Orthrus: Accelerating Multi-BFT Consensus through Concurrent Partial Ordering of Transactions

Hanzheng Lyu*, Shaokang Xie†, Jianyu Niu‡, Ivan Beschastnikh§, Yinqian Zhang‡, Mohammad Sadoghi†, Chen Feng*

University of British Columbia (*Okanagan Campus, §Vancouver Campus)
†University of California, Davis, ‡Southern University of Science and Technology
*{hzlyu@student.ubc.ca, chen.feng@ubc.ca}, †{skxie,msadoghi}@ucdavis.edu
‡niujy@sustech.edu.cn, §bestchai@cs.ubc.ca

Abstract—Multi-Byzantine Fault Tolerant (Multi-BFT) consensus allows multiple consensus instances to run in parallel, resolving the leader bottleneck problem inherent in classic BFT consensus. However, the global ordering of Multi-BFT consensus enforces a strict serialized sequence of transactions, imposing additional confirmation latency and also limiting concurrency. In this paper, we introduce Orthrus, a Multi-BFT protocol that accelerates transaction confirmation through partial ordering while reserving global ordering for transactions requiring stricter sequencing. To this end, Orthrus strategically partitions transactions to maximize concurrency and ensure consistency. Additionally, it incorporates an escrow mechanism to manage interactions between partially and globally ordered transactions. We evaluated Orthrus through extensive experiments in realistic settings, deploying 128 replicas in WAN and LAN environments. Our findings demonstrate latency reductions of up to 87% in WAN compared to existing Multi-BFT protocols.

Index Terms—Byzantine fault tolerance, Multi-BFT consensus, Blockchain, Leader bottleneck, Partial ordering.

(a) Multi-BFT paradigm   (b) Breakdown latency

Figure 1: **(a) Multi-BFT paradigm.** The jth block produced by instance i is denoted as B_j^i. **(b) Breakdown latency with a straggler.** The green bar refers to transaction transmission delay (① and ④), the orange refers to the delay of a block being delivered from consensus (②), and the blue refers to the global ordering delay (③).

I. INTRODUCTION

Byzantine Fault Tolerant (BFT) consensus has received renewed interest due to its adoption in blockchain applications [1]–[6] ranging from cryptocurrency [7]–[10] to Decentralized Finance (DeFi) [11]. BFT consensus enables a network of replicas to agree on the same sequence of transactions, effectively mitigating double-spending attacks [12]—where a user might attempt to use the same asset in multiple transactions—even in the presence of Byzantine replicas that can behave maliciously. Most BFT consensus protocols (*e.g.*, PBFT [13]) employ a leader-based scheme, where a designated leader coordinates with other replicas (referred to as backups) to reach an agreement on its proposals (*e.g.*, transactions). However, this leader-based approach faces a significant performance bottleneck [14]–[17]: the leader's workload scales linearly with the number of replicas, which in turn impacts system throughput and latency.

To mitigate this bottleneck, Multi-BFT consensus protocols enable multiple leader-based consensus instances to run in parallel [15]–[19]. In these protocols, each replica simultaneously acts as the leader for one instance and a backup for others. Replicas run each instance to agree on a sequence of blocks, which are then ordered into a global sequence, as shown in Fig. 1a. By the global ordering, Multi-BFT consensus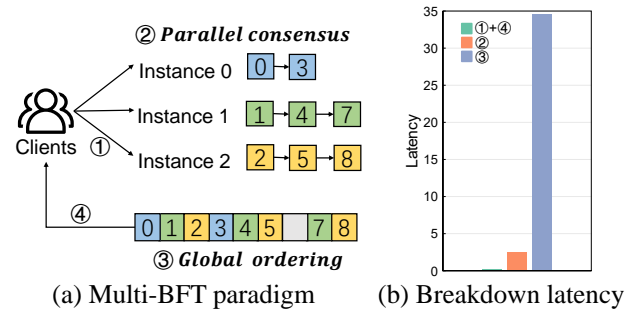 appears as a single BFT instance, but distributes workload among replicas to better utilize bandwidth and increase throughput. For example, ISS [16], a state-of-the-art Multi-BFT protocol, demonstrates significant performance improvements over single-leader protocols, achieving up to 37× and 56× better throughput for PBFT and HotStuff, respectively, on a network of 128 replicas. RCC [17], another Multi-BFT protocol, improves upon ISS by optimizing the recovery mechanism.

Our motivation. The global ordering mechanism in Multi-BFT consensus, while essential for ensuring consistency, significantly limits concurrency and can increase latency. By enforcing a strict, serialized sequence across all transactions, global ordering prevents replicas from processing non-conflicting transactions concurrently, even when they target different records and could be processed in parallel. For example, consider two independent payment transactions: one in which Alice transfers funds from her account and another in which Bob transfers funds from his account. In existing Multi-BFT consensus protocols [15]–[19], both transactions would need to wait for a globally agreed-upon order before execution, blocking them from concurrent processing despite having no direct conflicts.

Second, global ordering increases the overall system latency,

particularly when there are straggler instances that are much slower than other instances. As illustrated in Fig. 1a, if Instance 0 is a straggler, the missing block (*i.e.*, block 6) creates a gap in the global log, which then prevents subsequent blocks (*i.e.*, blocks 7 and 8) from being executed until the missing block appears. As illustrated in Fig. 1b, the latency breakdown for ISS [16] with 16 replicas in a WAN setting shows that global ordering delays can account for up to 92.8% of the total latency when there is a straggler instance that is $10\times$ slower than other instances. Specifically, while the consensus instance delivers a transaction in about 2.5 seconds, the global ordering process takes significantly longer—an additional 34.5 seconds—highlighting the substantial delay caused by global ordering. (Further evaluation results are presented in Sec.VII.)

The expensive global ordering significantly affects the performance of Multi-BFT, especially with stragglers. Thus, this raises a vital question: *How to minimize the use of global ordering in Multi-BFT consensus?*

Our approach. We observe that there are two primary classes of transactions in blockchain-based financial applications: *conflict-free transactions* (*e.g.*, payments with different payers) [20], [21] and *general non-commutative transactions* (*e.g.*, smart contracts modifying the same record). The conflict-free transactions inherently enable concurrent execution. Notably, conflict-free transactions account for a significant proportion of the overall transactions. For instance, more than 46% of transactions in Ethereum are conflict-free payment transactions. To optimize performance, we present Orthrus, which accelerates the processing of conflict-free transactions by introducing concurrency, while ensuring support for general non-commutative transactions, which may require stricter ordering due to dependencies between operations. Unlike traditional systems where a transaction can only be executed once it has received a global order and all preceding blocks in the global sequence have been executed, our approach allows for a more relaxed execution condition.

Conflict-free transactions. We often observe that there are two types of operations in payment transactions: incremental (e.g., adding funds) and decremental (e.g., withdrawing funds). Incremental operations are inherently commutative, as they only increase the balance of the accounts. Decrements in different accounts are also commutative, as they do not affect each other's state. This means that many transactions can be executed concurrently without using strict ordering.

To accomplish this, Orthrus assigns client transactions to instances based on the payer, ensuring that transactions from the same payer are processed within the same instance. This *transaction partition* mechanism introduces concurrency and prevents double spending [12]. However, some transactions may involve multiple payers. In such cases, the transaction is assigned to multiple instances, each instance managing the actions of one payer, allowing the system to handle complex transactions without sacrificing concurrency. However, this approach also introduces a challenge in maintaining transaction atomicity, as some payers might succeed while others fail,

potentially leaving the transaction in an incomplete state.

To ensure atomicity, we draw inspiration from the *escrow transactions* [22] to design an *escrow mechanism* which temporarily reserves the transfer amounts for each payer in an escrow state. The transaction is only confirmed if all escrow requests are successfully committed. If any escrow request fails, the reserved funds for all other escrow requests are also released, ensuring transactional atomicity and consistency across instances.

General non-commutative transactions. Beyond conflict-free transactions, there are also general transactions—such as those involving smart contracts—that include non-commutative operations. For these, we enable confirmation through global ordering to ensure safety and consistency. This hybrid design allows Orthrus to maximize concurrency for commutative transactions while maintaining the necessary cross-instance guarantees to support general non-commutative transactions. To facilitate seamless interaction between contract transactions (which require strict global ordering) and standard payment transactions, we implement the previously mentioned *escrow mechanism*. This mechanism allows contract-related payments to be temporarily escrowed within the partial log, preventing them from delaying payment transaction confirmations. Consequently, payment transactions can proceed without interruption, while contract transactions maintain the necessary ordering constraints.

We build an end-to-end prototype of Orthrus in Go [23] and conduct an extensive evaluation on AWS. We use a real-world dataset with 200,000 transactions from 18,000 active users on the Ethereum network. This dataset provides a more accurate representation of transaction behaviors, which leads to more valid and reliable results. We compare Orthrus with the state-of-the-art Multi-BFT consensus, including ISS [16], Mir-BFT [18], RCC [17], DQBFT [24] and Ladon [19] in terms of throughput and latency.

Our contributions. To summarize, this paper makes the following contributions:

- We introduce Orthrus, a Multi-BFT consensus that leverages partial ordering of transactions to enhance concurrency while maintaining consistency across instances.

- We propose a novel escrow mechanism to ensure transaction atomicity and seamless integration of partially ordered and globally ordered transactions.

- We evaluate Orthrus with extensive experiments over WAN and LAN with 8-128 replicas distributed across 4 regions. In the WAN setting, Orthrus achieves at most 87% lower latency than other protocols on 128 replicas with a straggler. The LAN setting exhibits a similar trend.

II. MOTIVATIONS

We first examine blockchain scenarios where partial ordering is sufficient for transactions and then highlight the necessity of hybrid ordering. Transactions in blockchains typically fall into two categories: *payment* (*i.e.*, conflict-free

transactions) and *contract transactions* (*i.e.*, general transactions). Payment transactions are straightforward exchanges, by which clients transfer funds. Each payment transaction involves incremental and decremental operations on payers' and payees' accounts. Due to the commutative characteristic of these operations, partial ordering is sufficient.

In contrast, contract transactions represent more complex interactions involving various operations beyond simple transfers. Specifically, the operations on state variables can be accessed by multiple clients. Unlike payment transactions, contract transactions often include non-commutative operations where the order in which they are performed affects the result, such as assignments. This distinction between payment and contract transactions highlights Orthrus's dual approach: maximizing concurrency for payment transactions while employing global ordering for contract transactions to maintain system consistency.

A. Why Partial Ordering Works?

In a typical payment transaction, decremental operations reduce the payers' account balance, while incremental operations increase the payees' account balance. By confining transactions that involve the same payer to a single instance, transactions across instances can be executed concurrently without global ordering. To better understand the benefits of partial ordering, let's consider three simple single-payer, single-payee transactions: tx_1: Alice $\rightarrow$ Carol, tx_2: Bob $\rightarrow$ Carol, and tx_3: Alice $\rightarrow$ Bob, where $\rightarrow$ denotes the transfer tokens from the lefter to the righter. Based on the three transactions, we have the following two observations.

- *Observation 1:* If two transactions have different payers and neither transaction affects the balance of the other's payer, they can be executed concurrently.

  Example (independence): Transactions tx_1 and tx_2 have different payers and do not affect each other's payer balance. Even though both transactions share Carol as a payee, they have different payers, Alice and Bob. Since tx_1 and tx_2 consume tokens from different payer accounts, their execution order does not impact the outcome; both transactions succeed regardless of whether tx_1 or tx_2 executes first.

- *Observation 2:* If one transaction may affect the balance of the other transaction's payer, they may need to be executed sequentially.

  Example (dependency): The payee of tx_3 is the payer of tx_2. The transaction tx_3 affects Bob's balance, and tx_2 depends on Bob's balance to succeed. If Bob has sufficient balance, tx_2 and tx_3 can be executed concurrently. If Bob does not have enough balance to support tx_2 before tx_3 executes, then tx_3 must be executed first to ensure that tx_2 can succeed.

  Example (confliction): Transactions tx_1 and tx_3 have the same payer. Since both transactions share Alice as the payer, they directly reduce her account balance. If Alice has sufficient balance, tx_1 and tx_3 can be executed concurrently. If Alice has limited funds, the order of tx_1 and tx_3 becomes crucial:

executing tx_1 first may deplete her balance, resulting in a successful transfer to Bob but causing tx_3 to fail. Conversely, if tx_3 is executed first, tx_1 may fail due to insufficient funds.

These examples illustrate that we can enforce effective partial ordering to manage transaction dependencies and prevent conflicts. Transactions involving the same payer are assigned to the same instance and executed sequentially within that instance, ensuring consistency. For dependencies across instances, Orthrus establishes a cross-instance partial order by generating references between dependent transactions. These designs guarantee that if one transaction affects the payer's balance in a way that impacts another transaction's execution, the two transactions are processed in the correct sequence to preserve consistency.

For transactions involving multiple payers and payees, each transaction can be split into multiple single-payer, single-payee sub-transactions, thereby adhering to the above partial ordering rules. To guarantee atomicity in multi-payer transactions, Orthrus employs an escrow mechanism discussed in a later section. This incurs minimal overhead, as each sub-transaction can be processed concurrently, and the escrow mechanism adds only a slight processing cost.

B. Why Needs *Hybrid* Ordering?

While partial ordering suffices for payment transactions, contract transactions often involve complex, non-commutative operations (*e.g.*, assignments) that require stricter ordering. The outcome of these operations depends on execution order, necessitating global ordering to maintain consistent results.

- *Observation 3:* Transactions modifying the same record must be executed sequentially to preserve consistency.

Example: Transactions tx_1 and tx_2 modify the same record. Assume that transaction tx_1 assigns $value_1$ to record o, while transaction tx_2 assigns $value_2$ to record o. Here, the order between tx_1 and tx_2 is crucial: executing tx_1 first and then tx_2 results in o having $value_2$, while reversing the order leaves o with $value_1$.

Although transactions modifying distinct records can execute concurrently without ordering, most contract transactions involve non-commutative operations that inherently depend on execution order. Therefore, a global ordering mechanism is essential to maintain consistency across replicas and ensure that all contract transactions yield the same result.

The above examples underscore the necessity of hybrid ordering in Orthrus. For payment transactions, partial ordering maximizes concurrency, while contract transactions with complex operations rely on global ordering to ensure consistency across replicas.

III. SYSTEM MODEL AND PROPERTIES

A. System Model

We consider a system composed of $n = \{r_i\}_{i=0}^{n-1}$ replicas, denoted as the set $\mathcal{N}$. We assume a subset of up to f replicas are *Byzantine*, represented by the set $\mathcal{F}$, where $|\mathcal{F}| \leq f$ and

$n \geq 3f + 1$. The remaining replicas, denoted as $\mathcal{H} = \mathcal{N} \setminus \mathcal{F}$, are honest and strictly follow the protocol. All Byzantine replicas are assumed to be controlled by a single adversary, which is computationally bounded and cannot break cryptographic primitives. Each replica r_i has a public/private key pair (pk_i, sk_i) established through a public-key infrastructure (PKI), enabling them to sign and verify messages.

Network model. Honest replicas are connected through authenticated point-to-point channels. We adopt the partial synchrony model introduced by Dwork et al. [25], commonly used in BFT consensus protocols [13], [26]. This model defines an unknown Global Stabilization Time (GST), after which the system behaves synchronously with a known message delivery bound Δ. Formally, for any two honest replicas $r_i, r_j \in \mathcal{H}$, and any time $t \geq$ GST, a message sent from r_i to r_j at time t will be delivered by time $t + \Delta$.

B. Data Model

Object. We follow the object-centric design [27], [28], in which objects provide operations that can be called by transactions to examine and modify the object's state. These objects are long-lived, like accounts, and are represented as $o = (key, value, op, con, type)$. Here, key is a cryptographically unique identifier, and $value$ denotes the object's current state, which can be updated as operations op are performed. The op attribute specifies the operation to be executed on the object (*e.g.*, increment or decrement). The con attribute sets a condition that must be satisfied following the execution of any operation. The $type$ attribute categorizes objects as either *owned* or *shared*.

1) Owned objects. They are associated with a specific owner, with key corresponding to the owner's address. Owned objects support two operations: incremental and decremental. The decremental operation requires the owner's authorization through a digital signature. For example, Alice's account object, which holds her balance, is an owned object. If Alice wishes to transfer tokens, she must authorize the transaction with her digital signature.

2) Shared objects. They have no specific owner and can be processed by anyone with authorization in the smart contract. Shared objects may support additional operations such as complex state changes or contract-specific actions.

Transaction. Transactions are defined as $tx = (O, id, \sigma)$, where O denotes the set of objects o involved in tx, each specifies an involved object and the specified operation op to be performed. The id field provides a unique identifier for each transaction. σ is the signature, included for owned objects requiring authorization. Each transaction involves at least one owned object, as every transaction must be initiated by a client, whose account is classified as an owned object.

Transactions are categorized into two types: payment and contract transactions. Payment transactions involve only owned objects, while contract transactions may involve both objects, allowing for greater flexibility in executing complex operations under a smart contract framework.

Block. A block is a tuple $b = (txs, ins, sn, S, \sigma)$, where: txs represents a batch of transactions and ins indicates the specific instance to which the block belongs, which helps in identifying which instance is responsible for processing the transactions in the block. sn denotes the sequence number of the block, which maintains the order of blocks within an instance. S denotes the system state (defined in Sec. III-D) on which the block depends, which suggests the state under which the transactions within the block can be executed successfully. σ is the cryptographic signature on B, which provides authenticity and integrity to the block.

C. Core Components

Sequenced broadcast (SB). Sequenced broadcast is a critical variant of Byzantine total order broadcast [29] that ensures the total ordering of transactions and preserves the consistency among honest replicas. The SB operates through two fundamental primitives: *broadcast* and *deliver*. A replica acts as the leader to broadcast a transaction with a sequence number, and all replicas collaborate to deliver the transaction with the sequence number. There is a failure detector that can detect leader failures. SB guarantees that an honest replica will eventually deliver a transaction for every sequence number sn (*i.e.*, *termination property*) and that all honest replicas will deliver the same transaction with the same sn (*i.e.*, *agreement property*). In this work, we utilize SB protocol as a black box that inputs client transactions and outputs a sequence of delivered transactions.

Escrow method. The escrow method is a concurrency control and data consistency mechanism originally developed to handle high-contention scenarios in database systems, particularly when aggregate quantities, such as inventory levels, must be managed under high transactional loads with minimal locking [22]. In the Escrow method, operations are divided into incremental and decremental actions, each modifying a resource quantity without directly conflicting with others. By assigning each transaction an escrow balance, the system can accommodate multiple requests on the same resource simultaneously, as long as the escrowed quantities suffice for each transaction's demand. If a transaction aborts, the escrowed quantity is simply returned to the available pool. In this work, we leverage the Escrow method to handle concurrent transaction execution in replicated state environments.

D. System Goals

We consider a Multi-BFT system composed of m SB instances, indexed from 0 to $m - 1$, which take clients' transactions as input. Each instance has a leader who broadcasts transactions with sequence numbers and coordinates all replicas to deliver a sequence of transactions. Transactions are executed after being partially ordered within an SB instance or globally ordered across instances. A transaction is confirmed once it is executed, either successfully or unsuccessfully. The system's state S can be represented as a tuple, where each element corresponds to the maximum sequence number sn of
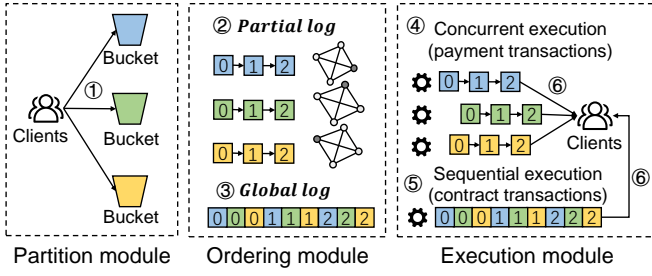
4

Figure 2: **An overview of Orthrus.** Transactions are partitioned into distinct buckets, ordered according to their types, and finally executed.

an SB instance. Specifically, the state of the Multi-BFT system is given as $S = (sn_0, sn_1, \ldots, sn_{m-1})$, where sn_i denotes the maximum sequence number for the instance indexed by i. The state of the system in the view of replica r_i is denoted as S_{r_i}. The system must ensure the following properties:

- **Safety**: If two honest replicas reach the same state S, they must have consistent values for all objects.
- **Liveness**: Given a transaction tx from a correct client c, tx is eventually confirmed by all honest replicas.

IV. Orthrus Overview

A. System Architecture

The architectural overview of Orthrus is provided in Fig. 2, which consists of three modules: partition module, ordering module, and executions module.

Partition module. It employs the bucket mechanism first introduced by Mir-BFT [18] to partition client transactions into buckets, with each bucket assigned to a unique instance. Each transaction is assigned to one or more buckets based on the objects it contains. Specifically, we define an assignment function assign that maps each owned object to a specific bucket. For example, assign may compute a bucket index for an object o as the hash of the key modulo m, where m is the total number of instances. Furthermore, the function can also be designed to balance loads across instances and minimize cross-instance interactions. Transactions containing decremental operations on owned objects are assigned to the corresponding buckets of those objects.

Ordering module. It contains multiple SB instances and a global ordering algorithm. Each instance independently selects a set of transactions from its corresponding bucket as input and outputs a sequence, or *partial log* of transactions, agreed upon by all replicas within that instance. Transactions in a partial log are *partially ordered*, meaning they are ordered within an instance. Payment transactions can be output in partial logs as soon as they are partially ordered, while contract transactions require a global order. The global ordering algorithm merged the partial logs from all instances into a *global log*, which imposes a *global order* across all transactions in the system.

Execution module. It processes the ordered transactions produced by the ordering module. Payment transactions from

each partial log can be executed concurrently, leveraging their partial ordering for parallel processing. In contrast, contract transactions in the global log are executed sequentially to maintain consistency across all instances. The execution results are then sent back to the clients.

B. Transaction Workflow

Each transaction in Orthrus follows a workflow through the partition, ordering, and execution modules. Below, we outline the processing flow for a single transaction in Orthrus.

① Upon receiving a transaction, a replica assigns it to one or more buckets in the partition module, based on the involved owned objects (Sec. V-A).

② The transaction is then packed into blocks and broadcast within the SB instances, which eventually deliver it with a designated sequence number (Sec. V-B).

③ Contract transactions delivered by the instances are merged into a global log to establish a global order, while payment transactions can bypass this step (Sec. V-B).

④ Payment transactions from partial logs are executed concurrently (Sec. V-C).

⑤ Contract transactions in the global log are executed sequentially (Sec. V-C).

⑥ Once successfully executed, the transaction is confirmed, and a response is sent to the client.

C. Challenges and Solutions

This section discusses two key challenges of the hybrid ordering design of Orthrus and corresponding solutions.

Challenge-I: Ensuring atomicity of transactions. A payment transaction involving multiple owned objects with decremental operations (*i.e.*, multiple payers) may be assigned to multiple instances according to the partition module. If one payer executes the payment in their instance successfully while another payer fails in theirs, the transaction loses atomicity, leading to inconsistent states across instances. Maintaining atomicity across multiple instances, therefore, poses a significant challenge.

Solution-I: Escrow mechanism for atomicity. We design an escrow mechanism inspired by the escrow method proposed in [22], but simplify the escrow process by focusing on the temporary reservation of funds. When a transaction with multiple payers is assigned to different instances, each instance performs an escrow operation on its respective payer's balance, temporarily deducting the amount from each payer. If all escrow requests within the transaction are successfully committed, these temporary deductions become permanent. However, if any escrow request fails, all escrows associated with this transaction are canceled, and the amounts are refunded to the respective payers. Once a replica observes that all payers in a transaction have successfully escrowed their amounts, it commits all escrow requests. If any escrow request fails, it aborts all escrowed requests for the transaction, ensuring atomicity is preserved.

5

Challenge-II: Avoiding payment transactions blocking.
When a contract transaction and a subsequent payment transaction share the same payer, the contract transaction may block the payment transaction due to the need for global ordering. Since the contract transaction must be globally ordered for consistency across all instances, any dependent transactions involving the same payer are delayed until the contract transaction is confirmed. This dependency leads to inefficiencies and increased latency for the payment transaction.

Solution-II: Escrow mechanism to avoid blocking. To address this issue, we extend the escrow mechanism to allow payment transactions to proceed without being blocked by pending contract transactions. When a replica encounters a contract transaction involving a payer, it escrows the required amounts from the payer. This approach allows subsequent payment transactions involving the same payer to be evaluated and processed as though the contract transaction's decremental operation has been executed, thus avoiding blocking. Once the contract transaction reaches global order and is confirmed, the replica either commits the escrowed amounts if the transaction executes successfully or refunds the amounts if it fails. This solution ensures that contract transactions do not block subsequent payments, improving system efficiency while maintaining consistency.

V. ORTHRUS ALGORITHM

In this section, we present the core algorithm of Orthrus, as illustrated in Algorithm 1. Overall, Orthrus operates in epochs, following the design in [16], [19], with specific sequence numbers assigned to each instance in each epoch. A replica ends an epoch only after confirming all sequence numbers assigned to it within that epoch. In each epoch, Orthrus handles several key processes: transaction partitioning (Sec. V-A), partial and global ordering (Sec. V-B), and transaction execution (Sec. V-C). At the end of each epoch, it creates checkpoints and performs garbage collection (Sec. V-D).

A. Partition Transactions

Upon receiving a transaction tx, replica r adds tx to one or more buckets based on its involved objects. First, it verifies the validity of the transaction's format and checks the owner's signature. Once verified, replica r calls the $\mathsf{assign}(o)$ function to determine the appropriate bucket(s) for tx. Specifically, the replica iterates through each object involved in the transaction. For each object, if the object is of type *owned* and the transaction's operation on the object is decremental, the replica calls $\mathsf{assign}(o)$ to determine the appropriate bucket index for the object. After obtaining the bucket index, tx is pushed into the corresponding bucket. Each bucket is an append-only list for backups but allows both push and pull operations for its leader. If tx is already in a bucket, it will not be added again to avoid duplication.

B. Order Transactions

We employ two types of logs for transaction ordering: the partial log, denoted as $plog$, and the global log, denoted as

---

Algorithm 1 Orthrus Algorithm for replica r

1: **upon** initialize system
2:    **for** $i \in [0, m-1]$ ▷ m is the number of instances
3:       **if** $\mathsf{isLeader}(i, r)$ **then** ▷ r is the leader of $instance_i$
4:          **for** $sn \in \{0, 1, 2, ...\}$
5:             $b.S \leftarrow \mathsf{currentState}$
6:             $b.txs \leftarrow \mathsf{pullValidTx}(bucket_i, b.S)$
7:             $b.ins \leftarrow i; \; b.sn \leftarrow sn; \; b.\sigma \leftarrow \mathsf{sign}(b, r)$
8:             **trigger**$\langle$sb-broadcast$|b\rangle$
9:
10: **upon** receive tx ▷ partition transactions
11:    **if** $\mathsf{validateTx}(tx)$ **then**
12:       **for** $o \in tx.O$ **where**
        $o.type = owned \wedge o.op = decremental$
13:          $i \leftarrow \mathsf{assign}(o)$
14:          $\mathsf{push}(tx, bucket_i)$
15:
16: **upon event** $\langle$sb-deliver$|b\rangle$ ▷ order transactions
17:    $plog[b.ins][b.sn] \leftarrow b$
18:    $\mathsf{globalOrder}(b, glog)$
19:
20: **upon** $\mathsf{firstPending}(plog[i]) \neq \perp$ ▷ execute transactions
21:    $tx \leftarrow \mathsf{firstPending}(plog[i])$
22:    **for** $o \in tx.O$ **where**
      $\mathsf{assign}(o) = i \wedge o.type = owned \wedge o.op = decremental$
23:       $\mathsf{escrow}(o, tx)$
24:       **if** $(o, tx) \notin elog$
25:          $\mathsf{abortEscrow}(tx)$
26:          $\mathsf{abort}(tx)$ ▷ remove tx from all logs
27:       **else if** $\mathsf{allEscrowed}(tx) \leftarrow true$
           $\wedge tx.type = payment$
28:          $\mathsf{commitEscrow}(tx)$
29:          **for** $o \in tx.O$ **where** $o.op = incremental$
30:             $o.value \leftarrow \mathsf{apply}(o, tx.o.op)$
31:
32: **upon** $\mathsf{firstPending}(glog) \neq \perp$
33:    $tx \leftarrow \mathsf{firstPending}(glog)$
34:    **if** $\mathsf{isLastPosition}(tx, glog)$
35:       **if** $\mathsf{exe}(tx) = true \wedge \mathsf{allEscrowed}(tx) = true$
36:          $\mathsf{commitEscrow}(tx)$
37:       **else**
38:          $\mathsf{abortEscrow}(tx)$
39:          $\mathsf{abort}(tx)$ ▷ remove tx from $glog$
40:    **else**
41:       $\mathsf{remove}(tx, glog)$ ▷ remove this tx from $glog$

$glog$. Each log consists of multiple entries, with each entry capable of holding a batch of transactions. Each instance maintains its $plog$, which is used for the partial ordering of transactions within that instance. Meanwhile, the system as a whole maintains a single $glog$ to ensure a global ordering of transactions across the system.

We treat the SB as a black box, interacting with it through two defined events: $\langle$sb-broadcast$|b\rangle$ and $\langle$sb-deliver$|b\rangle$. Here,

Algorithm 2 Escrow method

```
1:  function escrow(o, tx)
2:      value ← apply(o, o.op)
3:      if value ≥ o.con
4:          o.value ← value
5:          append((o, tx), elog)
6:
7:  function allEscrowed(tx)
8:      allEscrowed(tx) ← true
9:      if ∃ o ∈ tx.O s.t. ((o.type = owned)∧
            o.op = decremental) ∧ ((o, tx) ∉ elog)
10:         allEscrowed(tx) ← false
11:
12: function commitEscrow(tx)
13:     for o ∈ tx.O ∧ (o, tx) ∈ elog
14:         remove((o, tx), elog)
15:
16: function abortEscrow(tx)
17:     for o ∈ tx.O ∧ (o, tx) ∈ elog
18:         undo(o, o.op)
19:         remove((o, tx), elog)
```

$\langle\text{sb-broadcast}|b\rangle$ represents the event where a block b is broadcasted into the SB, while $\langle\text{sb-deliver}|b\rangle$ signifies the event when the SB delivers the block b after ordering.

Broadcast transactions. Upon initializing Orthrus, for each instance, if replica r is the leader of $instance_i$, it enters a loop in which it creates a block b for each sequence number sn in the current epoch and then broadcasts b in the SB instance. In each iteration, the leader creates a block b and references the current system state S, which consists of the sequence number of the last block on each instance. Using this state S, the leader pulls a specified number of the oldest transactions from the instance's bucket. These transactions are valid under S, meaning that S provides a consistent baseline against which each transaction's prerequisites are satisfied. Backup replicas can check the validation of transactions based on this state. If there are insufficient transactions to meet this number, the leader waits for a timeout and then pulls available transactions from the bucket. The leader then sets the instance index and sequence number for the block, signs it, and broadcasts it in the SB instance, where all replicas participate in ordering and delivering the block. If r is not the leader, it participates in the SB instances as a backup.

Deliver transactions. Upon delivering a block b from an SB instance with a sequence number, the replica orders the block by appending it to the partial log $plog$ at the instance index $b.ins$ and sequence number $b.sn$ as well as invokes the globalOrder function to append it to the global log $glog$. Each replica locally computes the global index of a block based on its parameters, which determine its position in the global order. To achieve this, Orthrus adopts the dynamic global ordering algorithm from Ladon [19], which is detailed in Appendix A.

Failure detector. In Orthrus, a failure detection module (also called the view-change mechanism) is integrated into the SB protocol. This module enables replicas to change the faulty leader for an SB instance and has been widely used in prior work [26], [30]–[32]. For instance, in PBFT, replicas begin a *view change* to replace the leader when suspecting the leader of Byzantine behavior. When l_i, the leader of $instance_i$ fails at sequence number sn, the recovery process involves three steps: 1) All honest replicas detect the failure of l_i. 2) All honest replicas agree on the state of $instance_i$ and the new leader $l_i^{'}$. 3) All honest replicas restart $instance_i$ from sn.

The failure detector is used to deal with censorship attacks and spoofing attacks, In a censorship attack, Byzantine leaders can selectively ignore or censor certain transactions. To address this, we require that a client broadcast the transaction tx to at least $f+1$ replicas, where f is the maximum number of faulty replicas. This ensures that at least one honest replica will receive and push tx to the corresponding bucket. If the leader fails to propose tx within a reasonable period, other replicas can detect this failure and request a leader replacement. In a spoofing attack, a Byzantine leader might broadcast a block referencing an incorrect state S. If a replica finds that a transaction within the block is invalid based on the specified state S, it detects the leader's failure. The replica can also request the blocks it missed in the state S from the leader. Should the leader be unable to provide these blocks, it detects the leader's failure as well. Additionally, we limit the number of messages that one replica can send to another within a given period to prevent Denial of Service (DoS) attacks.

C. Execute Transaction

To introduce the transaction execution process, we first provide an overview of the escrow mechanism, a foundational component of Orthrus 's execution phase.

Escrow mechanism. Each replica maintains an escrow log $elog$ to manage the escrow of objects and transactions. The Escrow mechanism in Orthrus is shown in Algorithm 2.

- The escrow function attempts to perform an escrow operation on a single object o within the transaction tx. It first calculates the $value$ by applying the operation $o.op$ on o's current state. If the resulting $value$ satisfies the condition $o.con$, the function updates $o.value$ and appends this escrow request to the $elog$.

- The allEscrowed function is used to check whether owned objects in tx with the decremental operation have been successfully escrowed. It initializes $allEscrowed(tx)$ as $true$, then iterates over each object o in tx. If an object with a decremental operation and owned type has not been escrowed, $allEscrowed(tx)$ is set to $false$, and the function breaks out of the loop early.

- The commitEscrow function is called to remove all the escrow requests of tx from the $elog$.

- The abortEscrow is called to undo and remove all the escrow requests of tx in the $elog$.

The replica monitors the $plog$s and $glog$ to identify the next transaction ready for execution, referred to as the first pending transaction in the log. The first pending transaction is defined as the transaction for which all preceding transactions in the log, with smaller indexes, have been confirmed.

Execute transactions in $plog$**.** Upon identifying the first pending transaction tx in $plog[i]$, for each object o in tx, if o is an owned object belongs to block b in the current instance with a decremental operation, the replica attempts to perform an escrow on o for tx. The escrow is performed on the system state $b.S$ referred to by the transaction or any subsequent state derived from it through valid updates. If the escrow operation on o fails, all escrows in tx are aborted. Then the transaction tx is aborted, which means tx is removed from all the partial logs $plog$ and the global log $glog$. If all required escrows for tx are successful and tx is a payment transaction, the system commits all escrows in tx, and proceeds to other objects in tx with incremental operations by applying the incremental operation to update $o.value$ accordingly.

Execute transactions in $glog$**.** When a replica identifies the first pending transaction tx in $glog$, it first checks if this is the last occurrence of tx in $glog$, as a transaction may appear in multiple positions within $glog$. If it is not the last occurrence, tx is simply removed from the current position in $glog$. If it is the last occurrence, the replica proceeds to execute tx and verify whether all escrow requests within the transaction are successful. If both the execution and escrow checks are successful, the replica commits all escrow requests associated with tx. If any escrow request fails, all escrows in tx are aborted, and tx is removed from $glog$ in all its occurrences. Notably, executing tx in the global log ($glog$) must strictly align with the global state at its designated position in $glog$.This ensures that tx is processed consistently across all replicas under the same system state, thereby maintaining overall consistency.

D. Checkpoint and Garbage Collection

Orthrus runs a simple checkpoint protocol at the end of each epoch. Upon epoch completion, each replica broadcasts a checkpoint message to all other replicas, containing a signed digest summarizing the blocks it processed during that epoch. When a replica receives a sufficient quorum of matching checkpoint messages (*i.e.*, from at least $2f + 1$ replicas), it creates a stable checkpoint, which serves as a verified snapshot of the epoch's blocks. This stable checkpoint enables the replica to securely discard the data from the completed epoch, allowing for efficient garbage collection. Additionally, unexecuted transactions can be discarded, preventing the retention of transactions that will never be executed.

VI. CORRECTNESS ANALYSIS

In this section, we prove the safety and liveness properties of Orthrus. For safety, we show that replicas in the same state execute an identical set of transactions, with payment transactions yielding consistent object values across replicas

due to their commutative nature, and contract transactions producing consistent results by being executed sequentially under a global ordering. For liveness, we demonstrate that any transaction received by an honest replica will eventually be proposed and delivered by its corresponding SB instances, with all instances achieving consistent execution outcomes (success or failure). As a result, each transaction will eventually be confirmed by all honest replicas.

Lemma 1. *If two honest replicas reach the same state S, they must successfully execute the same set of transactions.*

Proof. Let r_1 and r_2 be two honest replicas in the same state $S = (sn_0, sn_1, \ldots, sn_{m-1})$, where sn_i represents the maximum sequence number of transactions delivered by SB instance i. By the agreement property of the SB protocol, both replicas r_1 and r_2 must have delivered the same sequence of transactions for each SB instance up to sn_i. For contract transactions, which are globally ordered, both replicas execute them in the same sequence and under the same state, ensuring that the outcome is identical. For a payment transaction in a block b, which refers to the state $b.S$ agreed upon by all the honest replicas, the execution is guaranteed to succeed as long as both replicas execute them under $b.S$ or any subsequent state derived from it through valid updates. □

Lemma 2. *If two honest replicas successfully execute the same set of transactions T, they must have consistent final values for all owned objects involved in T.*

Proof. If two honest replicas r_1 and r_2 successfully execute the same set of transactions T. Let $O = \{o_1, o_2, \ldots, o_n\}$ be the set of all owned objects involved in T, each transaction $tx_i \in T$ has the form:

$$tx_i = \{(o_{i_1}, o_{j_1}, \Delta_1), (o_{i_2}, o_{j_2}, \Delta_2), \ldots, (o_{i_k}, o_{j_k}, \Delta_k)\},$$

where each tuple $(o_{i_p}, o_{j_p}, \Delta_p)$ denotes a decremental operation on object o_{i_p} and an incremental operation on object o_{j_p} by the same amount Δ_p. The final value of any object $o \in O$ after applying all transactions in T is:

$$value(o) = value_0(o) + \underbrace{\sum_{\substack{tx_i \in T, \\ (o, o_j, \Delta) \in tx_i}} \Delta}_{o \ is \ payee} - \underbrace{\sum_{\substack{tx_i \in T, \\ (o_i, o, \Delta) \in tx_i}} \Delta}_{o \ is \ payer}.$$

Here $value_0(o)$ is the initial value of o, the first sum accumulates all increments to o, and the second sum accumulates all decrements from o. Therefore, for any permutation L of T that ensures the successful execution of all transactions, executing in the order specified by L will result in the same final values for all objects in O. □

Lemma 3. *If two honest replicas successfully execute the same set of transactions T, they must have consistent final values for all shared objects involved in T.*

Proof. Since the execution order of contract transactions in T is the same for both replicas, we can consider the sequence

$tx_1, tx_2, \ldots, tx_n$ representing this fixed order. For each transaction tx_i in the sequence, all replicas apply tx_i at the same step in the execution. Let the value of any shared object o be $value_i(o)$ after transaction tx_i is executed on it. Because tx_i is deterministic and is executed at the same step on all replicas, any change to $value_i(o)$ will be identical across replicas. Thus, the final values for all shared objects involved in T will be consistent across replicas r_1 and r_2, provided they started with consistent initial values. $\square$

Theorem 1 (Safety). *If two honest replicas reach the same state S, they must have consistent values for all objects.*

Proof. Given that r_1 and r_2 are in the same state S ($S_{r_1} = S_{r_2}$), Lemma 1 ensures that both replicas successfully execute the same set of transactions. By Lemma 2 and Lemma 3, if the replicas have consistent initial values for all objects (owned and shared) involved in T, they must have consistent final values for those objects after execution. Thus, for all objects o, $value_{r_1}(o) = value_{r_2}(o)$. $\square$

Lemma 4. *For any transaction tx sent by a correct client c, tx is eventually broadcast in all SB instances it belongs to.*

Proof. For any transaction tx sent by a correct client c, the transaction is sent to at least $f + 1$ replicas, ensuring that at least one honest replica receives it and adds it to the relevant buckets. For each instance it belongs to, if the leader is honest, it will broadcast tx. If the leader is Byzantine, it will be detected and a new honest leader will be selected eventually. This new leader will broadcast tx from the bucket, ensuring that tx is eventually broadcast. $\square$

Lemma 5 (Atomicity). *For a payment transaction tx, either all its sub-transactions are executed successfully, or aborted.*

Proof. By the Lemma 4 and the termination property of the SB protocol, all sub-transactions of tx will eventually be delivered. Each sub-transaction is then processed using the escrow mechanism. According to this mechanism, if any sub-transaction fails during the escrow phase, all associated escrow requests of tx are aborted, meaning that all sub-transactions will be aborted. Conversely, if all escrow requests are successfully committed, all sub-transactions are executed successfully. Thus, the atomicity of the transaction is preserved. $\square$

Lemma 6. *For any transaction tx broadcast by an honest replica, tx is eventually confirmed by all the honest replicas.*

Proof. By the termination property of the SB protocol, a transaction broadcast by an honest replica will eventually be delivered by all honest replicas. For a payment transaction, all the sub-transactions will be executed or aborted after being delivered by the SB instances (Lemma 5). Thus, the payment transaction will be confirmed. Contract transactions are processed by the global ordering algorithm after being delivered by the SB instances. This algorithm traverses all delivered transactions to ensure they are globally ordered and confirmed. Consequently, all honest replicas will eventually confirm all delivered contract transactions. $\square$

Theorem 2 (Liveness). *Given a transaction tx from a correct client c, tx is eventually confirmed by all honest replicas.*

Proof. Assuming the transaction tx is sent by a correct client, by Lemma 4, tx will be broadcast in all SB instances it belongs to. Following this, according to Lemma 6, tx will be confirmed by all honest replicas. $\square$

VII. PERFORMANCE EVALUATION

In this section, we evaluate the performance of Orthrus in different scenarios and compare Orthrus against ISS [16], RCC [17], Mir-BFT [18], DQBFT [24], and Ladon [19], the state-of-the-art Multi-BFT protocols. We implemented Orthrus in Golang [23]. The evaluation results are illustrated using ChiPlot[1]. We use PBFT [30] consensus protocol to implement SB instances. With our experiments, we answer the following questions:

- **Q1:** How does Orthrus perform as compared with ISS, RCC, Mir-BFT, DQBFT, and Ladon with a varying number of replicas? (Sec. VII-B)
- **Q2:** How does Orthrus perform with different proportions of payment transactions? (Sec. VII-C)
- **Q3:** What is the latency breakdown of Orthrus as compared to ISS? (Sec. VII-D)
- **Q4:** How does Orthrus perform under faults? (Sec. VII-E)

A. Implementation and Experimental Setup

Implementation. Orthrus[2] is implemented in Golang and built on top of the ISS[3] [16] platform. Golang was chosen for its concurrency features and performance, and the ISS platform provides a solid foundation for its infrastructure. Our implementation is fully integrated with the ISS platform, ensuring compatibility and allowing for direct comparisons with other protocols developed on the same foundation.

Experimental setup. We deploy our protocols on AWS EC2 instances (c5a.2xlarge), with each instance representing a replica in our distributed system. Each instance is equipped with 8vCPUs, 16GB of RAM, and runs Ubuntu Linux 22.04. The experiments are conducted in both LAN and WAN environments. In the LAN setup, machines communicate over private network interfaces with a bandwidth of 1Gbps. For the WAN setup, instances are distributed across four Amazon cloud data centers located in France, the United States, Australia, and Tokyo, with network interfaces limited to 1Gbps. We utilize NTP for clock synchronization.

The dataset for our experiments is derived from the Ethereum blockchain, consisting of approximately 200,000 transactions extracted from blocks with heights ranging from 17,198,000 to 17,202,000. These transactions are drawn from a pool of 18,000 active accounts and include both payment and contract transactions, with payment transactions comprising

---

[1]https://www.chiplot.online/
[2]The source code is available at https://github.com/Hanzheng2021/Orthrus/.
[3]The source code for ISS is available at https://github.com/hyperledger-labs/mirbft/tree/research-iss.

(a) #Stragglers = 0, WAN   (b) #Stragglers = 0, WAN

(c) #Straggler = 1, WAN   (d) #Straggler = 1, WAN

Figure 3: **Throughput and latency of Orthrus, ISS, RCC, Mir, DQBFT, and Ladon in WAN.**

(a) #Stragglers = 0, LAN   (b) #Stragglers = 0, LAN
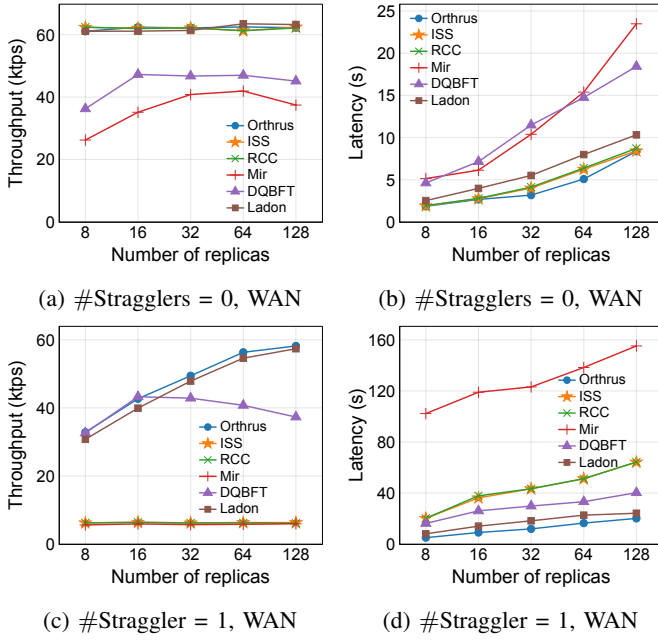

(c) #Straggler = 1, LAN   (d) #Straggler = 1, LAN

Figure 4: **Throughput and latency of Orthrus, ISS, RCC, Mir, DQBFT, and Ladon in LAN.**

46% of the transactions. This distribution is preserved in our experimental setup, using real transaction data as input. Once these transactions are processed, we reset the account states by introducing specific transactions and then re-execute the same set of 200,000 transactions to evaluate system performance under repeated workloads.

Each replica can function both as a leader for one instance and as a backup for others, i.e., $m = n$. To maximize throughput, we allow a large batch size of 4096 transactions, each carrying a payload of 500 bytes. We evaluate system performance under two different network conditions: with and without stragglers. In the straggler scenario, one instance operates at 10 times slower speed than the others. All experiments are repeated five times and the average results are reported.

B. Throughput and Latency

We evaluate the performance of Orthrus, ISS, RCC, Mir, DQBFT, and Ladon without stragglers and with one straggler in both We evaluate two performance metrics: 1) throughput: the number of transactions responded 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 measure the peak throughput in kilo-transactions per second (ktps) before reaching saturation, along with the associated latency in seconds (s).

1) Performance in WAN. Fig. 3a and Fig. 3c demonstrate that Orthrus consistently maintains throughput within the top tier. Without stragglers, Orthrus shows very similar throughput with ISS and RCC. With one straggler, Orthrus achieves comparable or superior throughput compared to dynamic global ordering Multi-BFT protocols. Additionally, Orthrus significantly surpasses pre-determined Multi-BFT protocols
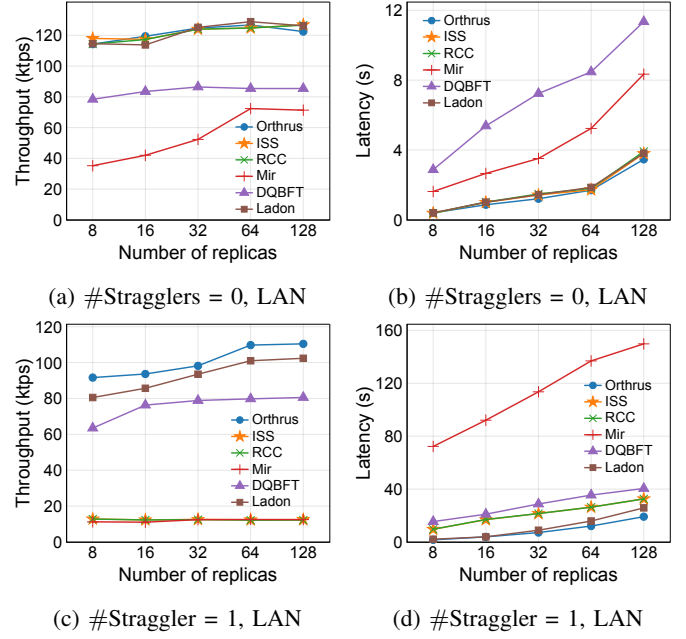
(*e.g.*, ISS, RCC, Mir), delivering $9.5\times$ higher throughput under the same conditions. Comparing Fig. 3a and Fig. 3c, we observe that a straggler significantly impacts pre-determined Multi-BFT protocols, causing a drop of 89.9% in throughput compared to those without stragglers on 128 replicas. Orthrus and Ladon were less affected by stragglers, experiencing only a 6.5% and 9.2% drop in throughput, respectively, compared to their performance without stragglers on 128 replicas.

Fig. 3b and Fig. 3d illustrate that Orthrus consistently achieves the lowest latency compared to other protocols. Without stragglers, Orthrus reduces the latency of 18.6% of ISS and RCC with 64 replicas, 18.9% of Ladon, 54.5% and 64.3% of DQBFT and Mir, on 128 replicas, respectively. With a straggler, the effect becomes more pronounced. On 128 replicas, Orthrus reduces the latency of 68.6% of ISS and RCC, 16.7% of Ladon, 50.0%, and 87.0% of DQBFT and Mir, respectively. By comparing Fig. 3b and Fig. 3d, we observe a similar impact of stragglers on latency, consistent with their effect on the throughput across different protocols.

2) Performance in LAN. Fig. 4 shows that Protocols show similar trends in LAN with WAN, with higher throughput and lower latency. Without stragglers and with a straggler, Orthrus always shows high throughput and low latency among all protocols. With one straggler, Orthrus demonstrates approximately $8\times$ higher throughput than ISS, RCC, and Mir, and 37.1% and 7.9% higher than DQBFT and Ladon, respectively, on 128 replicas. The latency is reduced by 16.7%, and 50.0% compared to Ladon and DQBFT, respectively, on 128 replicas.

The excellent performance of Orthrus is mainly due to its fast confirmation mechanism for simple payment transactions, which allows some transactions to avoid global ordering and thus reduces latency. In the absence of stragglers, the delay
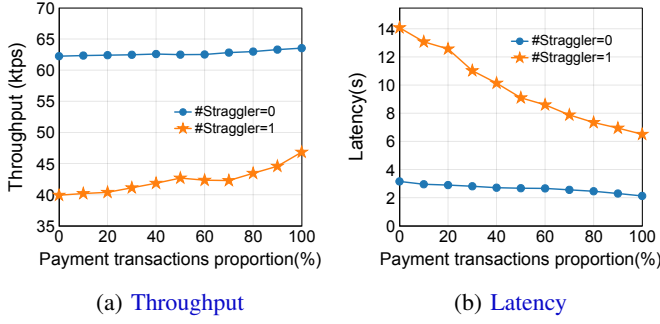
(a) Throughput   (b) Latency

Figure 5: **Throughput and latency of Orthrus under different payment transactions proportions in WAN.**

Figure 6: **Breakdown of latency in ISS versus Orthrus.**

(a) Throughput average (over 0.5s intervals) over time.

(b) Latency average (over 0.5s intervals) over time.

Figure 7: **Throughput and latency average of Orthrus over time with 0, 1, 5 faults. The faults occur at 9 seconds.**

introduced by global ordering is minimal, so Orthrus shows only a slight advantage. However, in the presence of stragglers, the delay caused by global ordering is significant, giving Orthrus a substantial advantage over pre-determined systems.

C. Varying Payment Transactions Proportions

Fig. 5 illustrates the performance of Orthrus with varying proportions of payment transactions. The experiments are conducted on 16 replicas in WAN with/without a straggler. As depicted in Fig. 5, the throughput increases and the latency decreases as the proportion of payment transactions rises. Specifically, without stragglers, when the proportion of payment transactions increases from 0% to 100%, the throughput increases by up to 2.1% and the latency decreases by 32.7%. However, with stragglers, the throughput sees a substantial increase of up to 17.3%, while latency drops by 53.8%. This pronounced change in the presence of stragglers is due to the higher proportion of payment transactions, allowing more transactions to be quickly confirmed, thereby greatly enhancing the overall processing efficiency.

D. Latency Breakdown

We present a detailed latency breakdown of Orthrus and ISS to understand the average spent time of different stages. We divide the process into five distinct stages: 1) Sending transactions: This stage spans the period from when the client sends the transaction until the replica receives it; 2) Preprocessing: This stage spans from receiving the transaction to broadcasting the transaction at the replica; 3) Partial Ordering: This stage extends from when the replica broadcasts a transaction to the delivery of the transaction; 4) Global Ordering: This begins with the delivery of the transaction and ends with the confirmation of the transaction; and 5) Reply: This final stage covers when $f + 1$ replicas confirm the transaction to when the client receives $f + 1$ replies.

The experiment was conducted in WAN with 16 replicas and one straggler. For each stage, the average time across all transactions is measured. The results are illustrated in Fig. 6. For the transaction sending and reply stages, the latency of Orthrus and ISS are comparable, indicating similar communication infrastructure efficiencies. A similar trend is observed in the preprocessing and partial ordering stages. One notable difference between the two protocols is evident in
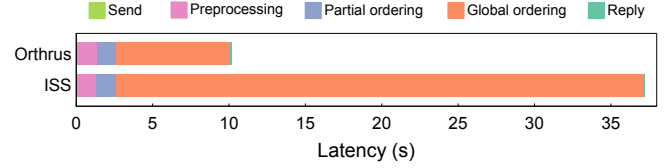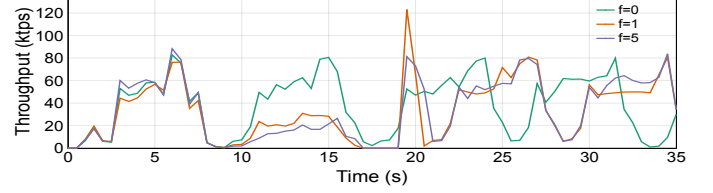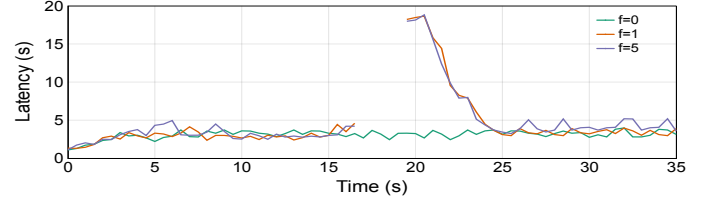
the global ordering stage, which exhibits a substantial delay in ISS, taking 34.5 seconds when a straggler is present and accounting for a remarkable 92.8% of the total latency.

This detailed breakdown demonstrates that the global ordering stage in ISS introduces a significant delay when stragglers are involved, becoming a bottleneck and exacerbating latency issues. In contrast, Orthrus achieves significantly lower latency in the presence of stragglers, taking only 7.4 seconds for the global ordering. This lower latency is attributed to Orthrus's efficient handling of transactions through its fast path mechanism, which minimizes the impact of slow global ordering. Additionally, the cost of global ordering can be further reduced as the proportion of payment transactions increases.

E. Performance Under Faults

In this section, we study the performance of Orthrus under faults in a WAN of 16 replicas, allowing up to 5 faulty replicas. Our analysis covers both detectable faults and undetectable faults. We evaluate scenarios with 0, 1, and 5 faulty replicas. The PBFT view change timeout is set at 10 seconds.

Detectable faults. We analyze how detectable faults impact the throughput and latency of Orthrus. Fig. 7 shows the throughput and latency average over time. In Fig. 7a, the short drops to 0 in throughput correspond to the epoch change. During the first epoch (0-9 seconds), no faults occur, and the system operates at its expected performance level.

At the start of the second epoch (9 seconds), faults are introduced, causing a significant drop in throughput by more

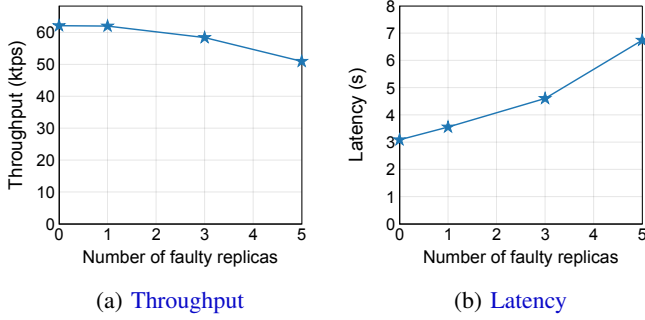(a) Throughput   (b) Latency

Figure 8: **Throughput and latency of Orthrus under different number of undetectable faulty replicas in WAN.**

than 50%. Specifically, when there is one faulty replica, the throughput drops by approximately 50%, whereas, with 5 faulty replicas, the decrease is slightly more pronounced. This behavior is influenced by the transaction composition: 54% of the transactions are contract-based and require global ordering, making them unconfirmed regardless of the number of faults. In contrast, partially ordered payment transactions are only affected in instances led by faulty replicas, meaning their impact scales with the number of faults. During the decline in throughput, latency remains nearly unchanged. This is because transactions confirmed in this phase are from instances led by honest replicas, which are less affected by faulty replicas.

Upon fault detection, the system initiates the view change process to recover, completing at 19 seconds. Following this, throughput rises sharply due to the confirmation of transactions in the global log. In Fig. 7b, latency exhibits a similar trend, as these confirmed transactions have been blocked for a significant duration, resulting in a high average latency. The brief interruption in the latency curve before 19 seconds corresponds to the period where throughput drops to zero in Fig. 7a, indicating that no transactions were confirmed during this time.

By the beginning of the third epoch (22 seconds), throughput recovers as the system stabilizes, and latency gradually stabilizes as well, demonstrating Orthrus 's resilience to detectable faults.

Undetectable faults. In this section, we evaluate undetectable Byzantine behaviors, where a replica avoids participating in other instances while continuing to propose blocks in the instance it leads, thereby preventing a leader timeout. This behavior does not trigger a view change. Fig. 8 presents the throughput and latency of Orthrus under different numbers of faulty replicas in a 16-replica WAN setting. As the number of faulty replicas increases, throughput declines moderately, while latency is impacted more significantly. This is because when faults reach the maximum threshold, system latency is primarily limited by the slowest honest replica.

VIII. RELATED WORK

In this section, we introduce prior works of single-leader BFT consensus protocols and Multi-BFT consensus protocols. Due to space constraints, we refer readers to Appendix C for works about the partial ordering of payment systems and other scaling approaches of BFT consensus.

Single BFT consensus. Castro and Liskov propose PBFT [13], the first practical BFT protocol that inspires many subsequent protocols [26], [31]–[39] Notable examples include Zyzzyva [33], Tendermint [34], and HotStuff [26]. These protocols adopt a leader-based scheme, which simplifies the procedure of reaching consensus on proposals, however, leads to a leader bottleneck limiting system scalability [40]–[42].

Multi-BFT consensus. Multi-BFT consensus allows replicas to operate multiple consensus instances in parallel, effectively addressing the leader bottleneck of single BFT consensus [16]–[19]. Mir-BFT [18] is among the first Multi-BFT protocols, which runs in epochs to globally order blocks delivered from instances by some pre-determined indices. An epoch change is triggered if any leader is faulty, making it vulnerable to Byzantine leaders. Later, ISS [16] improved on Mir-BFT by permitting replicas to deliver no-op messages to prevent frequent epoch changes. RCC [17] also adopts a pre-defined global ordering for delivered blocks. Thus, a straggler instance could delay the start of the next epoch for other replicas, leading to a slowdown of the entire system. DQBFT [24] uses a unique BFT instance to globally order output transactions from other instances to promote efficiency. However, the reliance on a single ordering BFT instance can be susceptible to targeted attacks [43]. Ladon [19] proposes monotonic ranks to dynamically order blocks across instances, mitigating the impact of slow replicas. These approaches reduce the latency caused by slow instances, but the global ordering phase still accounts for a significant proportion (almost half) of the system latency.

Unlike the above-mentioned protocols, TELL [44] optimizes the execution layer rather than consensus layer. TELL employs a State Hash Table (SHT) to track read/write sets, enabling concurrent execution of transactions across instances and within blocks. TELL resolves conflicts through re-execution and merges instances at the epoch level, which reduces execution waiting latency but limits its overall improvement in end-to-end client latency. In contrast, Orthrus tries to avoid cross-instance conflicts through a tailored transaction assignment strategy, eliminating the need for re-executing payment transactions to improve end-to-end latency.

IX. CONCLUSION

In this paper, we introduced Orthrus, a novel approach to enhance the performance of Multi-BFT protocols by sidestepping the limitations typically introduced by global ordering. Leveraging a hybrid design, Orthrus integrates partial ordering for conflict-free transactions and global ordering for general non-commutative transactions. We also introduce a customized escrow mechanism to ensure transaction atomicity and seamless interaction between partial and global logs. Our experiments show that in WAN of 128 replicas, Orthrus decreases latency by at least 68% as compared to pre-determined global ordering Multi-BFT protocols with one straggler.

REFERENCES

[1] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *NSDI*, 2024.

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Caper: a Cross-Application Permissioned Blockchain. In *PVLDB*, 2019.

[3] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global Scale Resilient Blockchain Fabric. In *PVLDB*, 2020.

[4] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Proof-of-Execution: Reaching Consensus Through Fault-Tolerant Speculation. In *EDBT*, 2021.

[5] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. Blockchains vs. Distributed Databases: Dichotomy and Fusion. In *SIGMOD*, 2021.

[6] Shlomi Dolev, Bingyong Guo, Jianyu Niu, and Ziyu Wang. SodsBC: A Post-quantum by Design Asynchronous Blockchain Framework. In *TDSC*, 2023.

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

[8] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[9] Marko Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Open Problems in Network Security*, 2016.

[10] Dahlia Malkhi. Blockchain in the Lens of BFT. In *ATC*, 2018.

[11] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. SoK: Decentralized Finance (DeFi). In *AFT*, 2022.

[12] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending Fast Payments in Bitcoin. In *CCS*, 2012.

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

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

[15] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. FnF-BFT: A BFT Protocol with Provable Performance under Attack. In *SIROCCO*, 2023.

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

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

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

[19] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Chen Feng, Yinqian Zhang, and Ivan Beschastnikh. Ladon: High-Performance Multi-BFT Consensus via Dynamic Global Ordering. In *EuroSys*, 2025.

[20] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without Proof of Work. In *FC*, 2016.

[21] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online Payments by Merely Broadcasting Messages. In *DSN*, 2020.

[22] Patrick E O'Neil. The Escrow Transactional Method. *TODS*, 1986.

[23] Golang. https://go.dev/.

[24] Balaji Arun and Binoy Ravindran. Scalable byzantine fault tolerance via partial decentralization. *VLDB*, 2022.

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

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

[27] William E Weihl. Commutativity-based Concurrency Control for Abstract Data Types. *IEEE Trans Comput*, 1988.

[28] Same Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *CCS*, 2024.

[29] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. Introduction to Reliable and Secure Distributed Programming. In *Springer Science & Business Media*, 2011.

[30] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *TOCS*, 2002.

[31] Yehonatan Buchnik and Roy Friedman. FireLedger: A High Throughput Blockchain Consensus Protocol. *VLDB*, 2020.

[32] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *DSN*, 2019.

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

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

[35] Jian Liu, Wenting Li, Ghassan O Karame, and Nadarajah Asokan. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput*, 2018.

[36] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, and Chen Feng. Byzantine Protocols with Asymptotically Optimal Communication Complexity. In *SecureComm*, 2023.

[37] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-Hotstuff: A Fast and Robust BFT Protocol for Blockchains. *TDSC*, 2023.

[38] Mohammad M Jalalzai, Chen Feng, Costas Busch, Golden G Richard, and Jianyu Niu. The Hermes BFT for Blockchains. *TDSC*, 2021.

[39] Jianyu Niu, Runchao Han, Shengqi Liu, Fangyu Gai, Ivan Beschastnikh, Yinqian Zhang, and Chen Feng. EBFT: Simplifying BFT Consensus Through Egalitarianism. In *arXiv preprint arXiv:2012.01636*, 2020.

[40] Salem Alqahtani and Murat Demirbas. Bottlenecks in Blockchain Consensus Protocols. In *COINS*, 2021.

[41] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *SIGMOD*, 2021.

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

[43] Ernesto Estrada. Network Robustness to Targeted Attacks. The Interplay of Expansibility and Degree Distribution. *EPJ B*, 2006.

[44] Xing Tong, Zheming Ye, Zhao Zhang, Cheqing Jin, and Aoying Zhou. TELL: Efficient Transaction Execution Protocol Towards Leaderless Consensus. In *ICDE*, 2024.

[45] Andrei Tonkikh, Pavel Ponomarev, Petr Kuznetsov, and Yvonne-Anne Pignolet. Cryptoconcurrency: (Almost) Consensusless Asset Transfer with Shared Accounts. In *CCS*, 2023.

[46] Jakub Sliwinski and Roger Wattenhofer. Asynchronous Proof-of-Stake. In *SSS*, 2021.

[47] Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-Performance Byzantine Fault Tolerant Settlement. In *AFT*, 2020.

[48] Andrew Lewis-Pye, Oded Naor, and Ehud Shapiro. Flash: An Asynchronous Payment System with Good-Case Linear Communication Complexity. In *arXiv preprint arXiv:2305.03567*, 2023.

[49] Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof. In *arXiv preprint arXiv:2006.12276*, 2020.

[50] Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and Asynchronous Asset Transfer. In *Distrib Comput*, 2023.

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

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

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

[54] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving DAG-BFT Latency and Robustness. 2024.

[55] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. In *FC*, 2024.

[56] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. MYSTICETI: Reaching the Latency Limits with Uncertified DAGs. In *NDSS*, 2025.

[57] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. In *NDSS*, 2018.

[58] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards Scaling Blockchain Systems via Sharding. In *SIGMOD*, 2019.

[59] Huawei Huang, Xiaowen Peng, Jianzhou Zhan, Shenyang Zhang, Yue Lin, Zibin Zheng, and Song Guo. BrokerChain: A Cross-Shard Blockchain Protocol for Account/Balance-based State Sharding. In *INFOCOM*, 2022.

[60] Jelle Hellings and Mohammad Sadoghi. Byshard: Sharding in a Byzantine Environment. In *VLDB*, 2021.

[61] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. Gridb: Scaling Blockchain Database via Sharding and Off-Chain Cross-Shard Mechanism. In *VLDB*, 2024.

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

[63] 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.

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

[65] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding Permissioned Blockchains over Network Clusters. In *SIGMOD*, 2021.

[66] Ramakrishna Kotla and Michael Dahlin. High Throughput Byzantine Fault Tolerance. In *DSN*, 2004.

[67] Aldenio Burgos, Eduardo Alchieri, and Fernando Dotti. On the Performance of Using Parallel State Machine Replication to Implement Blockchains. In *LADC*, 2021.

[68] Krishnasuri Narayanam, Akshar Kaul, Ken Kumar, and Pankaj Dayama. Partial Order Transactions on Permissioned Blockchains for Enhanced Scalability. In *BRAINS*, 2021.

[69] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. SpotLess: Concurrent Rotational Consensus Made Practical Through Rapid View Synchronization. In *ICDE*, 2024.

APPENDIX A
GLOBAL ORDERING ALGORITHM

In this section, we introduce the dynamic global ordering algorithm in Ladon [19], which is used in this paper. The dynamic global ordering algorithm requires two parameters to order blocks: the instance index and the rank. Given a set of delivered blocks with these parameters, honest replicas can independently execute the ordering algorithm to produce a consistent sequence of globally confirmed blocks without additional communication.

Rank. Before broadcasting a block, a leader first collects the highest ranks from at least $2f + 1$ replicas. It then increments the highest observed rank by one and assigns this value to the proposed block. The leader then broadcasts the block along with its assigned rank. Once consensus is reached, no additional procedure is needed to ensure agreement since the rank is piggybacked with the block.

The rank parameter has two key properties:

- **Agreement:** All honest replicas have the same rank for a delivered block.

- **Monotonicity:** If a block b' is generated after an intra-instance (or a delivered inter-instance) block b, then the rank of b' is larger than the rank of b.

Algorithm 3 The Global Ordering Algorithm

1: **function** globalOrder($b, glog$)
2:     append($b, \mathcal{W}$) //$\mathcal{W}$: blocks waiting to be confirmed
3:     $\mathcal{P}' \leftarrow$ getLastBlock($\mathcal{P}$) //$\mathcal{P}$: delivered blocks
4:     $b^* \leftarrow$ findLowestBlock($\mathcal{P}'$)
5:     $bar = (b^*.rank + 1, b^*.index)$ //compute the bar
6:     $bcan =$ findLowestBlock($\mathcal{W}$) //find candidate block
7:     **while** $b_{can} \prec bar$ //b_{can} has a lower index than bar
8:         append($b_{can}, glog$) //globally confirm b_{can}
9:         $\mathcal{W} \leftarrow \mathcal{W} \setminus b_{can}$ //update $\mathcal{W}$
10:         $b_{can} =$ findLowestBlock($\mathcal{W}$) //find next b_{can}
11:     **end while**
12: **end function**

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

A detailed description and formal proof can be found in [19].

Global ordering algorithm. Algorithm 3 shows the global ordering function running at a replica. The blocks are globally ordered by increasing ranks and a tie-breaking to favor blocks with smaller instance 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'.

When a block b is delivered, the globalOrder function is triggered to append the block b to $\mathcal{W}$, which is the set of blocks delivered by SB instances waiting to be globally ordered, *i.e.*, blocks delivered by SB instances but have not been globally ordered. The replica then fetches the last delivered block from the set $\mathcal{P}$ of blocks delivered by SB instances, denoted by the set $\mathcal{P}'$. It then finds the block b^* that has the lowest ordering index among the blocks in $\mathcal{P}'$, *i.e.*, $\forall b' \in \mathcal{P}'$ with $b' \neq b^* : b^* \prec b'$. Thereafter, bar can be computed as a tuple of $(rank, index)$:

$$bar := (rank, index) = (b^*.rank + 1, b^*.index).$$

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 and decides which blocks to confirm. Specifically, the replica finds the block $b_{can} \in \mathcal{W}$ that has the lowest ordering index, which is referred to as the *candidate* block. If
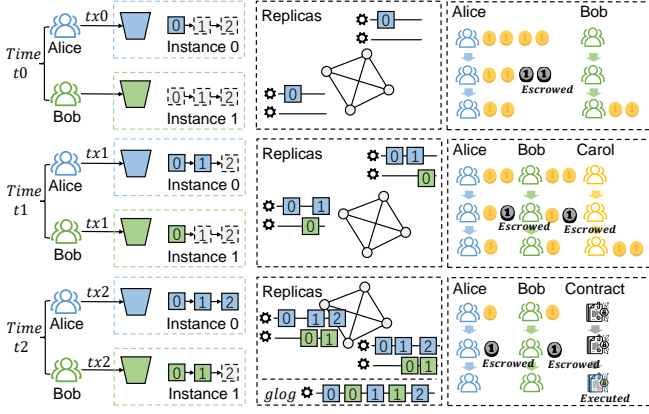
Figure 9: Protocol running example with two instances, three clients, and a contract. tx_i is a transaction arrived at time t_i.

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 appended to the $glog$ and removed from the set $\mathcal{W}$. The process repeats until no such b_{can} can be found.

APPENDIX B
PROTOCOL RUNNING EXAMPLE

To give readers a better understanding of how the protocol operates, we provide a running example of Orthrus, as shown in Fig. 9. Consider a system with two instances, Instance 0 and Instance 1, and three clients: Alice, Bob, and Carol, with initial balances of $4, $0, and $0, respectively. Assume Alice is assigned to Instance 0 and Bob is assigned to Instance 1. Additionally, there is a smart contract that requires two clients to invoke it together, incurring a cost of $1 per client.

Example of payment transactions. Consider a payment transaction tx_0 with a single payer, Alice, and a single payee, Bob, where Alice transfers $2 to Bob. The transaction involves two owned objects, Alice and Bob, and is assigned to Instance 0, as Alice performs a decremental operation of $2. Assume tx_0 is included in Block 0 of Instance 0. Since there is only one payer, tx_0 is executed immediately, escrowing $2 from Alice's account and transferring it directly to Bob's account.

Now consider another transaction tx_1, where both Alice and Bob each transfer $1 to a single payee, Carol. This transaction involves three owned objects: Alice, Bob, and Carol. Alice and Bob perform decremental operations of $1 each, while Carol performs an incremental operation of $2. As a result, tx_1 is assigned to both Instance 0 (for Alice) and Instance 1 (for Bob). Assume tx_1 is included in Block 1 of Instance 0 and Block 0 of Instance 1. Block 0 of Instance 1 must refer to state $S = \{0, \perp\}$, which represents Block 0 of Instance 0 (containing tx_0) and the initial state of Instance 1. This dependency ensures that Bob's transfer relies on the completion of tx_0, where Bob receives sufficient balance. In Block 1 of Instance 0, $1 is escrowed from Alice's account. Concurrently, in Block 0 of Instance 1, $1 is escrowed from

Bob's account. These operations can be processed in parallel because they belong to different instances and do not conflict. However, the escrow operation in Instance 1 must occur after tx_0, as indicated by the reference to Block 0 of Instance 0. Once both escrow operations are successfully processed, the replicas commit them, and $2 is transferred to Carol's account, completing the transaction tx_1.

Example of contract transactions. Consider a contract transaction, tx_2, where both Alice and Bob call the contract together. In this transaction, tx_2 involves two owned objects: Alice and Bob. Alice and Bob each perform a decremental operation of $1. Since tx_2 involves both Alice and Bob, which are owned objects with a decremental operation, it is assigned to Instance 0 and Instance 1.

Assume that tx_2 is included in Block 2 of Instance 0 and Block 1 of Instance 1. In Block 2 of Instance 0, $1 is escrowed from Alice's account, while in Block 1 of Instance 1, $1 is escrowed from Bob's account. These two operations can be executed concurrently because they are payment operations (*i.e.*, decremental operations on owned objects).

Once a replica observes that both escrow requests have been successfully processed in their respective instances, it can execute other operations in the contract after the previous 4 blocks in the $glog$ and commit both escrow operations. This ensures consistent execution results across different replicas, since the contract may contain shared objects that can not be confined to a certain instance.

APPENDIX C
OTHER RELATED WORKS

Partial ordering of payment systems. CryptoConcurrency [45] introduces dynamic detection of overspending based on the balance of the account, determining whether concurrent transactions can be satisfied without requiring consensus. Astro [21] maintains a log separately for each client to prevent double-spending. ABC [46] enables validators to parallelize transaction processing without requiring consensus, increasing system efficiency. FastPay [47] takes advantage of payment semantics to minimize shared state between accounts, allowing asynchronous operations to run with greater concurrency. Flash [48] sidesteps the need for reliable broadcast by utilizing a partially-ordered, DAG-like structure. A non-sequential specification for money transfer objects is presented in [49], which operates on a reliable broadcast abstraction. Pastro [50] introduces a partially ordered transaction set that defines active participants and stake distribution, offering an adaptable approach for applications that do not require a global order. These approaches focus on payments but lack support for smart contracts.

Other scaling approaches. Similar to Multi-BFT consensus, DAG and sharding systems also utilize instance parallelism to enhance system scalability. Notable DAG protocols include DAG-Rider [51], Narwhal and Tusk [52], Bullshark [53], Shoal [54], BBCA [55] and Mysticeti [56] require each block to reference at least $2f + 1$ predecessor blocks. BFT sharding

protocols [57]–[61] such as Elastico [62], OmniLedger [63], and RapidChain [64], Sharper [65], Byshard [60], divide replicas into multiple subcommittees for parallel processing. Except for the above approaches, some systems also employ concurrent execution of transactions to improve scalability [66]–[69].