including the same transaction in a block. Clients' transactions are divided into disjoint buckets, and these are rotationally assigned to consensus instances when an epoch changes. Here, bucket rotation can mitigate censoring attacks, in which a malicious leader refuses to include transactions from certain clients.

The epoch pacemaker ensures CERBERUS proceeds in epochs. At the beginning of an epoch, CERBERUS has to configure the number of consensus instances, and the associated leader for each instance, create checkpoints, and initialize systems parameters. Epoch pacemaker is detailed in Section 5.2.1.

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

5.2 The CERBERUS-BA Algorithm

5.2.1 Epoch Pacemaker. CERBERUS proceeds in epochs with the main algorithm shown in Algorithm 1. We start with epoch number 0 (line 2), an empty undelivered block set (line 3), and no delivered blocks (lines 4). The parameter *curRank* is a tuple of a *rank* value

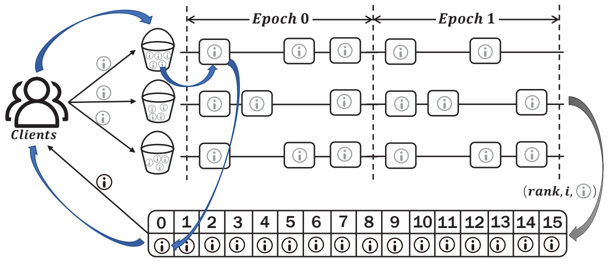


Figure 5: An overview of CERBERUS that proceeds in epochs to process clients' transactions.

and a *QC* for it (lines 5 and 6). All buckets are initially empty (line 8). Then, we initialize CERBERUS from epoch(0) (line 10).

Algorithm 1 CERBERUS Algorithm for replica r

```
▷ CERBERUS initialization
1:  upon $init()$ do
2:      $currentEpoch \leftarrow 0$
3:      $undeliverdSet \leftarrow \emptyset$
4:      $sn \leftarrow 0$
5:      $curRank.rank \leftarrow -1$
6:      $curRank.QC \leftarrow \perp$
7:      for $bucket \in bucketSet$ do
8:          $bucket \leftarrow \emptyset$
9:      end for
10:     INITEPOCH(0)
11:
▷ Epoch initialization
12: function INITEPOCH($e$)
13:     currentEpoch $\leftarrow e$
14:     $(minRank, maxRank) \leftarrow$ RANKRANGE($e$)
15:     $(minIndex, maxIndex) \leftarrow$ INSINDEXES($e$)
16:     $leaderSet \leftarrow$ leader selection policy($e$)
17:     for $l \in leaderSet$ do
18:         $ins \leftarrow$ new instance
19:         $ins.leader \leftarrow l$
20:         $ins.bucketSet \leftarrow$bucket assignment policy($e, ins$)
21:         $ins.index \leftarrow$ index assignment policy($e$)
22:     end for
23:     RUNINSTANCE $ins$
24:
▷ Epoch advancement
25: upon $B.maxRank$ is partially confirmed in each instance do
26:     CHECKPOINT($currentEpoch$)
27:     INITEPOCH($currentEpoch + 1$)
```

Epoch initialization. At the start of each epoch e, CERBERUS will do the following: (1) calculating the range of *ranks* and instance index (line 14 and 15), (2) calculating the set $Leaders(e)$ of replicas that will act as leaders in epoch e based on the leader selection policy (line 19), (3) for each replica l in $Leaders(e)$, creating a new instance with leader l (lines 18 and 19), (4) assigning buckets and indexes to the created instances (lines 20 and 21), (5) running instance *ins* (line 23, see details in Section 5.2.2).

Here, we omit the details of the leader selection policy, bucket assignment policy, and index assignment policy, because they are essentially the same as the policies in ISS [42]. We refer our readers to [42] for a detailed description of these policies.

The function RANKRANGE is used to calculate the range of *rank*s for an epoch. For epoch($e = 0$), $minRank(e) = 0$; for epoch($e \neq 0$), $minRank(e) = maxRank(e-1)+1$. Assume that the length of epoch$e$ is $l(e)$. Then, we have $maxRank(e) = minRank(e) + l(e) - 1$.

Epoch advancement. CERBERUS advances from epoch(e) to epoch($e+1$) when all the instances reach the *maxRank*, *i.e.*, the blocks with *maxRank* in all instances have been partially confirmed (line 25). Only then does the replica start processing messages related to epoch($e + 1$). To prevent transaction duplication across epochs, CERBERUS requires a replica to have output all batches in epoch(e) before proposing batches for epoch($e + 1$). So it will execute a CHECKPOINT protocol to broadcast a checkpoint message on the current epoch (line 26) before moving to the next epoch (lines 27). Upon receiving a quorum of $2f + 1$ valid checkpoint messages, a replica creates a stable checkpoint on 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 proves the integrity of the data.

5.2.2 *Consensus Instance.*

This section gives the protocol description of instances in CERBERUS instantiated with the PBFT [8] protocol.

Data structure. The general messages have a format of the tuple $\langle type, v, n, d, i, rank\rangle_\sigma$, 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 message, 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. As mentioned above, an aggregation of $2f+1$ signatures of a message *msg* is called a *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 CERBERUS protocol in the normal case without faults. Every replica runs RUNINSTANCE (Section 5.2) to start an instance. 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 (line 1) and round 1 (line 2) and a unique instance index (line 3). 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 CHECKMSG 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.

Algorithm 2 RUNINSTANCE Instantiated with PBFT

```
1:  $v \leftarrow 0$ //$v$ is the view number
2:  $n \leftarrow 1$ //$n$ is the round number
3:  $i \leftarrow ins.index$
    ▷ PRE-PREPARE phase (only for leader)
4:  upon receive $2f + 1$ $rankmsg$ do
5:      $rankSet \leftarrow rankSet[n]$
6:      $rank_m, QC \leftarrow$GETRANK$(rankSet)$
7:      $txs \leftarrow$ CUTBATCH$(ins.bucketSet)$
8:      $d \leftarrow hash(txs)$
9:      $rank \leftarrow min\{rank_m + 1, maxRank\}$
10:     $ppremsg \leftarrow \langle$PRE-PREPARE$, v, n, d, i, rank\rangle_\sigma$
11:     multicast $\langle ppremsg, txs, QC, rankSet\rangle$
12:     if $rank = maxRank$
13:         return //stop propose
14:     end if
15:
    ▷ PREPARE phase
16: upon receive $ppremsg$ do
17:     if CHECKMSG$(ppremsg)$
18:         $premsg \leftarrow \langle$PREPARE$, v, n, d, i, rank\rangle_\sigma$
19:         multicast $premsg$
20:     end if
21:
    ▷ COMMIT phase
22: upon receive $2f + 1$ $premsg$ do
23:     if CHECKMSG$(premsg)$
24:     $commsg \leftarrow \langle$COMMIT $, v, n, d, i, rank\rangle_\sigma$
25:     multicast $commsg$
26:     if $commsg.rank > curRank.rank$
27:         $curRank.rank \leftarrow commsg.rank$
28:         $curRank.QC \leftarrow$ AGG$(premsg)$
29:     end if
30:     $rankmsg \leftarrow \langle$RANK$, v, n, \perp, i, curRank.rank\rangle_\sigma$
31:     send $\langle rankmsg, curRank.QC\rangle$ to leader
32:
    ▷ Finally
33: upon receive $2f + 1$ $commsg$ do
34:     if CHECKMSG$(commsg)$
35:         $B \leftarrow \langle txs, i, n, rank\rangle$
36:         commit $B$
37:     end if
38:
39: upon receive $rankmsg$ do
40:     if CHECKMSG$(rankmsg)$
41:         $n \leftarrow rankmsg.n + 1$
42:         $rankSet[n] \leftarrow append(rankSet[n], rankmsg)$
43:         if $rankmsg.rank > curRank$
44:             $curRank.rank \leftarrow rankmsg.rank$
45:             $curRank.QC \leftarrow rankmsg.QC$
46:         end if
47:     end if
```

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 0, which only contains a *rankmsg* of the leader. Note that when $n = 1$, the leader doesn't need to wait for *rankmsg*, but let $rankSet[n] \leftarrow \langle \text{RANK}, v, n-1, \perp, i, curRank.rank\rangle_\sigma$. 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 5), and picks the maximum rank value with its QC from *rankSet*, denoted as $rank_m$ and QC (line 6). Then the leader cuts a batch *txs* of transactions (line 7), and calculates the digest of *txs* (line 8). A rank number $rank = rank_m + 1$ is assigned to *txs*, which should not exceed the *maxRank* of the current epoch (line 9). The leader multicasts a pre-prepare message (line 11) 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*, the leader stops proposing (lines 12 and 13).

2) PREPARE. A backup accepts the pre-prepare message provided:

- 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 $rank_m + 1 \leq maxRank$, $rank = rank_m + 1$; if $rank_m + 1 > maxRank$, $rank = maxRank$.

It then enters the prepare phase by multicasting a $\langle \text{PREPARE}, v, n, d, i, rank\rangle_\sigma$ message to all other replicas (lines 18 and 19). Otherwise, it does nothing.

3) COMMIT. Upon receiving $2f+1$ valid prepare messages from different replicas (lines 22 and 23), a replica multicasts a commit message to other replicas (line 25). If the *rank* carried in the commit message is greater than the current highest rank the replica knows *curRank.rank*, the replica updates its *curRank* by setting *curRank.rank* to *commsg.rank* (line 27), and generates a *QC* for it by aggregating the $2f+1$ *premsg* (line 28). Then, a backup sends a *rankmsg* together with the QC for *rank* to the leader to report its *curRank* (lines 30 and 31).

Finally, upon receiving $2f+1$ valid commit messages, a replica commits a block B with its corresponding parameters (lines 33 to 36). All the committed blocks will be globally confirmed and delivered to clients.

4) Respond to clients. Every time a replica commits a block, it checks whether the undelivered blocks satisfy the conditions to be globally confirmed introduced in Section 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. 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 [8].

5.3 The CERBERUS-OPT Algorithm

CERBERUS-BA requires the leader to broadcast at least $2f+1$ rank messages for the purpose of allowing backups to authenticate the accuracy of the leader's *rank* calculation. This measure is implemented to prevent 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 the PBFT protocol). Recall that the overall communication complexity of the PBFT protocol is $O(n^2)$. So, our protocol doesn't increase the overall complexity compared with PBFT. Interestingly, we can use the following optimization to reduce the communication complexity of our pre-prepare phase from $O(n^2)$ to $O(n)$.

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. Therefore, we need to define rank messages in an innovative way: rather than encoding the *rank* value information in the message directly, we encode it in the private keys.

Specifically, we modify the rules for generating rank messages. For each replica, we generate several private keys. In round n, when replica r creates a rank message, it computes the difference between the highest rank that it has known (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.*, $rankmsg \leftarrow \langle \text{RANK}\ v, n, \perp, i, rank\rangle_{\sigma_{r_k}}$. Clearly, this allows each replica to sign on 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 the *rank* and k.

We are now ready to present an optimized version of CERBERUS for round $n \neq 1$.

In the 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 let $ppremesg.rank = k_m + rankmsg.rank + 1$.

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

In the COMMIT phase, a replica calculates the difference between the highest rank it knows and the *rank* of current round as k, *i.e.*, $k \leftarrow curRank.rank - commsg.rank$. 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.

Upon receiving a rank message, a replica updates its *curRank* if $rankmsg.rank + rankmsg.k > curRank.rank$.

Once again, Cerberus does not increase communication complexity compared with the original PBFT protocol [8]. The basic protocol is of complexity $O(n^2)$, and the optimized protocol does not increase the communication complexity for each phase.

5.4 Correctness Analysis

In this section, we provide a brief security analysis of Cerberus. Due to space constraints, we leave detailed proofs to Appendix A in [31]. We prove that Cerberus satisfies *agreement*, *totality*, *liveness*, and *causality* properties [1].

Proof sketch. To show agreement, we use the proof-by-contradiction method. Suppose that two replicas globally confirm two different blocks B and B' with the same *sn*. According to BAB-Agreement, all honest replicas will have the same view of partially confirmed blocks. The replicas will globally order the partially confirmed blocks using the same rules. Based on the globally ordering rules, B and B' must have the same *rank* and instance index if they are assigned the same *sn*. According to the MR-Monotonicity, the *rank* on the same instance is strictly monotonic, meaning that B and B' are the same block. This contradicts our assumption. To establish totality, we first prove that all partially confirmed blocks will eventually be globally confirmed. If a replica globally confirms a block B, it must have partially confirmed it. According to BAB-Agreement, all honest replicas will partially confirm B. Therefore, all honest replicas will globally confirm B. Regarding liveness, our bucket rotation mechanism ensures that all transactions will eventually be assigned to an honest leader for processing. According to BAB-validity, the transactions will be partially confirmed and eventually globally confirmed. As for causality, we prove that if block B is partially confirmed before the generation of B', then $B'.rank$ must be greater than $B.rank$. According to the global ordering rules, this is indeed the case.

6 PERFORMANCE EVALUATION

In this section, we evaluate the performance and causality of Cerberus in different scenarios, and compare Cerberus against ISS [42], a state-of-the-art Multi-BFT protocol. We implemented Cerberus in Go [2] and used the Go BLS library for aggregate signatures. In Cerberus, consensus instances are instantiated by PBFT [9]. With our experiments, we answer the following questions:

- **Q1:** How does Cerberus perform in the presence of stragglers? (Section 6.2)
- **Q2:** How does Cerberus resist front-running attacks as compared to ISS with stragglers? (Section 6.3)
- **Q3:** How much overhead does Cerberus introduce for realizing dynamic global ordering as compared to ISS? (Section 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 Cerberus in LAN and WAN. For LAN, each machine is equipped with one private network interface with a bandwidth of 1Gbps. For WAN, machines span 4 datacenters across France, America, Australia, and Tokyo on Amazon cloud. Each machine is equipped with two network interfaces, the public and private, and both bandwidths are limited to 1 Gbps. In particular, in WAN, we use the public interface for client transactions. We use NTP for clock synchronization across the servers.

System settings. Each transaction carries 500 bytes payload, which is the same as the average transaction size in Bitcoin [32]. 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 setting a limited proposing rate for PBFT. This prevents the leader from trying to propose too many batches in parallel, which will trigger a view change timeout. This measure limits peak throughput but is effective at protecting against unnecessary view changes. Specifically, the batch rate is set to 8 $batches/s$. We allow a large batch size of 4096 transactions to prevent rate-limiting the throughput. The epoch length is fixed at 64 ranks for both protocols since a shorter epoch length reduces latency in case a fault occurs while increasing latency due to epoch change.

Straggler settings. We simulate stragglers by following ISS, in which a straggling leader can delay its proposals for a specific time without triggering any time-out mechanisms. In our experiments, straggling leaders are randomly selected, and their proposing rates are fixed to $1/k$ of normal leaders. The parameter k is adjustable. For example, in WAN with 8 replicas, honest replicas propose a new block every 1s, and the straggling leaders propose a new block every 5s (*i.e.*, $k = 5$). Note that as mentioned in Section 2.3, straggling leaders are only one possible reason for straggler instances.

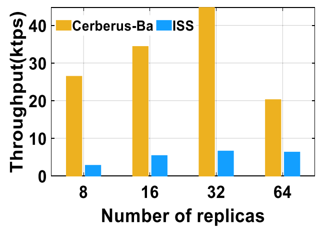
6.2 The Impact of Stragglers

We evaluate two performance metrics: 1) throughput: the number of transactions delivered to clients per second, and 2) latency: the average end-to-end delay from the moment clients submit transactions until they receive $f + 1$ responses. We let the straggler leaders not add transactions in its proposals to harm throughput and latency.

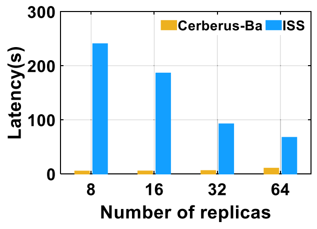
6.2.1 Performance in WAN. We evaluate throughput and latency of Cerberus and ISS with stragglers in WAN.

Varying number of replicas. Figure 6 shows the throughput and latency of Cerberus-Ba and ISS with varying number of replicas. Specifically, there is one straggler, and the number of replicas varies among 8, 16, 32, and 64. The straggling leader's proposing rate is set to be $1/5$ of normal leaders. The results show that with the increasing number of replicas, the throughput of both Cerberus and ISS first increases and then decreases after reaching certain peaks. This is because at the beginning, increasing replicas can prompt the parallelism of the system; however, after reaching the peak value, more replicas will lead to more message overhead, resulting in more wasted bandwidth. We also note that the throughput of Cerberus-Ba consistently outperforms that of ISS, by a factor of up to 8.5. The latency of Cerberus-Ba is consistently lower than that of ISS, by approximately 84.7% to 97.9%.

[1]The causality property assumes an infinite epoch, or the case before the epoch end, as we have made some specific measures to handle the last block of each epoch instance.
[2]https://github.com/JeffXiesk/cerberus



(a) Throughput.



(b) Latency.

Figure 6: Throughput and latency of Cerberus-Ba and ISS under one straggler with different numbers of replicas in WAN.

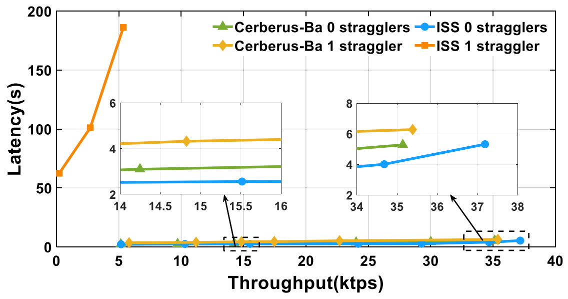
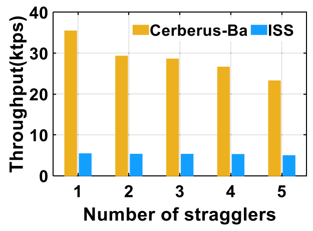


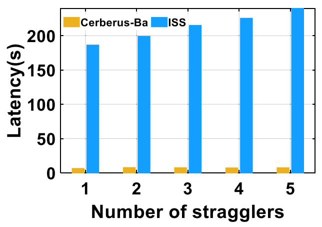
Figure 7: Latency over throughput of Cerberus and ISS with/without stragglers in WAN.

Throughput vs. latency. Figure 7 evaluates Cerberus-Ba and ISS while increasing the throughput until system saturation. For this experiment, there are 16 replicas, and the number of stragglers is set to zero or one. The results show that the impact of stragglers on Cerberus-Ba's performance is slight, resulting in only a 25% reduction in throughput and an 18.9% increase in latency. On the contrary, with one straggler, the maximum throughput of ISS decreases by 85.6% of that without stragglers, while the latency is increased by 34x before saturation. All of these illustrate that Cerberus-Ba can significantly improve the performance of the Multi-BFT protocols with stragglers.

Varying number of stragglers. Figure 8 evaluates the impact of stragglers on the performance of Cerberus and ISS with a varying number of stragglers. For this experiment, there are 16 replicas, and the number of stragglers ranges from one to five. From the figure, we observe that as the number of stragglers increases, the throughput of both protocols decreases, and the latency increases. Compared to the scenario with one straggler, the throughput of Cerberus decreases by 34.5% while the latency increases by 10.9% when there are 5 stragglers. In ISS, the throughput decreases by 8.6%, and latency increases by 28.7% with 5 stragglers. We observe that multiple stragglers in ISS do not hurt the throughput and latency much compared with one straggler. This is because the system performance is limited by the slowest straggler, as discussed in Section 2.1.



(a) Throughput.

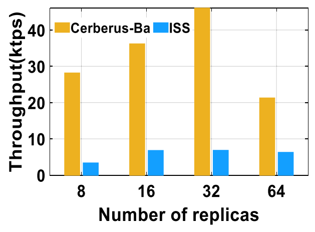


(b) Latency.

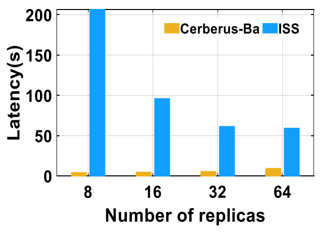
Figure 8: Throughput and latency of Cerberus-Ba and ISS with different numbers of stragglers in WAN.

6.2.2 *Performance in LAN.* We evaluate the throughput and latency of Cerberus and ISS with stragglers in LAN.

Varying number of replicas. Figure 9 shows the throughput and latency of Cerberus-Ba and ISS with a varying number of replicas. Specifically, there is one straggler, and the number of replicas varies among 8, 16, 32, and 64. The straggling leader's proposing rate is set to be 1/5 of normal leaders. Similar to the results in WAN, with the number of replicas increases, the throughput of both Cerberus and ISS first increases and then decreases after reaching certain peaks. We also note that the throughput of Cerberus-Ba consistently outperforms that of ISS, by a factor of up to 7.5. The latency of Cerberus-Ba is consistently lower than that of ISS, by approximately 85.5% to 97.1%.



(a) Throughput.



(b) Latency.

Figure 9: Throughput and latency of Cerberus-Ba and ISS under one straggler with different number of replicas in LAN.

Throughput vs. latency. Figure 10 evaluates Cerberus-Ba and ISS while increasing the throughput until system saturation. For this experiment, there are 16 replicas, and the number of stragglers is set to zero or one. The results show that the impact of stragglers on Cerberus-Ba's performance is slight, resulting in only a 2.4% reduction in throughput and a 29.5% increase in latency. By contrast, with one straggler, the maximum throughput of ISS decreases by 81.8% of that without stragglers, while the latency increases by 36.9x before saturation.

6.3 Resistance to Front-Running Attacks

We evaluate the security of Cerberus by studying the system's resistance to front-running attacks.

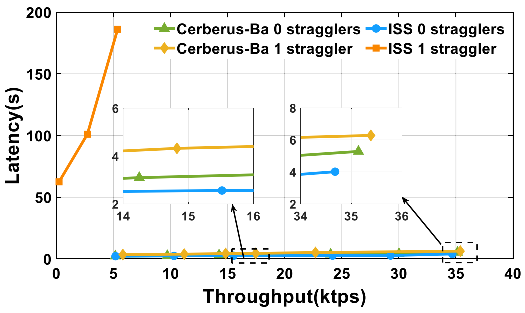


Figure 10: Latency over throughput of Cerberus and ISS with/without stragglers in LAN.

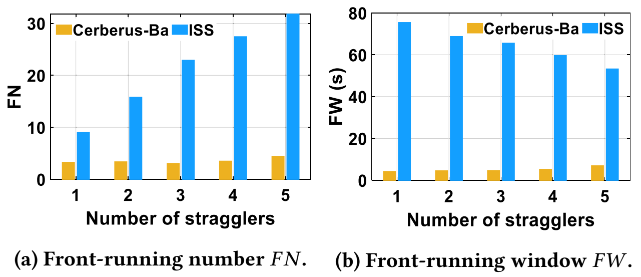
Metrics. We define two metrics front-running Number (FN) and front-running window (FW) as:

- **Front-Running Number FN.** Assume that a series of n blocks $\{B_1, B_2, ..., B_n\}$. If $i < j$, but B_i is proposed after B_j, we say B_i front runs B_j. The number of blocks front running B_j is denoted as FN.
- **Front-Running Window FW.** The time difference between the partial confirmation and global confirmation of a block is denoted as FW.

The front-running number FN measures the severity of the result of a block being front-run. The front-running window FW measures the potential risk of a block being front-run. Since higher FN and FW values indicate a weaker resistance to front-running attacks, one would like to minimize these metrics to improve the system's security.

Attack strategy. The adversary delays the instances with smaller indexes to launch front-running attacks to maximize its inflicted damage. Here, we change the propose delay of the adversary to study its impact on FN and FW.

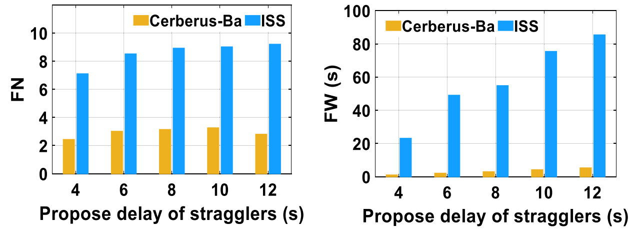
Evaluation results. We evaluate the FN and FW of Cerberus and ISS by considering two factors, namely the number of stragglers and the propose delay of the stragglers. We conduct experiments on WAN and 16 replicas.



(a) Front-running number FN. (b) Front-running window FW.

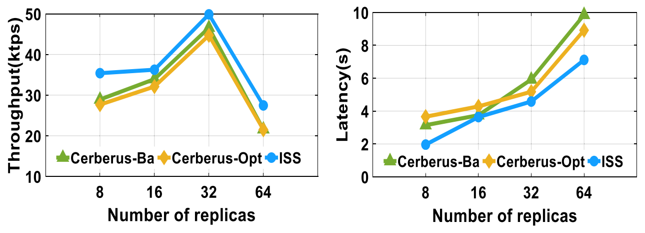
Figure 11: Front-running number FN and Front-running window FW of Cerberus and ISS with different numbers of stragglers.

Figure 11 shows the FN and FW of Cerberus and ISS with different numbers of stragglers. As depicted in Figure 11a, as the number of stragglers increases from 1 to 5, FN of ISS increases from approximately 10 to 30, while FN of Cerberus remains stable between 3 and 4, significantly lower than ISS. As depicted in Figure 11b, the variation of FW of both Cerberus and ISS is insignificant as the number of stragglers increases from 1 to 5. However, Cerberus consistently maintains a much lower FW compared to ISS (lower by up to 94.5%). This observation indicates that Cerberus is highly resilient to front-running attacks even as the number of stragglers increases.



(a) Front-running number FN. (b) Front-running window FW.

Figure 12: Front-running number FN and Front-running window FW of Cerberus and ISS with different propose delays of stragglers.



(a) Throughput. (b) Latency.

Figure 13: Throughput and latency of Cerberus-Ba, Cerberus-Opt and ISS.

Figure 12 shows the FN and FW of Cerberus and ISS with different proposing delays of stragglers. As depicted in Figure 12a, the variation of FN of both Cerberus and ISS is insignificant as the straggler delays the proposal from 4s to 12s. Still, Cerberus consistently maintains a much lower FN (2-3) compared to ISS (7-9). As depicted in Figure 12b, as the straggler delays the proposal from 4s to 12s, FW of ISS increases from approximately 23s to 85s, while FW of Cerberus stays low within 5s, which is significantly lower than ISS. This observation indicates that Cerberus is highly resilient to front-running attacks even as the proposing delay of stragglers increases.

6.4 Overhead Measurement

Figure 13 evaluates the performance of Cerberus-Ba, Cerberus-Opt and ISS without stragglers in WAN to measure the overhead. In particular, we measure the peak throughput before saturation and the associated delay. The results show that Cerberus-Ba and Cerberus-Opt have almost the same throughput and latency. This is because, compared with Cerberus-Ba, Cerberus-Opt reduces the message size, resulting in lower message delay, but, at the same time, it introduces additional computation time. In addition, the throughput and latency of Cerberus-Ba are slightly lower and higher than these of ISS, respectively. Based on these results, we demonstrate that Cerberus and Cerberus-Opt only incur minimal overhead. Also, we notice that all three protocols show an increasing trend in throughput from 8 to 32 replicas, followed by a decrease at 64 replicas. The latency increases with the number of replicas.

7 RELATED WORK

Existing leader-based BFT protocols, such as PBFT [10], Zyzzyva [28] and HotStuff [45], 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.

7.1 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, 40–42], in which several consensus instances run in parallel to handle transactions. Stathakopoulou *et al.* [40, 41] 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 [42] 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.

Another solution in this category is to use a DAG structure, in which each block contains hash references to multiple predecessor blocks. Example protocols include Narwhal-HotStuff [13], DAG-Rider [26], and Bullshark [38]. In particular, DAG-Rider and Bullshark are designed under asynchronous network conditions, and so are more complicated than Multi-BFT protocols. DAG-based BFT is shown to have poor efficiency under the assumption of strict consistency [44], and still suffer from the bottleneck caused by the ordering algorithm.

Summary. Multi-BFT consensus is simple and has high-performance in the ideal settings. However, as analyzed in Section 2, Multi-BFT systems suffer from severe performance issues and security issues.

7.2 Reducing Committee Size

The key insight behind this approach is to reduce the number of participants in the consensus protocols, avoiding the leader bottleneck of BFT consensus in large-scale settings. The representative solution is to randomly select a small group of replicas as the subcommittee, who is responsible for validating and ordering new transactions. This solution has been adopted by Algorand [20]. Instead of using one subcommittee, the sharding solution takes one step further and divides replicas into several disjoint subcommittees. Subcommittees run BFT protocols in parallel to process clients' transactions, resulting in better efficiency than the single subcommittee solution. Due to the promising performances, many BFT sharding protocols, such as Elastico [30], OmniLedger [27] and RapidChain [46], are proposed.

Summary. Reducing committee size can significantly prompt the scalability of BFT systems, however, it also wakens the fault tolerance of the system and introduces additional complexity. First, systems using this approach can tolerate a lower fraction of Byzantine replicas (*e.g.*, tolerating 25% Byzantine replicas in Algorand rather than 33%). Second, the subcommittee formulation, and the state synchronization between subcommittees make the system much more complicated.

7.3 Optimizing Message Transmission

The key insight behind this approach is to balance the transmission workloads between the leader and other replicas, making the best utilization of their network bandwidth. Specifically, the message transmission topology can be divided into two classes: the structured and unstructured. The representative solution using unstructured transmission topology is gossip protocol [6], in which a replica sends its messages to some randomly sampled replicas. Gossip has been used in Tendermint [5], Gosig [29], and Stratus [19], which can remove the leader bottleneck in large-scale settings. By contrast, in structured topology, each replica sends its messages to a fixed set of replicas after configuration. For example, in Kauri [33], replicas disseminate and aggregate messages on trees, and in Hermes [25], the leader sends blocks to an impetus committee, which helps to relay the block and vote messages.

Summary. Despite the improved performance, this approach also introduces new performance overhead and security issues. First, this method will significantly increase message transmission latency in a small network setting (less than thousands of replicas) due to the increasing number of transmission hops. Second, this approach will introduce new attack vectors for message transmission.

8 CONCLUSION

We propose Cerberus, a Multi-BFT protocol to effectively eliminate the impact of stragglers on performance and preserve inter-block causality. The insight is that in prior work, blocks across different instances are assigned pre-determined sequence numbers, to set their position in the global order. We propose dynamic global ordering to assign a sequence number to blocks according to the real-time status of all instances. 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 and pipeline their formulation with the consensus process. Furthermore, we adopt the aggregate signature to reduce the size of rank information. Our evaluation shows that Cerberus has demonstrated significant advantages over ISS in the presence of stragglers. Specifically, Cerberus achieved up to 8x increase in throughput and reduced latency by 98%, while exhibiting robustness against front-running attacks.

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on Ethereum smart contracts (SoK). *IACR Cryptology ePrint Archive* 2017 (2017), 764.

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

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

[4] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Advances in Cryptology — EUROCRYPT 2003*, Eli Biham (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 416–432.

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

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

[7] Miles Carlsten, Harry Kalodner, S. Matthew Weinberg, and Arvind Narayanan. 2016. On the Instability of Bitcoin Without the Block Reward. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 154–167.

[8] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186.

[9] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

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

[11] Chainlink. 2022. Another Advantage of DAG-based BFT: BEV Resistance. https://malkhi.com/posts/2022/07/dag-fo/

[12] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 177–190.

[13] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. https://doi.org/10.48550/ARXIV.2105.11827

[14] Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. 2000. *Atomic Broadcast In A Byzantine Model*. Springer Netherlands, Dordrecht, 179–195. https://doi.org/10.1007/978-94-015-9608-4_14

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* (1988).

[16] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2020. SoK: Transparent dishonesty: front-running attacks on blockchain. In *International Conference on Financial Cryptography and Data Security*. Springer, 170–189.

[17] Ittay Eyal and Emin Gün Sirer. 2018. Majority is Not Enough: Bitcoin Mining is Vulnerable. *Commun. ACM* 61, 7 (June 2018), 95–102.

[18] Fangyu Gai, Ali Farahbakhsh, Jianyu Niu, Chen Feng, Ivan Beschastnikh, and Hao Duan. 2021. Dissecting the performance of chained-BFT. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 595–606.

[19] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. 2022. Devouring the Leader Bottleneck in BFT Consensus. *arXiv preprint arXiv:2203.05158* (2022).

[20] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies *(SOSP '17)*. ACM, New York, NY, USA, 51–68.

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

[22] Tiantian Gong, Mohsen Minaei, Wenhai Sun, and Aniket Kate. 2022. Towards overcoming the undercutting problem. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 444–463.

[23] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT Protocols *(EuroSys '10)*. New York, NY, USA, 363–376.

[24] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1392–1403.

[25] Mohammad M. Jalalzai, Chen Feng, Costas Busch, Golden G. Richard, and Jianyu Niu. 2022. The Hermes BFT for Blockchains. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2022), 3971–3986. https://doi.org/10.1109/TDSC.2021.3114310

[26] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. https://doi.org/10.48550/ARXIV.2102.08325

[27] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. 583–598.

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative Byzantine fault tolerance. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 45–58.

[29] Peilun Li, Guosai Wang, Xiaoqi Chen, Fan Long, and Wei Xu. 2020. Gosig: a scalable and high-performance Byzantine consensus for consortium blockchains. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 223–237.

[30] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 17–30.

[31] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Ivan Beschastnikh, Yinqian Zhang, and Chen Feng. 2023. *Cerberus: High-Performance and Secure Multi-BFT Consensus via Dynamic Global Ordering*. Technical Report. https://github.com/JeffXiesk/technical-reports/Cerberus.pdf

[32] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Working Paper* (2008).

[33] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 35–48. https://doi.org/10.1145/3477132.3483584

[34] J. Niu and C. Feng. 2019. Selfish Mining in Ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1306–1316.

[35] Malay Kumar Patra, Subhendu Rakshit, and Neeraj Kumar Sharma. 2017. On the security of Byzantine fault tolerant consensus protocols against front-running attacks. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*. ACM, 206–218.

[36] Michel Raynal and Mukesh Singhal. 1996. Logical time: Capturing causality in distributed systems. *Computer* 29, 2 (1996), 49–56.

[37] United States. Securities, Exchange Commission. Special Study of the Options Markets, United States. Congress. House. Committee on Interstate, and Foreign Commerce. 1979. *Report of the Special Study of the Options Markets to the Securities and Exchange Commission*. US Government Printing Office.

[38] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. https://doi.org/10.48550/ARXIV.2201.05677

[39] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J Alex Halderman. 2014. Security analysis of the Estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 703–715.

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

[41] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2022. Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research (JSys)* 2, 1 (2022).

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

[43] Christof Ferreira Torres, Ramiro Camino, and Radu State. 2021. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. *arXiv preprint arXiv:2102.03347* (2021).

[44] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. 2020. SoK: Diving into DAG-based blockchain systems. *arXiv preprint arXiv:2012.06128* (2020).

[45] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness *(PODC '19)*.

[46] M Zamani, R Jurdak, Y Liu, B O'Flynn, and P Dutta. 2018. RapidChain: A Fast and Scalable Blockchain Protocol. In *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 1–10.

A PROOF OF CORRECTNESS

In this section, we provide sketch of proofs of Cerberus protocol properties.

A.1 Correctness of Monotonic Rank

In this section, we prove that Cerberus satisfies the two properties of monotonic ranks: MR-Agreement and MR-Monotonicity.

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

Proof. By BAB-Agreement, if an honest replica partially confirms a block $B =< txs, index, round, rank >$, then all honest replicas eventually partially confirms $B =< txs, index, round, rank >$. □

Theorem 2 (MR-Monotonicity). *The ranks of later generated blocks are larger than those of a partially confirmed block.*

Proof. Assuming a block B' is generated after B is partially confirmed. At least $f+1$ honest replicas have known $B.rank$ when B is partially confirmed. When the leader collects $2f+1$ rank messages for B', at least one of them is from the $f+1$ honest replicas who have known $B.rank$. So $B'.rank \geq B.rank + 1 > B.rank$. □

A.2 System Properties

Lemma 1. *If an honest replica partially confirms a block B, it will eventually globally confirm B.*

Proof. A partially confirmed block B will be globally confirmed if it satisfies the conditions in 4.2. 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 continually propose valid blocks until the *maxRank* of current epoch. By BAB-Validity and BAB-Agreement, all honest replicas eventually partially confirm the block with *maxRank*. Thus, eventually $B^*.rank = maxRank$. For the candidate block B'', $B''.rank \leq maxRank < maxRank + 1 = B^*.rank + 1$. So B'' will be eventually globally confirmed. Since $\mathcal{S}$ is finite, each block will eventually become the candidate block. So all blocks in $\mathcal{S}$ will be eventually globally confirmed. □

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

Proof. At the commit phase of B^i_j, replicas have finished the prepare phase, which means they have known $B^i_j.rank$. So all the honest replicas will send $rank \geq B^i_j.rank$ to the leader. Since the leader will collect $2f+1$ *rank*s from different replicas, at least $f+1$ of them are from those honest replicas. So $B^i_{j+1}.rank \geq B^i_j.rank + 1 > B^i_j.rank$. □

Theorem 3 (Agreement). *If two honest replicas globally confirm $B.sn = B'.sn$, then $B = B'$.*

Proof. By BAB-Agreement, all honest replicas will eventually have the same view of partially confirmed blocks. By Lemma 1, all partially confirmed blocks will be golbally confirmed. Assuming $B \neq B'$ and $B.sn = B'.sn$. Since blocks are ordered by increasing *rank* values, with tie-breaking favoring smaller instance index, we have 1) if $B.rank \neq B'.rank$, $B.sn \neq B'.sn$; 2) if $B.rank = B'.rank$ and $B.index \neq B'.index$, $B.sn \neq B'.sn$. Thus, if $B.sn = B.sn$, we have $B.rank = B'.rank$ and $B.index = B'.index$, which is contradicted with Lemma 2. So $B = B'$. □

Theorem 4 (Totality). *If an honest replica globally confirms B, then all honest replicas eventually globally confirm B.*

Proof. If an honest replica globally confirms B, it must have partially confirmed B. By BAB-Agreement, if an honest replica partially confirmed B, all honest replicas eventually partially confirm B. By Lemma 1, all honest replicas eventually globally confirm B. □

Lemma 3. *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 instance i refuses to propose tx, tx will be left in $\hat{b}$. According to the buckets rotation policy, $\hat{b}$ will be assigned to another instance in the next epoch. Thus, $\hat{b}$ will be eventually assigned to an instance with an honest leader, who will propose a batch contains tx. □

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

Proof. If a correct client broadcasts a transaction tx, by Lemma 3, some honest replica eventually proposes B with $tx \in B.txs$. By BAB-Validity, the replica eventually partially confirms B. By Lemma 1, the replica eventually globally confirms B. □

Theorem 6 (Block causality). *If a block B is partially confirmed by any honest replicas before the creation of block B', then block B is globally confirmed before block B' (i.e., $B.sn < B'.sn$).*

Proof. If a block B is partially confirmed by any honest replicas before the creation of block B', by MR-Monotonicity, $B.rank < B'.rank$. By BAB-Validity and BAB-Agreement, all honest replicas partially confirmed $B.rank < B'.rank$. Since blocks are ordered by increasing *rank* values, $B.sn < B'.sn$. By Lemma 1, all honest replicas globally confirmed $B.sn < B'.sn$. □