Rashnu: Data-Dependent Order-Fairness

Mohammad Javad Amiri, Heena Nagda, Shubhendra Pal Singhal, Boon Thau Loo*
University of Pennsylvania
{mjamiri,hnagda,Shubhpal,boonloo}@seas.upenn.edu

Abstract

Distributed data management systems use state Machine Replication (SMR) to provide fault tolerance. The SMR algorithm enables Byzantine fault-tolerant (BFT) protocols to guarantee safety and liveness despite the malicious failure of nodes. However, SMR does not prevent the adversarial manipulation of the order of the transactions, where the order assigned by a malicious leader differs from the order in that transactions are received from clients. While *order-fairness* has been recently studied in a few protocols, such protocols rely on synchronized clocks, suffer from liveness issues, or incur significant performance overhead. This paper presents *Rashnu*, a high-performance fair ordering protocol. Rashnu is motivated by the fact that fair ordering among two transactions is needed when both transactions access a shared resource. Based on this observation, we define the notion of *data-dependent order fairness* where replicas capture only the order of data-dependent transactions and the leader uses these orders to propose a dependency graph that represents fair ordering among transactions. Replicas then execute transactions using the dependency graph resulting in the parallel execution of independent transactions. We implemented a prototype of Rashnu on top of HotStuff, where our experimental evaluation reveals the efficiency of Rashnu compared to the state-of-the-art order-fairness protocol and its low overhead compared to HotStuff.

1 Introduction

Distributed data management systems rely on consensus protocols to provide robustness and high availability [12, 16, 19, 28, 33, 49, 70]. Consensus protocols use the State Machine Replication (SMR) algorithm [61, 76] to ensure that all honest replicas execute transactions in the same order (safety), and all correct transactions are eventually executed (liveness). Existing consensus protocols typically include a designated leader replica that receives transactions from clients, assigns an order (e.g., a sequence number) to each transaction, which represents the position of the transaction in the final log, and initiates agreement on the order of transaction among all replicas. A malicious leader, however, can control transactions' inclusion and final ordering without violating safety or liveness. Existing BFT protocols do not prevent the adversarial manipulation of the actual ordering of transactions.

Adversaries manipulation of transactions order has been studied in the domain of decentralized finance (DeFi) [13, 31, 37, 47, 56, 74, 85] where miners can make profit by including, excluding, or reordering transactions within blocks, known as blockchain extractable value (BEV) [31]. As an example, consider an exchange transaction to buy a particular asset. A malicious leader can perform a front-running sandwich attack by placing the transaction between two buy and sell transactions to manipulate asset prices. The attacker buys assets for a lower price to let the victim buys at a higher value and then sells them again, typically at a higher price afterwards. Such adversarial manipulations of transactions in the Ethereum network resulted in extracting more than USD 6M in revenue from unsophisticated users [31]. Other than profitability, the execution order of transactions might determine the validity of a transaction typically when multiple transactions access limited assets, e.g., two money transfers from the same account when the account balance is not sufficient to perform both, or a ticket booking scenario where the number of available tickets is limited.

Different techniques, such as censorship resistance, random leader election, and threshold encryption, have been proposed to prevent the manipulation of transaction ordering. Censorship resistance [69] only ensures that correct transactions are eventually ordered, i.e., not censored. However, reordering transactions, e.g., sandwich attacks, is still possible. Similarly, reputation-based systems [9, 30, 58, 63] only detect unfair censorship. The random leader (committee) election or in general participation equity, on the other hand, provides opportunities for every replica to propose and commit its transactions, e.g., by becoming a proposer [4, 9, 32, 43, 44, 50, 54, 63, 67, 73, 80]. However, a malicious proposer can still order transactions unfairly in its turn. Finally, using threshold encryption [9, 21, 69, 79], transactions are encrypted, and their content is revealed once their order is fixed. This technique suffers from metadata leakage as well as collusion attacks between clients and the leader, where the leader becomes aware of a client transaction before ordering it and manipulates its order [52, 53, 59].

Recently, the notion of *order-fairness* is presented to address the manipulation of transaction ordering [22, 52, 53, 59, 60, 84]. Intuitively, order-fairness ensures that if a sufficiently large number of replicas (known as γ-fraction) receive a transaction t_1 before another transaction t_2, then t_1 must be ordered before t_2 [53]. To support order-fairness, clients broadcast their transactions to all replicas. In each round, every replica locally orders transactions it has received according to their received times and sends it to the to the designated leader of that particular round. The leader then constructs a fair-ordered proposal from the received blocks and initiates consensus on the transaction block.

While the notion of order-fairness has been presented in several recent studies, i.e., Wendy [59, 60], Pompe [84], Aequitas [53], Themis [52] and Quick-order-fairness [22], these protocols suffer from serious limitations. First, both Wendy [59, 60] and Pompe [84] rely on synchronized clocks between replicas, making these protocols impractical in asynchronous networks. Pompe [84] also determines fair order using the median of timestamps assigned by replicas, which faulty replicas can easily manipulate. Second, Aequitas [53], and Quick-order-fairness [22] only guarantee weak liveness and transactions might need to wait an arbitrarily long time before getting committed [52]. Finally, Themis [52], which bootstraps from HotStuff [82], is the only fair ordering protocol implementation with no synchronized clocks assumption. Themis,

*The first and the second authors contributed equally.

however, suffers from significant performance overhead, making it impractical in real-world applications.

One main reason behind the poor performance of order-fairness protocols is the time required to generate a fair order among *all* received transactions. In an asynchronous network, different replicas receive transactions in different orders, and transactions might be arbitrarily delayed. Byzantine replicas might also send maliciously manipulated local ordering to the leader. Moreover, collecting the local ordering of different replicas might lead to cycles in the final order even when all replicas are honest, as demonstrated by the Condorcet paradox. Therefore, it becomes time-consuming for the leader to check the order of each pair of transactions in every local ordering and achieve a fair order among all transactions.

Fair ordering of transactions is essential when transactions access the same data objects and manipulating the order gives an unfair advantage to some transactions. However, the execution order of transactions with no data dependency does not impact the execution results. This is crucial, especially since the contention level of workloads in many practical applications is typically low. Based on this observation, a key insight for our work is that while order-fairness ensures fair ordering among all transactions, a more practical notion of order-fairness can limit the fair ordering only to data-dependent transactions.

In this paper, we present the notion of *data-dependent order-fairness* to ensure that if a large number of replicas receive data-dependent transactions in a particular order, the order is preserved in the execution. Using the notion of data-dependent order-fairness, we present a high-performance fair ordering protocol, *Rashnu*[1].

In Rashnu, each replica, instead of simply collecting a list of received transactions, constructs a dependency graph in the form of a directed acyclic graph (DAG). The dependency graph captures data dependencies between transactions according to their received order. Upon finishing a round (specified by a threshold, e.g., time window, number of transactions, or both), the replica sends the dependency graph to the leader. The leader then collects the local ordering of different replicas and constructs a fair-ordered proposal from the local order of transactions in received blocks.

To extract a fair-order, the leader needs to consider all data dependencies between transactions in all local dependency graphs. This is challenging because replicas might receive transactions in different orders or even receive different sets of transactions, e.g., due to the asynchronous nature of the network. Once the fair order is extracted, the leader initiates consensus on the transaction block by sending a proposal. The proposal includes transactions, their fair order, and a proof, e.g., all received blocks, to show that the proposed order is fair. Since the fair ordering of transactions is performed before the initiation of consensus, Rashnu can bootstrap from any leader-based protocol.

Proposing data-dependent order-fairness, Rashnu, on one hand, reduces the leader latency by capturing the order of only data-dependent transactions and, on the other hand, enables replicas to execute data-independent transactions in parallel. Rashnu further resolves Condorcet cycles and chained Condorcet cycles using the batch-order-fairness [53] and the deferred ordering [52] techniques, respectively.

[1]Rashnu is the Avestan language name of the Zoroastrian deity of justice.

Rashnu can be implemented on top of any leader-based consensus protocol. To be able to compare with the state-of-the-art order-fairness protocol, i.e., Themis [52], we implemented Rashnu on HotStuff [82] (used by Themis) as the underlying BFT protocol. The code will be publicly available to be used in future research.

Overall, this paper makes three main contributions.

- The notion of data-dependent order-fairness is defined as providing fair ordering only among data-dependent transactions.
- We design Rashnu, a high-performance fair-ordering protocol that decouples ordering from consensus and leverages graph-based techniques to achieve order-fairness among data-dependent transactions.
- We implement a prototype of Rashnu bootstrapped from HotStuff [82]. Our evaluation results demonstrate the ability of Rashnu to provide significant throughput improvement compared to Themis.

2 Background

A Byzantine fault-tolerant (BFT) protocol is a key component of distributed data management systems with non-trustworthy infrastructures such as permissioned blockchains [1, 2, 8, 25, 77], permissionless blockchains [20, 57, 58, 64, 83], distributed file systems [5, 23, 26], locking service [27], firewalls [15, 41, 42, 75, 78, 81], certificate authority systems [87], SCADA systems [10, 55, 71, 86], key-value datastores [14, 34, 45, 48, 75], and key management [66]. A BFT protocol runs on a network consisting of a set of replicas that may exhibit arbitrary, potentially malicious, behavior. BFT protocols use the State Machine Replication (SMR) algorithm [61, 76] to ensure that honest replicas execute client requests in the same order despite the concurrent failure of f Byzantine replicas. SMR BFT protocols must provide safety and liveness.

A recent line of work, e.g., Wendy [59, 60], Aequitas [53], Pompe [84], Themis [52] and Quick order-fairness [22] have proposed to add *order-fairness* as the third property that SMR BFT protocols need to guarantee. Order-fairness properly aims to ensure that the transactions are committed in the same order as they arrived at the network. The order-fairness property is parameterized by an order-fairness parameter γ representing the fraction of replicas that receive transactions in a particular order. Wendy [59, 60] and Pompe [84] requires replicas to access synchronized local clocks. Pompe further determines the fair order by relying on timestamps assigned by replicas, which can be manipulated by malicious replicas. As a result, we mainly focus on Aequitas [53], Themis [52] and Quick order-fairness [22], which do not consider synchrony assumptions, making them more suitable for asynchronous networks.

Validity requirement. While order-fairness is proposed as a new property for SMR BFT protocols, it has a strong connection to the validity requirement of Byzantine agreement [62]. Specifically, validity states that if all honest replicas propose the same value v, then replicas must agree on value v. In the context of fair ordering, value v can be interpreted as the order of two different transactions t_1 and t_2. Hence, the validity property can be redefined as if all honest replicas propose transaction t_1 before transaction t_2, then t_1 must be ordered before t_2 in the final order. This validity notion, however, requires all honest replicas to propose the same order, leaving the decision value open even if one correct replica proposes

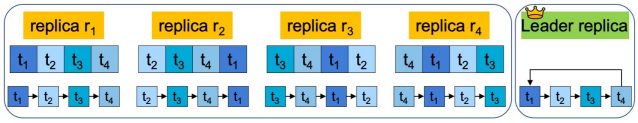


Figure 1: A Condorcet cycle

something different. The *differential validity* notion addresses this point by replacing "all" with "sufficiently many" in the validity definition [40].

Receive-Order-fairness. In a fair-ordering protocol, different replicas need to individually order transactions locally and send their local order to the leader. Each replica can order transactions based on (1) the timestamp assigned by clients, (2) the propagation time (that can be estimated by measuring network latency), and (3) the received time. Since clients might maliciously assign timestamps to their transactions, replicas cannot rely on the assigned timestamps (unless each client is equipped with trusted hardware). The propagation time also cannot be captured precisely as the network is asynchronous and transactions might be arbitrarily delayed. As a result, existing protocols [22, 52, 53] rely on the transactions receive time and use the notion of *receive-order-fairness*. The notion of (strong) receive-order-fairness specifies that if γ fraction of replicas receive a transaction t_1 before another transaction t_2, then all honest replicas must order t_1 (strictly) before t_2.

Condorcet cycles. In an asynchronous network, replicas might receive transactions in different orders. Hence, as demonstrated by the *Condorcet paradox*, defining a fair order among all transactions becomes impossible, even if all replicas are honest. Condorcet paradox states that even if the local order of all individual replicas is transitive, there might be situations that lead to non-transitive collective voting preferences. Figure 1 demonstrates the Condorcet paradox with four transactions between four replicas. As can be seen, a majority (3 out of 4) of replicas received t_1 before t_2 (replicas r_1, r_3 and r_4). Similarly t_2 is received before t_3 by a majority of replicas (r_1, r_2 and r_4) and t_3 is received before t_4 by replicas r_1, r_2 and r_3. However, t_4 is also received by replicas r_2, r_3 and r_4 before t_1. The collection of these ordering results in a cyclic global ordering as generated by the leader (the leader is one of the replicas).

To deal with Condorcet cycles, batch-order-fairness is presented [53] where transactions involved in a Condorcet cycle are delivered concurrently within the same batch. Specifically, given two transactions t_1 and t_2 where t_1 has been received by sufficiently many replicas before t_2; while strong order-fairness requires the leader to send t_1 to replicas before t_2, batch-order-fairness relaxes this requirement by saying t_1 should be delivered no later than (before or at the same time as) t_2. Batch-order-fairness does not specify the order among the transactions within a batch and respects a fair order up to this limit. However, many applications still require a total order among all transactions for transaction execution. To do so, a deterministic total ordering for transactions in the same cycle, e.g., alphabetical [53], can be used.

Weak liveness. Batch order-fairness circumvents the Condorcet impossibility results. However, it achieves only weak liveness. This is because Condorcet cycles, as noted by Aequitas [53] and shown by Themis [52], might chain together, and extend arbitrarily. In this case, using batch order-fairness, the leader waits for a chain to be completed before sending the batch to replicas. Hence, liveness might be violated as transactions need to wait for a long time.

The weak liveness issue of batch order-fairness can be addressed by delivering transactions within the same cycle contiguously in a set of successive blocks (instead of a single block) [52]. Using this technique, a part of the current cycle can be output later without violating order-fairness as long as no transaction from a later cycle comes before it. This *deferred ordering* technique [52] enables the leader to propose a partial (incomplete) ordering for some transactions within the block and defer their total ordering to the next consecutive blocks. The total ordering for deferred transactions does not depend on the chaining of Condorcet cycles. As a result, (standard) liveness can be achieved.

3 Rashnu Model

Rashnu deploys on a set of n known nodes (replicas) where at most f of them might be Byzantine (malicious). In the Byzantine failure model, faulty replicas may exhibit arbitrary, potentially malicious, behavior. In an asynchronous system, where replicas can fail, no consensus solutions guarantee both safety and liveness (FLP result) [39]. As a result, Rashnu assumes the partially synchronous communication model to circumvent the FLP impossibility. In a partial synchrony model, an unknown global stabilization time (GST) exists, after which messages between honest replicas are received within some unknown bound Δ. A strong adversary can coordinate malicious replicas and delay communication to compromise service. However, the adversary is computationally bounded and cannot subvert standard cryptographic assumptions. Replicas are connected with point-to-point bi-directional communication channels, and each client can communicate with any replica. Network links are pairwise authenticated, which guarantees that a malicious replica cannot forge a message from an honest replica. For communication between replicas, we assume the presence of standard digital signatures and public-key infrastructure (PKI). A collision-resistant hash function $D(.)$ is also used to map a message m to a constant-sized digest $D(m)$.

Byzantine fault-tolerant protocols, e.g., PBFT [24], require $3f + 1$ replicas to guarantee safety with f malicious replicas [17, 18, 29, 36, 62]. However, fair ordering of transactions requires the number of replicas n to be larger. The order-fairness is further parameterized by an order-fairness parameter γ representing the fraction of replicas that receive transactions in a particular order. Rashnu, similar to other (partially synchronous) order-fairness protocols [52, 53], requires$n$ to be larger than $\frac{4f}{2\gamma-1}$.

Lemma 3.1. Given a network consisting of n replicas from which at most f are malicious. The fair ordering of transactions is possible only when $n > \frac{4f}{2\gamma-1}$ where γ is the fraction of replicas that receive transactions in a particular order.

Proof. **(Using quorum size)** In a network consisting of n replicas, since f replicas might be faulty, the protocol can rely on a quorum of $n - f$ different replicas to generate the final order. These $n - f$ replicas, however, might include f malicious replicas, i.e., f replicas not participating are slow honest replicas. As a result, only $n - 2f$ replicas within the quorum are guaranteed to be honest. To realize order-fairness if γn replicas receive transactions in a particular order, the final ordering must reflect that order. Since only $n-2f$

replicas are guaranteed to be honest, the output of order-fairness must be the same as γn even with $\gamma n - 2f$ replicas broadcasting a particular order. On the other hand, to ensure that only one of the two different orders $t_1 \prec t_2$ or $t_2 \prec t_1$ is captured between two transactions t_1 and t_2, a majority of replicas must agree with the order, i.e., $\gamma n - 2f > \frac{n}{2}$. As a result $n > \frac{4f}{2\gamma-1}$. □

Proof. **(Using δ-differential validity)** The δ-differential validity [40] can also be used to prove the number of required replicas in a fair ordering protocol. Let $c(v)$ denote the number of honest replicas that propose value v. δ-differential validity states that if an honest replica decides v, then every other value v' proposed by another honest replica satisfies $c(v') \leq c(v) - \delta$. Based on this definition, a BFT protocol satisfies δ-differential validity if and only if it never decides a value v' with $c(v') < c(v) - \delta$ where v is the value proposed most often by honest replicas. In an asynchronous network, δ-differential consensus is achievable only if $\delta \geq 2f$ (i.e., $c(v') < c(v) - 2f$) [40].

In the context of fair ordering protocols, the value v is interpreted as the order of two different transactions t_1 and t_2. As stated before, the output of fair ordering must be the same even if $\gamma n - 2f$ replicas broadcast the order. As a result, $(1-\gamma)n < (\gamma n - 2f) - 2f$ where $(1-\gamma)n$ is the maximum number of replicas that might propose another order. Hence, $n > \frac{4f}{2\gamma-1}$. □

The order-fairness parameter γ represents the fraction of replicas that receive transactions in a particular order. The range of possible values for the order-fairness parameter γ can be calculated based on the total number of replicas n.

Lemma 3.2. Order-fairness parameter γ is between $\frac{1}{2}+\frac{2f}{n}$ and 1.

Proof. In Rashnu, Since $n > \frac{4f}{2\gamma-1}$, $\gamma \geq \frac{1}{2} + \frac{2f}{n}$. On the other hand, $\gamma = 1$ is the case where all replicas receive transactions in the same order. As a result, $\frac{1}{2} + \frac{2f}{n} < \gamma \leq 1$. □

In an asynchronous network, replicas might receive transactions in different orders. Even if all replicas are honest, defining a fair order among all transactions is impossible, as demonstrated by the Condorcet paradox. Condorcet paradox states that even if the local order of all individual replicas is transitive, there might be situations that lead to non-transitive collective voting preferences.

In an asynchronous network, as explained in Section 2, defining a fair order among all transactions might be impossible due to the presence of Condorcet cycles. Rashnu leverages the batching technique [53] to deal with Condorcet cycles where all transactions involved in a Condorcet cycle are batched and delivered at the same time. However, batch-order-fairness might lead to weak liveness issues where Condorcet cycles are chained together. To address the weak liveness issue resulting from chained Condorcet cycles, Rashnu uses the order deferring technique [52] where transactions of the same cycle are proposed contiguously in successive blocks.

The order of executing transactions matters when transactions compete with each other on the same resources and manipulating the order gives an unfair advantage to some transactions. As a result, our notion of order-fairness limits the fair ordering to data-dependent transactions. Each transaction performs a sequence of reads and writes, each accessing a single record. Rashnu assumes a priori knowledge of transactions' read- and write-set, where the read-set and write-set of transactions are pre-declared or can be obtained from the transactions via static analysis, e.g., all records involved in a transaction, are accessed by their primary keys. Note that even if that assumption does not hold, the system can employ speculative execution techniques [38] to obtain the read-set and write-set of each transaction. Given a transaction t, we use $R(t)$ and $W(t)$ to denote the read-set and write-set of transaction t respectively. Intuitively, two transactions t_1 and t_2 are data-dependent if they access the same data object and one performs a write operation on the data object. More precisely, two transactions t_1 and t_2 are *data-dependent* if $(R(t_1) \cap W(t_2)) \cup (W(t_1) \cap R(t_2)) \cup (W(t_1) \cap W(t_2)) \neq \emptyset$.

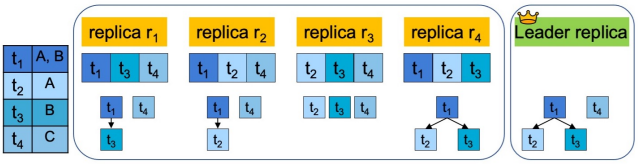


Figure 2: The local views of different replicas

Definition: (Data-dependent order-fairness). Given two data-dependent transactions t_1 and t_2. If γ-fraction of replicas receive t_1 before t_2, no honest replica outputs t_2 before t_1.

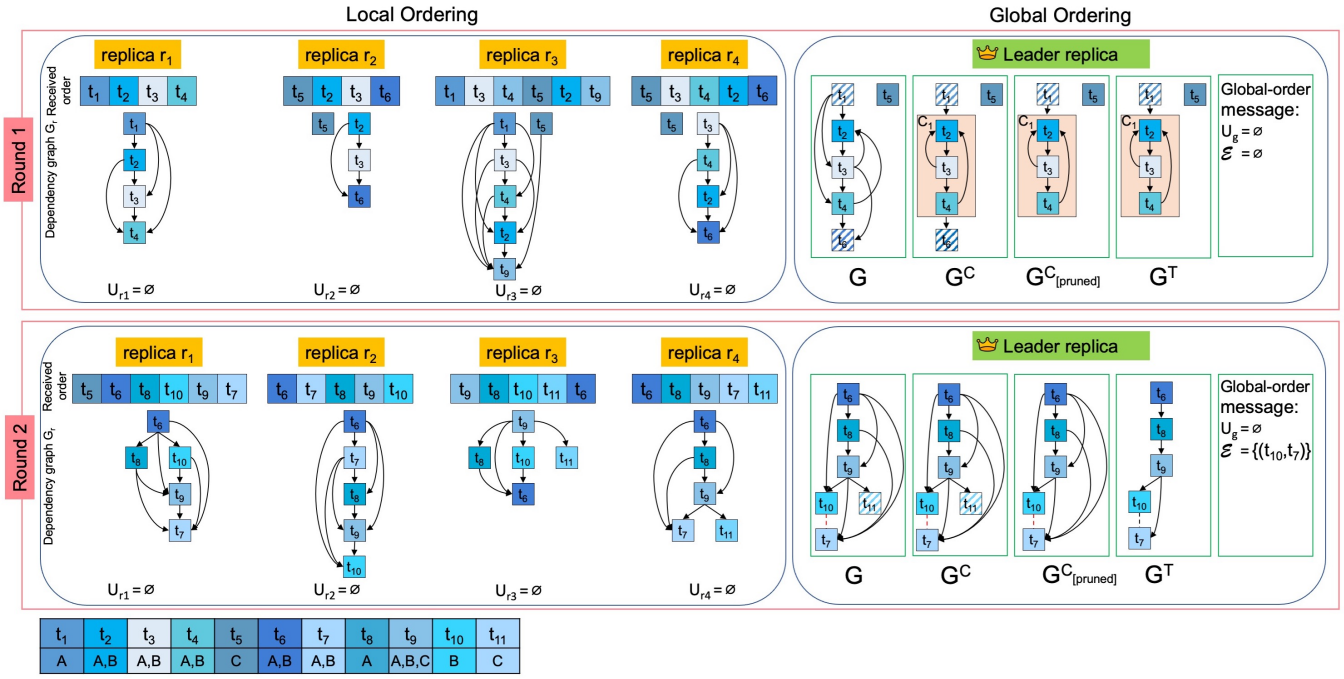
Rashnu, similar to all other fair ordering protocols [22, 52, 53, 59, 60, 84], processes transactions in rounds where each replica collects transactions received from clients and sends a block of transactions to the leader at the end of each round. Note that replicas might not receive the same set of transactions in each round due to the asynchronous nature of the network.

Figure 2 presents a simple example with a quorum of size 4 (assuming $f = 1$, n is $4f + 1 = 5$ and the quorum size $n - f = 4$). In a round, four transactions t_1 to t_4 are sent by clients where replicas receive different subsets of the transactions in various orders due to the asynchronous nature of the network. For example, replica r_3 receives t_2, t_3 and t_4 and since there is no data dependency between these three transactions, the dependency graph has no edges. However, replica r_4 receives t_1, t_2 and t_3 (in this order: $t_1 \prec t_2 \prec t_3$) where t_2 and t_3 write on A and B, both are written by t_1.

4 Fair Transaction Ordering

Rashnu is a fair ordering protocol that decouples ordering from consensus to ensure a fair order of client requests. In Rashnu, clients broadcast their transactions to all replicas. Each replica collects a batch of transactions, constructs a local dependency graph for transactions based on their received order, and sends the dependency graph to the leader. The leader then collects all local dependency graphs and generates a global dependency graph that captures fair order among transactions. If the order of two data-dependent transactions can not be determined by the leader, e.g., they are received by an insufficient number of replicas, the leader defers the order to the next blocks.

Figure 3 presents an overview of Rashnu in a simple example. We use this example throughout this section. The example includes four replicas r_1 to r_4 (assuming $f = 1$, n becomes $4f + 1 = 5$ and there are $n - f = 4$ replicas in the quorum) where one of them is the leader and presents two consecutive rounds of the



t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
A	A,B	A,B	A,B	C	A,B	A,B	A	A,B,C	B	C

Figure 3: Local and global ordering in Rashnu

protocol. A set of transactions t_1 to t_{12} are received from clients in these two rounds where each transaction accesses a subset of data objects A, B, and C, as shown in the figure (for simplicity, all data accesses are assumed to be write). This example includes many data dependencies among transactions to capture different corner cases. in real-world scenarios, however, a high percentage of transactions are typically data-independent.

This section demonstrates how different replicas construct their local ordering and how the leader orders transactions in a fair yet efficient manner. We then discuss the execution of transactions and the correctness of Rashnu.

4.1 Local Ordering

In the local ordering phase, each replica generates a dependency graph for client requests that the replica has received and their order is not determined yet. In each round i, a replica deals with three types of transactions.

- First, old transactions that have been received by the replica in a round j ($j < i$); however, they have not been proposed by a leader (in any round j to $i-1$) yet. This happens when an insufficient number of replicas receive a transaction in a round. Hence, the leader does not propose it until more replicas receive it (probably, in a later round).
- Second, new transactions that are received by the replica in the current round i; however, the leader has already proposed them in an earlier round. This is because, due to the asynchronous nature of the network, transactions might be delayed, and a replica receives a transaction that has already been received by other replicas, and its order is possibly determined in an earlier round.
- Third, new transactions that are received by the replica in the current round i, and have not been proposed by a leader in any earlier round.

As shown in Algorithm 1, each replica r initiates an empty graph $G_r = (T_r, E_r)$ at the beginning of each round i. The replica first adds all its old transactions (first type) to graph G_r (lines 2-3). The replica adds an edge (t', t) to the graph G_r for each transaction t if the replica has received t' before t (i.e., $t' \prec t$) and t and t' are data-dependent (lines 4-6). transactions t and t' might have been received by the replica in different rounds. Note that if the underlying consensus protocol relies on a stable leader, e.g., PBFT [24], the leader can keep track of the first type of transactions. In this way, replicas do not need to wait for the previous round leader's proposal before sending their local order in the current round. Upon receiving a valid signed request message $m = \langle \text{REQUEST}, t, \tau_c, c \rangle_{\sigma_c}$ from an authorized client c with timestamp τ_c to execute transaction t, the replica checks whether the transaction has already been proposed by the leader in an earlier round (second type). Otherwise (third type), the replica adds vertex t and all its dependencies with existing vertices to the graph G_r (lines 9-13).

If transaction t is already been proposed in an earlier round (second type), its order might not have been determined yet. When the leader finalizes the global order of transactions (as explained in Section 4.2), if the number of received orders between two data-dependent transactions t_1 and t_2 is insufficient, the leader can not determine the order and adds a pair (t_1, t_2) to a set of *missing pairs* (undirected edges) $\mathcal{E}$. This set is then used by replicas in the next round to specify the order of missing pairs and complete the previous proposals.

Algorithm 1 Local ordering on replica r

Input: (1) a set of incoming transactions in round i,
(2) the set of missing pairs (i.e., undirected edges) $\mathcal{E}$

Initiate an empty graph $G_r = (T_r, E_r)$
for every transaction t that is received in an earlier round but has not been proposed by a leader **do** ▷ First type
  Add transaction (vertex) t to T_r
  for every vertex $t' \in T_r$ **do**
    if t and t' are data-dependent **then**
      Add (t', t) to E_r
while round i has not been finished **do**
  for every incoming transaction t received from clients **do**
    if t is not proposed in an earlier round **then** ▷ Third type
      Add a vertex t to G_r
      for every vertex $t' \in T_r$ **do**
        if t' and t are data-dependent **then**
          Add (t', t) to E_r
    else if t is a vertex in an earlier proposal **then** ▷ Second type
      for every pair $(t', t) \in \mathcal{E}$ **do**
        if t' is already received **then**
          Add (t', t) to U_r
Send $\langle\langle$LOCAL-ORDER, $i, G_r, U_r\rangle_{\sigma_r}, T\rangle$ to the leader

When a replica receives a transaction t that was proposed in an earlier round, the replica checks the set of missing pairs (undirected edges) $\mathcal{E}$ sent by the leader of the previous round to see if t is part of any missing pairs. For any pair $(t', t) \in \mathcal{E}$, if the replica has already received t', the replica adds (t', t) to the set of *updated local ordering* U_r to specify the order between t' and t. Updated ordering includes a set of edges between transactions of proposals that have been received in previous rounds. Such ordering dependencies enable the leader to finalize previous proposals by adding the missing edges.

At the end of round i, each replica r sends a signed local-order message $\langle\langle$LOCAL-ORDER, $i, G_r, U_r\rangle_{\sigma_r}, T\rangle$ including the dependency graph G_r and the set of updated local ordering U_r to the leader. The set of received requests T is piggybacked (not included) to keep local-order messages small. This is important because local-order messages are used by the leader in the global ordering phase to prove that this set of requests has been received.

Figure 3 presents the local ordering phase of Rashnu on different replicas in two rounds. Data objects accessed (written) by each transaction are shown in the bottom of the figure. For example, in round 1, replica r_2 receives transactions in this order: $t_1 \prec t_5 \prec t_2 \prec t_3 \prec t_6$ and generates local dependency graph G_{r_2} by adding edges between data-dependent transactions, e.g., (t_1, t_2) as both write on data object A. As shown, replicas receive different sets of transactions in each round and need to deal with different cases. For instance, replica r_3 receives transaction t_9 in round 1. However, since it has not been proposed by the leader, replica r_3 includes t_9 in its graph of round 2 (first type). Similarly, while replica r_1 receives transaction t_5 in round 2, the replica does not send t_5 to the leader because it has already been proposed by the leader of round 1 (second type). The set of updated local ordering U_r is empty in both rounds for all replicas. This is because in round 1, there is no prior proposal to be updated and the proposal of round 1 has no missing edges ($\mathcal{E} = \emptyset$) to be updated in round 2.

4.2 Global Ordering

In the global ordering phase, the leader replica receives the local-order messages from different replicas and generates the final ordering of transactions. Since f replicas might be faulty and not send

Algorithm 2 Global ordering on leader replica π

Input: (1) $n - f$ local-order messages received from different replicas,
(2) list of missing edges $\mathcal{E}$

for every transaction t in some local-order messages **do**
  if t appears in at least $n - 2f$ local-order messages **then**
    Label t as fixed
  else if t appears in at least $n(1-\gamma) + f + 1$ local-order messages **then**
    Label t as pending
▷ Step 1: global dependency graph generation
Initiate an empty graph $G = (V, E)$
Initiate updated global ordering dependency list $L_g = \emptyset$
for every fixed or pending transaction t **do**
  add a vertex t to G
for each pair of vertices t_1 and t_2 **do**
  if $w(t_1, t_2) > w(t_2, t_1)$ & $w(t_1, t_2) \geq n(1-\gamma) + f + 1$ **then**
    Add (t_1, t_2) to E
  else if $w(t_2, t_1) > w(t_1, t_2)$ & $w(t_2, t_1) \geq n(1-\gamma) + f + 1$ **then**
    Add (t_2, t_1) to E
▷ Step 2: condensation graph generation
$G^C = (V^C, E^C) \leftarrow$ CONDENSATION(G)
▷ Step 3: graph pruning
for every pending vertex u **do**
  if there is no fixed vertex v such that $(u, v) \in E^C$ **then**
    Remove u from V^C
    Remove all edges involving u from E^C
for every pair of data-dependent transactions t_1 and t_2 with no edges **do**
  if t_1 and t_2 are in two different vertices of V^C **then**
    Add (t_1, t_2) to $\mathcal{E}$
▷ Step 4: DAG transitive reduction generation
$G^T = (V^T, E^T) \leftarrow$ TRANSITIVEREDUCTION(G^C)
▷ Step 5: updating previous proposals
for each pair of transactions t_1 and t_2 in any U_i **do**
  if $w(t_1, t_2) > w(t_2, t_1)$ & $w(t_1, t_2) > n(1-\gamma) + f + 1$ **then**
    Add (t_1, t_2) to U_g
  else if $w(t_2, t_1) > w(t_1, t_2)$ & $w(t_2, t_1) > n(1-\gamma) + f + 1$ **then**
    Add (t_2, t_1) to U_g
▷ Step 6: global order proposal
Send $\langle$GLOBAL-ORDER, $G^T, \mathcal{E}, \mathcal{L}, U_g, \mathcal{L}_u\rangle_{\sigma_\pi}$ to all replicas

their local-order messages, the leader collects a quorum of $n - f$ local-order messages from different replicas (including itself) to generate the global order. Algorithm 2 demonstrates global ordering on the leader.

We define two types of transactions: fixed and pending. Transaction t is fixed if it appears in at least $n - 2f$ local-order messages, whereas t is pending if at least $n(1-\gamma) + f + 1$ (and less than $n - 2f$) local-order messages include t. A fixed transaction has been received by sufficiently many replicas to be included in the final order. Specifically, since at most f local-order messages in the quorum might have been received from faulty replicas, the leader counts on only $n - 2f$ messages and if $n - 2f$ different replicas receive a transaction, the transaction is ordered safely. On the other hand, pending transactions are not received by enough replicas yet to finalize an order. However, an edge from a pending to a fixed transaction might occur. As a result, we keep such pending transactions in the leader proposal enabling the leader to propose more transactions. We show that this does not violate the order-fairness (Definition 3). In Figure 3, since $\gamma = 1$, $f = 1$ (and $n = 5$), a transaction t is fixed if it appears in 3 or 4 local orderings, e.g., transactions t_2, t_3, and t_4 in round 1, while transaction t is pending if only 2 local orderings include t, e.g., transactions t_1 and t_6 in round 1.

We also define a weight function $w : E \mapsto [0, n - f]$ to denote the number of local dependency graphs that include a particular edge. Given a set $\mathcal{L}$ of $n - f$ local dependency graphs and two transactions (vertices) t_1 and t_2 in round i, $w(t_1, t_2)$ represents the

number of graphs that include an edge from t_1 to t_2. Note that for any pair of data-independent transactions t_1 and t_2, $w(t_1, t_2) = 0$. In Figure 3 and in round 1, $w(t_3, t_4) = 3$ while $w(t_3, t_9) = 1$.

Step 1: global dependency graph generation. Upon receiving a quorum of $n - f$ local-order messages from different replicas in round i, the leader initiates an empty graph $G = (V, E)$ and adds all fixed and pending transactions to its vertex set, as presented in Algorithm 2, lines 6-9. For each pair of data-dependent transactions t_1 and t_2 in V, the leader calculates both $w(t_1, t_2)$ and $w(t_2, t_1)$. If the maximum of $w(t_1, t_2)$ and $w(t_2, t_1)$ is equal or greater than $n(1 - \gamma) + f + 1$, then the corresponding edge will be added to the graph G (lines 10-14).

The goal of this step is to include as many transactions as possible while making sure that the exclusion of any transaction from the proposal does not violate fairness (Definition 3). In particular, the order-fairness definition states that if γ fraction of replicas receives transaction t_1 before transaction t_2, then t_2 is not ordered before t_1. When at least $n(1 - \gamma) + f + 1$ replicas propose a particular order $t_1 \prec t_2$ for two transactions t_1 and t_2, at most, $n\gamma - 2f - 1$ replicas (from the set of $n - f$ replicas) can propose the reverse order $t_2 \prec t_1$, i.e., $(n - f) - (n(1 - \gamma) + f + 1) = n\gamma - 2f - 1$. This number of replicas is, nevertheless, still lower than the γ fraction of (honest) replicas (as discussed in lemma 3.1, only $n - 2f$ replicas within the quorum of $n - f$ replicas are guaranteed to be honest). As a result, if $n(1 - \gamma) + f + 1$ replicas propose an order $t_1 \prec t_2$, the order can be safely chosen and the order-fairness will not be violated even if all remaining replicas propose the reverse order $t_2 \prec t_1$.

Specifically, when $\gamma = 1$, based on the order-fairness definition, if all honest replicas (i.e., at least $n - 2f$ within a quorum of $n - f$) receive transaction t_1 before t_2, then t_2 should not be ordered before t_1. As a result, if the protocol observes that $f + 1$ replicas have received transaction t_1 before transaction t_2, it can safely order t_1 before t_2. This is because it becomes impossible for $n - 2f$ honest replicas within a quorum of $n - f$ to receive t_1 and t_2 in the reverse order (i.e., $t_2 \prec t_1$), hence, order-fairness can not be violated. Moreover, if neither of two possible orders satisfies order-fairness, i.e., some ($> f$) received $t_1 \prec t_2$ while others ($> f$) received $t_2 \prec t_1$, the protocol is free to choose one of the orders. As a result, deciding based on $f + 1$ local ordering does not violate order-fairness. In this case, the protocol chooses the order with the higher weight.

Continuing with the example of Figure 3, in round 1 the leader adds fixed transactions t_2, t_3, t_4, t_5 and pending transactions t_1 and t_6 to the graph while transaction t_9 is not added as it is received by only one replica. The leader adds an edge between two transactions if the edge appears in at least $n(1 - \gamma) + f + 1 = 2$ local graphs, e.g., edge (t_2, t_4) is not added because it appears only in G_{r_1}.

Note that malicious replicas might add invalid edges, e.g., between independent transactions, to their graph. However, the leader can detect such edges. Even if the leader does not validate edges, since at most f replicas are Byzantine, no invalid edges will be added to the final graph. Similarly, if the leader is malicious and adds incorrect edges, its malicious behavior can be easily detected by replicas as the leader must send the $n - f$ local ordering the replicas.

Step 2: condensation graph generation. The graph generated by the leader, as demonstrated by the Condorcet paradox, might contain cycles. Rashnu deals with cycles using the batch ordering technique [52, 53] where transactions involved in a cycle are delivered to replicas simultaneously and within a batch. To determine the order of transactions, the final graph must be acyclic. To generate an acyclic graph from a cyclic graph, Rashnu uses the graph condensation technique. Given a graph G, to generate the condensation graph G^C of G, Rashnu first identifies the strongly connected components of G. Each strongly connected component intuitively represents either a single vertex (transaction) or a cycle in graph G. More formally, a strongly connected component C is a maximal subset of vertices such that any two vertices of this subset are reachable from each other. The condensation of graph G is graph $G^C = (V^C, E^C)$ where each vertex $C \in V^C$ corresponds to a strongly connected component of graph G, and there is an edge $(C_i, C_j) \in E^C$ if and only if there are two vertices $u \in C_i$ and $v \in C_j$ such that $(u, v) \in E$. A vertex C in G^C is fixed if it includes at least one fixed transaction. Otherwise, C is pending.

As shown in Figure 3, transactions t_2, t_3 and t_4 construct a cycle in graph G of round 1. As a result, the condensation graph G^C of G consists of four vertices (strongly connected components): single-transaction vertices t_1, t_5, and t_6 and a vertex C_1 consisting of t_2, t_3 and t_4. The resulting graph G^C is acyclic.

Step 3: graph pruning. Once the condensation graph is generated, the next step is to remove pending transactions that have no outgoing path to a fixed transaction. These transactions are removed because they have not been received by a sufficient number of replicas and they do not incorporate in determining the order of a fixed transaction, i.e., we initially add them to the graph because there might be a path from a pending transaction to a fixed transaction, helping in determining the order. Given two data-dependent transactions t_1 and t_2 where t_1 is fixed, t_2 is pending, and $(t_1, t_2) \in E^C$. Since at least $n(1 - \gamma) + 1$ honest replicas have received t_1 before t_2, removing pending transaction t_2 does not violate fairness.

In Figure 3, transactions t_1 and t_6 are pending in round 1. However, since t_1 has outgoing paths to fixed vertex C_1, we keep it in the graph; while t_6 is removed from the graph. Removing t_6 enables the next proposer to propose the order of t_6 freely when t_6 appears in a sufficient number of local graphs.

Finally, for every pair of data-dependent transactions t_1 and t_2 with no edges in between, if t_1 and t_2 are not in the same vertex of G^C (i.e., they are not part of the same cycle), the leader adds a pair (t_1, t_2) to the set of missing edges $\mathcal{E}$. Maintaining missing edges is necessary because determining the order of a missing edge might result in a new or extended cycle. Hence, a transaction should not be executed until the order of all its predecessor transactions in the graph is determined. We do not maintain missing edges between transactions involved in a cycle, because such edges do not contribute in ordering transactions (i.e., the involving transactions already constructed a cycle). In Figure 3, while round 1 has no missing edges, the edge between t_7 and t_{10} is missing in round 2.

Step 4: DAG transitive reduction generation. The generated graph includes all determined orders between different transactions. As an optimization, the graph can be simplified by removing the transitive edges. The transitive reduction of a directed graph is another directed graph that has the same reachability relation with

Algorithm 3 Order finalization on replica r

Input: a global-order message received from the leader

▷ Step 1: Global order validation

1: Upon receiving $\langle$GLOBAL-ORDER, G^T, $\mathcal{E}$, $\mathcal{L}$, U_g, $\mathcal{L}_u\rangle_{\sigma_\pi}$
2: Validate G^T, $\mathcal{E}$ and U_g using $\mathcal{L}$ and $\mathcal{L}_u$

▷ Step 2: Establishing consensus

3: HotStuff(GLOBAL-ORDER)

▷ Step 3: Transaction execution

4: **for** every edge (t_1, t_2) in U_g where $t_1, t_2 \in$ Block B_i **do**
5: Add (t_1, t_2) to $B_i.G^T$
6: **for** every vertex v in $B_i.G^T$ **do**
7: **if** B_{i-1} is marked as completed, all predecessors of v in $B_i.G^T$ are executed and there is no missing edge in $\mathcal{E}$ involving v **then**
8: **if** v is a single transaction **then**
9: Execute transaction v
10: **else** ▷ v contains a Condorcet cycle
11: Let be $v_1, v_2, ...v_n$ be a Hamiltonian cycle of v
12: Execute transactions in the specified order
13: **if** All transactions of Block $B_i.G^T$ are executed **then**
14: Mark block B_i as completed

the same vertices and as few edges as possible. A transitive reduction of a graph $G^C = (V^C, E^C)$ is graph $G^T = (V^T, E^T)$ where (1) $V^C = V^T$ (the same set of vertices) and (2) for each pair of vertices v and u in G^T, there is a path from u to v in G^T if and only if there is a path from u to v in G^C [6]. Since the input graph G^C is finite and acyclic, its transitive reduction G^T is unique and is a sub-graph of G^C (i.e., the minimum equivalent graph).

In Figure 3 and in round 2, graph G^T has four fewer edges compared to graph G^C. For instance, (t_6, t_9) is removed as t_9 is reachable from t_6 through t_8.

Step 5: updating previous proposals. The leader also needs to update the previous proposals by adding the missing edges between data-dependent transactions. As explained earlier, each replica r sends a set of updated local ordering dependencies U_r to the leader in its local-order message. When the current leader receives an ordering dependency, i.e., an edge, between two data-dependent transactions t_1 and t_2 by at least $n(1-\gamma)+f+1$ replicas on some previous proposal, the leader adds an edge to its updated global ordering dependencies list U_g. If the leader receives both (t_1, t_2) and (t_2, t_1), each from at least $n(1-\gamma)+f+1$ replicas, the edge with the highest weight will be added to the list U_g. The leader also removes the edge from the list of missing edges $\mathcal{E}$.

Step 6: global order proposal. Once the graph is generated, the leader π multicasts a $\langle$GLOBAL-ORDER, G^T, $\mathcal{E}$, $\mathcal{L}$, U_g, $\mathcal{L}_u\rangle_{\sigma_\pi}$ message to all replicas. The global-order message includes the dependency graph G^T, the set of missing edges $\mathcal{E}$, the set $\mathcal{L}$ of $n-f$ local dependency graphs received from different replicas, the updated global ordering dependencies list U_g, and the set $\mathcal{L}_u$ of $n-f$ updated local ordering dependencies received from different replicas. The set $\mathcal{L}$ and $\mathcal{L}_u$ are included to enable replicas to verify the dependency graph and the lists constructed by the leader.

4.3 Order Finalization

The order finalization is performed by all replicas to finalize the order proposed by the leader and execute transactions. As shown in Algorithm 3, order finalization consists of three main steps. First, replicas validate the proposed order. Second, all replicas establish agreement on the proposed order using a BFT protocol, and finally, replicas execute transactions following the proposed order.

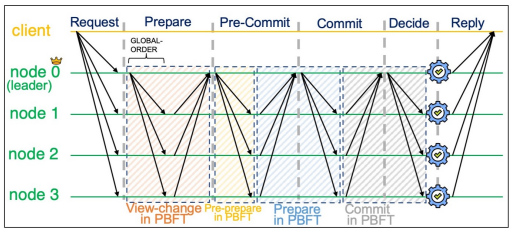


Figure 4: HotStuff protocol

Step 1: global order validation. Upon receiving a global-order message from the leader, each replica first validates the generated graph G^T, the list of missing edges $\mathcal{E}$ and the list of updated global ordering dependencies U_g (Algorithm 3, lines 1-2). To validate graph G^T, the replica ensures that fixed and pending transactions are labeled correctly, and edges are added only if the order is proposed by a sufficient number of replicas. Similarly, the replicas validate both $\mathcal{E}$ and U_g lists by checking the missing edges of the current and previous proposals.

Step 2: establishing agreement. Once the global-order message is validated, replicas establish agreement on the proposed order using the utilized consensus protocol. The current deployment of Rashnu uses HotStuff [82] as the underlying BFT protocol, enabling us to compare Rashnu with the existing order-fairness protocol, i.e., Themis [52]. HotStuff is a leader-based BFT protocol with two main properties. First, it provides linear communication complexity (rather than quadratic as in most BFT protocols, e.g., PBFT). Specifically, each all-to-all communication phase of PBFT is replaced with two linear phases in HotStuff; one from the replicas to the leader and one from the leader to the replicas. Second, HotStuff uses the leader rotation technique, where the leader is replaced after every single proposal in a predetermined manner (round-robin). This is in contrast to most existing protocols that rely on a stable leader, and the leader is changed only when it is suspected to be faulty. HotStuff, as shown in Figure 4, processes each request in four phases of communication: prepare, pre-commit, commit and decide. It should be noted that Rashnu can be easily integrated with any other leader-based BFT protocols, e.g., PBFT [24] or SBFT [46].

Step 3: transaction execution. Once the consensus is achieved, each replica updates the previous proposals by adding edges from U_g. Replicas start executing transactions of a block once all predecessor blocks are executed, i.e., marked as completed. A block B_i is marked as completed if its dependency graph $B_i.G^T$ has no missing edges, and all its transactions have been executed. In Rashnu, replicas follow the edges in the final dependency graph in executing transactions and are able to execute data-independent transactions of a block in parallel. Each vertex of a block is a strongly connected component consisting of either a single transaction or a set of transactions that construct a cycle. For each vertex that is a cycle, first, a Hamiltonian cycle is identified. A Hamiltonian cycle is a cycle that visits each vertex exactly once. If the graph includes more than one Hamiltonian cycle, all replicas deterministically use one, e.g., based on transaction ids, and execute transactions of the cycle in that order (algorithm 3, lines 6-12).

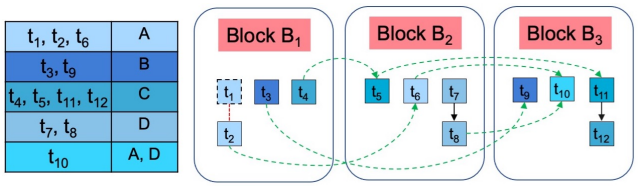


Figure 5: Cross-block data dependency

4.4 Performance Optimizations

We briefly discuss two optimizations that can be used by Rashnu to improve performance.

Cross-block parallel execution. While Rashnu executes data-independent transactions within a block in parallel, replicas still execute transactions block by block. Specifically, each replica waits for block B_i to be marked as completed before executing any transactions of B_{i+1}. This technique simplifies the execution process because the leader constructs a dependency graph for each block independently and there is no need to capture data dependencies across consecutive blocks. However, this might result in unnecessary latency for two main reasons. First, data-independent transactions can be executed in parallel. Hence, when transactions of a current block are being executed, there is no need for data-independent transactions of a successor block to wait. Second, when some edges are missing, no transactions from any successor blocks can be executed, even if the transaction order of a successor block does not depend on the missing edges. To address this issue, Rashnu can capture data dependencies across blocks. While the overhead of capturing data dependencies across blocks might be high in contentious workloads, it results in a significant performance gain in workloads with low to moderate contention.

Figure 5 demonstrates three consecutive blocks and their dependency graphs. Let's assume that $\mathcal{E} = \{(t_1, t_2)\}$, i.e., the edge between transaction t_1 and t_2 is missing. In the basic execution model, t_3 and t_4 can be executed, however, all transactions of blocks B_2 and B_3 wait for block B_1 to be marked as completed before being executed. In the optimized model, Rashnu captures dependencies within and across transaction blocks. Figure 5 shows cross-block dependencies (dashed green lines). For example, transaction t_9 can be executed as soon as transaction t_3 is executed. Once a transaction is executed, it is removed from the cross-block dependency graph.

Transaction dissemination. In determining the order of two valid transactions t_1 and t_2, Rashnu considers the set of replicas η that broadcast both t_1 and t_2. This might result in unspecified ordering between transactions, especially when replicas receive different sets of transactions within a round, due to the asynchronous nature of the network. The unspecified orders need to be captured in the following blocks and when replicas receive the missing transactions. To reduce the number of such deferred orderings, Rashnu can assume that all honest replicas eventually broadcast every transaction. This means that an honest replica that has received only one of the two transactions will also receive the other transaction later. This is a reasonable assumption because a replica that has received only the first transaction should eventually include the second transaction to be able to establish fair order. It also corresponds to how atomic broadcast is used in practice [22].

4.5 Correctness Argument

In this section, we briefly discuss the order-fairness, safety, and liveness of Rashnu. Some arguments are inspired by the correctness arguments of Themis [52].

Lemma 4.1. If transaction t appears in $n-2f$ local-order messages (i.e., fixed transaction), t is proposed by a leader.

Proof. The leader includes all fixed and pending transactions in its graph G and only the graph pruning step removes pending vertices (with no outgoing path to a fixed vertex) from the condensation graph. Since a vertex in the condensation graph is pending if it does not contain any fixed transactions, fixed transactions will always be proposed by the leader. Note that, the leader needs to send all local orderings to replicas (as a proof of correct construction of G), hence, if it maliciously excludes a fixed transaction, replicas detect that and do not accept the proposal. □

Lemma 4.2. Given two data-dependent transactions t_1 and t_2 in a leader proposal. The proposal includes either (t_1, t_2) or (t_2, t_1).

Proof. Rashnu assumes a partial synchrony model where at least $n - 2f$ replicas in the quorum (eventually) have sent both transactions to the leader. Since $n - 2f > 2(n(1-\gamma) + f)$, at least $w(t_1, t_2)$ or $w(t_2, t_1)$ is equal or greater than $n(1-\gamma)+f+1$. Since the leader adds only one edge (the edge with higher weight) even when both $w(t_1, t_2)$ are $w(t_2, t_1)$ are equal or greater than $n(1-\gamma)+f+1$, the final graph includes either (t_1, t_2) or (t_2, t_1) but not both. □

Lemma 4.3. The graph G^T proposed by the leader is acyclic.

Proof. While graph G might contain (Condorcet) cycles, each cycle is part of a vertex (i.e., strongly connected component) in the condensation graph G^C. Since graph pruning and transitive reduction generation steps do not introduce any new edges, the final graph is still acyclic. Note that, for transactions within a strongly connected component, replicas deterministically identify a Hamiltonian cycle and order transactions. □

Lemma 4.4. Given two order-dependent transactions t_1 and t_2 received in round i where t_1 is fixed. If the leader proposal includes only t_1 then there are at least $n(1-\gamma)+1$ honest replicas that have received t_1 before t_2.

Proof. If transaction t_2 was a fixed transaction, the leader proposal must include it. Hence, t_2 is not fixed. If t_2 is not a pending transaction, it has been received by at most $n(1-\gamma)+f$ replicas. Since t_1 is fixed, it appears in $n-2f$ local-order messages. As a result, $p = (n-2f) - (n(1-\gamma)+f) = \gamma n - 3f$ replicas ordered t_1 before t_2. $n > \frac{4f}{2\gamma-1}$, hence, $p > n(1-\gamma)+f$, from which at most f replicas might be faulty. Hence, at least $n(1-\gamma)+1$ honest replicas received t_1 before t_2. If t_2 is a pending transaction, since it is not included in the leader proposal, there is no path from t_2 to any fixed transactions that includes t_1. As a result, either (1) $w(t_2, t_1) \leq n(1-\gamma)+f$ or (2) $n(1-\gamma)+f+1 \leq w(t_2, t_1) \leq w(t_1, t_2)$. The second case implies that at least $n(1-\gamma)+1$ honest replicas received t_1 before t_2. In the first case, since t_1 is fixed, $w(t_1, t_2) \geq (n-2f)-(n(1-\gamma)+f) = \gamma n-3f > n(1-\gamma)+f$ since $n > \frac{4f}{2\gamma-1}$. □

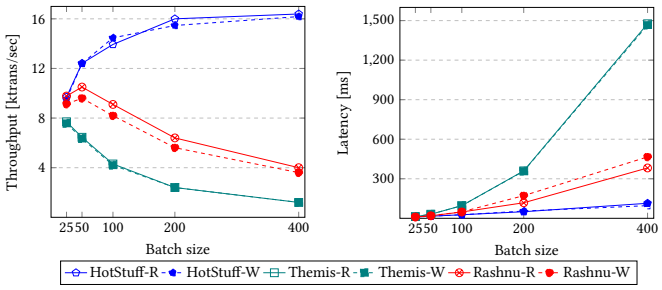


Figure 6: Impact of batch size

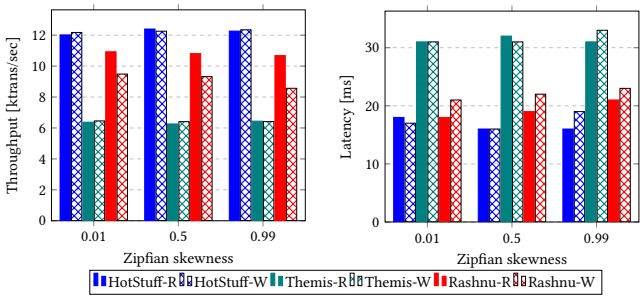


Figure 7: Impact of workload contention

Lemma 4.5. Given two order-dependent transactions t_1 and t_2, if the leader proposal includes only t_1 (and t_2 was not in an earlier proposal) and t_2 is received before t_1 by γn replicas, t_1 and $_2$ are in the same Condorcet cycle.

Proof. Since t_1 is included in the proposal, it is either fixed or pending. If t_1 is a fixed transaction, there are at least $n(1-\gamma)+1$ honest replicas that have received t_1 before t_2 (lemma 4.4); which contradict the condition (i.e., t_2 is received before t_1 by γn replicas). Hence, t_1 is pending. Since t_1 is pending there is a path $t_1, t_a, t_b, ..., t_k$ from t_1 to some fixed transaction t_k in the leader proposal. Since t_k is a fixed transaction, based on lemma 4.4, there are at least $n(1-\gamma)+1$ honest replicas that have received t_k before t_2. Since t_2 is received before t_1 by γn replicas, $t_1, t_a, t_b, ..., t_k, t_2$ construct a Condorcet cycle. □

Theorem 4.6. Rashnu guarantees data-dependent order-fairness.

Proof. Given two data-dependent transactions t_1 and t_2 where γn replicas receive t_1 before t_2. Four cases can happen in the final ordering. First, t_1 and t_2 are proposed in the same block and in different vertices of the graph G^T. Second, t_1 and t_2 are proposed in the same block and in the same vertex of the graph G^T. Third, t_2 is proposed in a later block than t_1 (lemma 4.4), and Fourth, t_1 is proposed in a later block than t_2 (lemma 4.5). Hence, Rashnu guarantees data-dependent order-fairness. □

Theorem 4.7. Rashnu guarantees safety.

Proof. The safety of Rashnu is a direct consequence of the safety of HotStuff [82], as Rashnu does not modify any phases of the underlying agreement protocol. □

Theorem 4.8. Rashnu guarantees liveness.

Proof. Rashnu considers a partial synchrony model where a correct client transaction t will be (eventually) received by all replicas. As a result, t will appear in at least $n-2f$ local-order messages (either in the same round or different rounds), becomes a fixed transaction, and is proposed by a leader. The execution of t only depends on missing edges between previously proposed order-dependent transactions and as soon as such edges are added (i.e., corresponding transactions appear in $n-2f$ local-order messages), t can be executed. However, the order of transaction t does not depend on any transaction that has not been proposed. Hence, in contrast to some other fair ordering protocols like Aequitas [53], Condorcet cycles can not be chained and violate liveness. □

5 Experimental Evaluation

Our evaluation has two main goals. First, measuring the overhead of supporting order-fairness in Rashnu compared to its (unfair) underlying BFT protocol; HotStuff [82]. Second, Comparing the performance of Rashnu and the state-of-the-art order-fairness protocol; Themis [52]. To this end, we analyze the impact of the following parameters on the performance of HotStuff, Rashnu, and Themis:

(1) The transaction batch size (Section 5.1),
(2) the degrees of contention (Section 5.2),
(3) the network size (Section 5.3),
(4) the order-fairness parameter (Section 5.4), and
(5) the geo-distribution of replicas (Section 5.5),

Our Rashnu implementation is bootstrapped from the HotStuff protocol [82]. We use the author's open-source libhotstuff codebase [3] and implemented Rashnu on top of that. We mainly change the HotStuff codebase by enabling the leader to generate a fair order and the replicas to send their local order to the leader and validate the proposed order. Other phases of the HotStuff protocol remain untouched. Since execution routine is unchanged, all transactions are executed sequentially. We have also implemented Themis on top of HotStuff in the same way as Rashnu to enable a fair comparison. We perform our experimental evaluation under the SmallBank benchmark. We initially populate the system with 10000 records and run each protocol under read-heavy ($P_w = 0.05$) and write-heavy ($P_w = 0.95$) workloads, e.g., Rashnu-R is Rashnu under the read-heavy, and Rashnu-W is Rashnu under the write-heavy workload. To determine the accounts accessed by each transaction, a Zipfian distribution is followed, which can be configured in terms of skewness, e.g., $s = 0$ corresponds to a uniform distribution.

We run our experiments on a set of c6220 bare-metal machines on CloudLab [35], each with two Xeon E5-2650v2 processors (8 cores each, 2.6Ghz), 64GB RAM and two 1TB SATA 3.5" 7.2K rpm hard drives. These machines are connected by two networks, each with one interface: (1) a 1 Gbps Ethernet control network; (2) a 10 Gbps Ethernet commodity fabric. We report latency and throughput. The results reflect end-to-end measurements from the clients.

5.1 Performance with Different Batch Sizes

In the first set of experiments, we measure the impact of transaction batch size on the performance of different protocols. In this set of experiments, the number of replicas is assumed to be 5 ($4f+1$ where $f=1$), the order-fairness parameter $\gamma = 1$, and the account selection follows a uniform distribution. Figure 6 depicts the results for all three protocols. The solid and dashed lines are used for read-heavy and write-heavy workloads, respectively. When

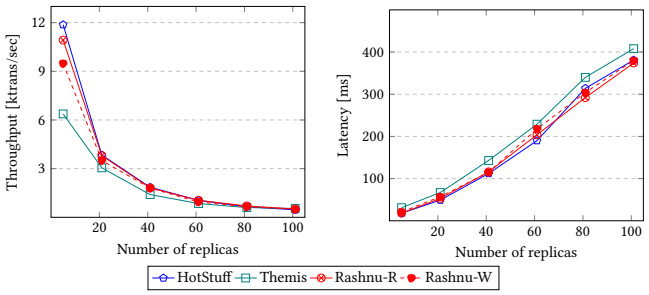


Figure 8: Impact of the network size

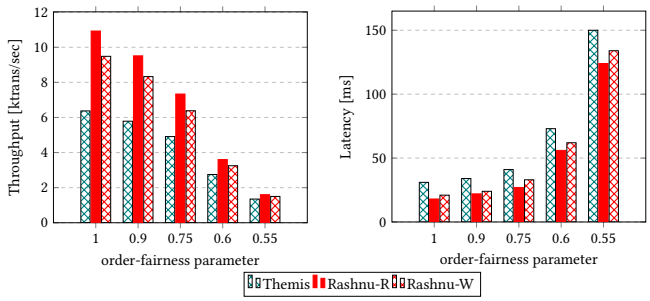


Figure 9: Impact of the order-fairness parameter

blocks are small (block size = 25), Rashnu provides fairness with zero overhead, i.e., Rashnu processes 9775 tps while HotStuff processes 9700 tps, both with 10 ms latency. This is because the cost of generating small dependency graphs is insignificant compared to running consensus among replicas. Increasing the block size, however, results in a gap between HotStuff and Rashnu due to the overhead of fair transaction ordering; generating local dependency graphs, constructing the global dependency graph, and validating the final order. Nevertheless, with block size = 50, Rashnu incurs only 15% throughput overhead; while with the same setting, Themis suffers from 45% throughput overhead. Further increasing the block size makes the gap between HotStuff and Rashnu larger. This is expected because Rashnu must construct larger local and global graphs, requiring checking more and more dependencies. However, compared to Themis, Rashnu shows 233% higher throughput and 74% lower latency with block size 400, demonstrating an efficient order-fairness protocol. The type of workload has negligible impact on the performance of HotStuff and Themis, as expected. The performance of Rashnu, however, is reduced by 7% to 10% in write-heavy workloads with different block sizes, compared to read-heavy ones. This is because, with more write operations, Rashnu needs to capture more data dependencies

5.2 Varying the Degree of Contention

In the next set of experiments, we study the impact of the workload contention by changing the Zipfian skewness of the smallbank benchmark from $s = 0.01$ (uniform distribution) to $s = 0.5$ and $s = 0.99$ (contentious workload). In this set of experiments, the batch size is 100, $n = 5$, and $\gamma = 1$. As shown in Figure 7, the performance of HotStuff and Themis are not affected by increasing the workload skewness due to the fact that they do not construct dependency graphs. Rashnu, however, shows $\sim$ 10% higher latency (in both read-heavy and write-heavy workloads) and 2% and 10% throughput reduction in read-heavy and write-heavy workloads when we increase the Zipfian skewness from $s = 0.01$ to $s = 0.99$. Overall, Rashnu incurs 22% throughput reduction and 27% higher latency by going from a uniform read-heavy workload to a skewed write-heavy workload. This, in fact, is the overhead of constructing local and global dependency graphs by replicas and the leader. Note that, even with $s = 0.99$ and $P_w = 0.95$, Rashnu demonstrates 34% higher throughput and 31% lower latency compared to Themis.

5.3 Performance with Different Network Size

In the third set of experiments, we measure the performance of Rashnu in networks with different sizes, i.e., 5, 21, 41, 61, 81, and 101. We consider request batches of size 100, $\gamma = 1$, and uniform account selection. Since the type of workloads (i.e., read-heavy and write-heavy), does not have a significant impact on the performance of HotStuff and Themis, we only report the results of the read-heavy workload ($P_w = 0.05$) for those two protocols. As depicted in Figure 8, increasing the number of replicas significantly reduces the performance of all three protocols. This is expected because establishing consensus among a large set of replicas is expensive due to the high communication cost. Interestingly, with more than 20 replicas, Rashnu demonstrates almost the same performance as HotStuff; providing order-fairness for free. This is because the cost of consensus in HotStuff becomes much higher than the overhead of fair ordering by Rashnu in a large network. This also shows that the low performance of Rashnu in large networks is caused by the HotStuff consensus routine and if Rashnu is bootstrapped from a high-performance protocol, it can produce better results.

5.4 Varying the Order-fairness Parameter

We next run Themis and Rashnu under different values of the order-fairness parameter γ. Since the network size is a function of the order-fairness parameter (i.e., $n = \frac{4f}{2\gamma - 1} + 1$), reducing γ requires a larger n. Specifically, with $f = 1$, we evaluate protocols with $\gamma = 1$ ($n = 5$), $\gamma = 0.9$ ($n = 6$), $\gamma = 0.75$ ($n = 9$), $\gamma = 0.6$ ($n = 21$), and $\gamma = 0.55$ ($n = 41$). In all experiments, the batch size is 100, and the account selection is uniform. The results for Rashnu-R, Rashnu-W, and Themis are shown in Figure 9. Similar to Figure 8, increasing the number of replicas (resulting from reducing γ) decreases the overall performance. Since the cost of communication between replicas dominates the fair ordering overhead, the gap between Rashnu and Themis becomes smaller by increasing the number of replicas; while Rashnu-R demonstrates 75% higher throughput than Themis with $\gamma = 1$ ($n = 5$), it shows only 19% higher throughput with $\gamma = 0.55$ ($n = 41$).

5.5 Performance in a Geo-distributed Setting

In the last set of experiments, we measure the performance of protocols in an emulated geo-distributed setup. We repeat the first set of experiments (Section 5.1) with an extra 50 ms latency (injected by Linux `netem`) for communication between any two replicas. The results can be seen in Figure 10. As expected, the throughput of all three protocols decreases once network latency is added. Interestingly, Rashnu and Themis reach their pick performance on larger block sizes, compared to the local setting (Figure 6). Specifically, while Rashnu demonstrated its best throughput with block size 50

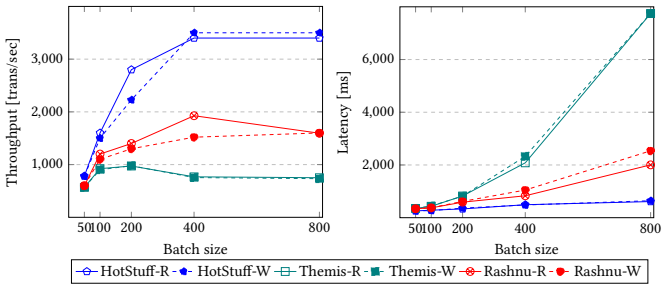


Figure 10: Impact of batch size in a distributed setting

in the local setting, it shows its highest throughput with a block size of 400 in the distributed setting. Similarly, Themis shows its best throughput on the block size of 200. This demonstrates a trade-off between communication latency and fair ordering. With large latency, the cost of communication becomes higher than the overhead of fair ordering, hence, large block sizes are beneficial. While with low communication latency, the overhead of fair ordering is much higher, thus smaller blocks give better performance. While with a block size of 400, Rashnu incurs 45% throughput reduction compared to HotStuff, its throughput is still 2.5 times the throughput of Themis. In this setting, Rashnu processes transactions with 61% lower latency compared to Themis.

6 Related Work

Order-fairness has been recently studied by a few protocols, e.g., Wendy [59, 60], Pompe [84], Aequitas [53], Themis [52] and Quick order fairness [22]. Both Wendy and Pompe rely on synchronized clocks between replicas, making these protocols impractical in asynchronous networks. In Wendy, all replicas have access to synchronized local clocks and if all honest replicas receive transaction t_1 before t_2, then t_1 is delivered before t_2. Pompe, on the other hand, uses a pre-ordering phase and determines the fair order using timestamps assigned by replicas in the pre-ordering phase. Specifically, Pompe orders transaction using their median timestamp. The median timestamp, however, can easily be manipulated by a malicious node that assigns a big timestamp. Moreover, Pompe is vulnerable to censorship [52]. Furthermore, both notions of timed order fairness, used in Wendy, and ordering linearizability, used in Pompe, are strictly weaker than order-fairness studied in Aequitas [53] and Themis [52], as stated in [51]. Aequitas [53] presents the notion of batch-order fairness where all transactions involved in a cycle are delivered to replicas in the same batch. While Aequitas circumvents the Condorcet paradox, it suffers from a liveness issue when Condorcet cycles chain together and extend for an arbitrarily long time. A subsequent study extends the Aequitas approach to permissionless settings [51]. Quick-order-fairness [22] leverages batch-order fairness and also introduces the notion of differential order fairness, inspired by the differential validity notion of consensus. Differential order fairness states that when the number of honest replicas that sends a transaction t_1 before a transaction t_2 is at least $2f + k$ more than the number of replicas that send a transaction t_2 before a transaction t_1 for some order-fairness parameter $k \geq 0$, the protocol must not deliver t_2 before t_1. However, similar to Aequitas, Quick-order-fairness suffers from liveness issues. Moreover, Aequitas and Quick-order-fairness have not been validated by any system implementation. Themis [52], as the only fair ordering protocol implementation with no synchronized clocks assumption, extends Aequitas by addressing its weak liveness issue. Themis introduces the notion of deferred ordering where the actual order of transactions might be deferred to a later proposal, enabling the leader to propose a block without waiting. However, Themis suffers from significant performance overhead, as shown in Section 5, resulting from its complex fair ordering routine. Compared to these protocols, Rashnu addressed both Condorcet cycles and weak liveness and demonstrates high performance.

The fair ordering of transactions has been partially addressed in a few BFT protocols. In Aardvark [27], the leader is monitored to ensure that it does not initiate two new requests from the same client before initiating an old request of another client. Similarly, in PBFT [23], replicas keep the requests in a FIFO queue and only stop the view-change timer when the first request in their queue is executed. Prime [7] also introduces a pre-ordering phase where replicas order the received requests locally and share their own ordering with each other. In Hashgraph [11], all replicas construct a hashgraph to capture all send and receive events. These protocols, however, do not address challenges like Condorcet cycles.

Fairness has also been used in the domain of consensus with different definition. In permissionless blockchains, e.g., Proof-of-Work, fairness is used to ensure that the mining rewards obtained by different miners are proportional to their relative computational power [4, 63, 65, 68, 72]. Similarly, fairness has been defined as providing opportunities for every replica to propose and commit its requests using fair leader election or fair committee election [4, 9, 43, 54, 63, 73, 80]. However, a malicious leader can still order transactions unfairly in its turn. As discussed in Section 1, using censorship resistance techniques [69] to ensure that correct transactions are eventually ordered, reputation-based systems [9, 30, 58, 63] to detect unfair censorship, and threshold encryption [9, 21, 69, 79] to hide transactions content in the ordering phase have also been studied to prevent the manipulation of transaction ordering. However, such techniques partially provide order-fairness and are vulnerable to different attacks, e.g., sandwich attack (transaction reordering) in censorship resistance techniques and reputation-based systems or collusion attacks (between clients and the leader) when threshold encryption is used.

7 Conclusion

This paper defines the notion of data-dependent order-fairness to support order-fairness only among data-dependent transactions. We presented a high-performance fair-ordering protocol, Rashnu, that leverages graph-based techniques to achieve order-fairness among data-dependent transactions. Rashnu further utilizes batch ordering and deferred ordering techniques to deal with Condorcet cycles and liveness issues. We implemented a prototype of Rashnu on top of HotStuff and open-sourced its code. Our evaluation demonstrates the efficiency of Rashnu in different scenarios. First, with small batch sizes or in large networks, the overhead of order-fairness in Rashnu is negligible, i.e., Rashnu performs similarly to its underlying consensus protocol HotStuff. Second, Rashnu shows significant performance improvement compared to Themis in different settings, especially in small networks, 27% to 233% throughput improvement on 5 replicas and with varying batch sizes.

References

[1] [n.d.]. Corda. https://github.com/corda/corda.

[2] [n.d.]. Hyperledger Iroha. https://github.com/hyperledger/iroha.

[3] 2018. libhotstuff: A general-purpose BFT state machine replication library with modularity and simplicity. https://github.com/hot-stuff/libhotstuff.

[4] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. 2017. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In *21st Int. Conf. on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[5] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. 2002. {FARSITE}: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*.

[6] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.

[7] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing* 8, 4 (2011), 564–577.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 30:1–30:15.

[9] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. 2018. A fair consensus protocol for transaction ordering. In *Int. Conf. on Network Protocols (ICNP)*. IEEE, 55–65.

[10] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying intrusion-tolerant SCADA for the power grid. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 328–335.

[11] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* (2016).

[12] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. on Innovative Data Systems Research (CIDR)*.

[13] 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).

[14] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: dependable and secure storage in a cloud-of-clouds. *Transactions on Storage (TOS)* 9, 4 (2013), 12.

[15] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2008. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy* 6, 6 (2008), 44–51.

[16] Kenneth P Birman, Thomas A Joseph, Thomas Raeuchle, and Amr El Abbadi. 1985. Implementing fault-tolerant distributed objects. *Trans. on Software Engineering* 6 (1985), 502–508.

[17] Gabriel Bracha. 1984. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Symposium on Principles of distributed computing*. 154–162.

[18] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.

[19] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Annual Technical Conf. (ATC)*. USENIX Association, 49–60.

[20] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. 2016. Corda: an introduction. *R3 CEV, August* 1, 15 (2016), 14.

[21] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual Int. Cryptology Conf.* Springer, 524–541.

[22] Christian Cachin, Jovana Mićić, and Nathalie Steinhauer. 2022. Quick Order Fairness. In *Int. Conf. on Financial Cryptography and Data Security (FC)*. Springer, 1–18.

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

[24] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, Vol. 99. USENIX Association, 173–186.

[25] JP Morgan Chase. 2016. Quorum white paper.

[26] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright cluster services. In *Symposium on Operating systems principles (SOSP)*. ACM, 277–290.

[27] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults.. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 9. USENIX Association, 153–168.

[28] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. 2013. Spanner: Google's globally distributed database. *Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[29] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Comput. J.* 49, 1 (2006), 82–96.

[30] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: a secure, fair and scalable open blockchain. In *Symposium on Security and Privacy (S&P'21)*. IEEE.

[31] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 910–927.

[32] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *European Conference on Computer Systems (EuroSys)*. 34–50.

[33] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Operating Systems Review (OSR)*, Vol. 41. ACM SIGOPS, 205–220.

[34] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. 2013. PoWerStore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 285–298.

[35] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

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

[37] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2019. Sok: Transparent dishonesty: front-running attacks on blockchain. In *International Conference on Financial Cryptography and Data Security*. Springer, 170–189.

[38] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proc. of the VLDB Endowment* 10, 5 (2017), 613–624.

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

[40] Matthias Fitzi and Juan A Garay. 2003. Efficient player-optimal protocols for strong and differential consensus. In *Symposium on Principles of distributed computing (PODC)*. 211–220.

[41] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2013. An intrusion-tolerant firewall design for protecting SIEM systems. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 1–7.

[42] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2016. SieveQ: A layered bft protection system for critical services. *IEEE Transactions on Dependable and Secure Computing* 15, 3 (2016), 511–525.

[43] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*. 51–68.

[44] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Bullshark: Dag bft protocols made practical. *arXiv preprint arXiv:2201.05677* (2022).

[45] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks, 2004.* IEEE, 135–144.

[46] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE/IFIP, 568–580.

[47] Lioba Heimbach and Roger Wattenhofer. 2022. SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance. In *Conf. on Advances in Financial Technologies (AFT)*. ACM, 1–14.

[48] James Hendricks, Gregory R Ganger, and Michael K Reiter. 2007. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 73–86.

[49] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[50] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 165–175.

[51] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. 2022. Order-fair consensus in the permissionless setting. In *ASIA Public-Key Cryptography Workshop*. ACM, 3–14.

[52] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2022. Themis: Fast, Strong Order-Fairness in Byzantine Consensus. *The Science of Blockchain Conf. (SBC)* (2022).

[53] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-fairness for byzantine consensus. In *Annual Int. Cryptology Conf.* Springer, 451–480.

[54] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual Int. Cryptology Conf.* Springer, 357–388.

[55] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. 2013. Survivable SCADA via intrusion-tolerant replication. *IEEE Transactions on Smart Grid* 5, 1 (2013), 60–70.

[56] Ariah Klages-Mundt and Andreea Minca. 2019. (In) stability for the blockchain: Deleveraging spirals and stablecoin attacks. *arXiv preprint arXiv:1906.02152* (2019).

[57] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Security Symposium*. USENIX Association, 279–296.

[58] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Symposium on Security and Privacy (SP)*. IEEE, 583–598.

[59] Klaus Kursawe. 2020. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Conf. on Advances in Financial Technologies (AFT)*. ACM, 25–36.

[60] Klaus Kursawe. 2021. Wendy Grows Up: More Order Fairness. In *Int. Conf. on Financial Cryptography and Data Security (FC)*. Springer, 191–196.

[61] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[62] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.

[63] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. 2019. FairLedger: A Fair Blockchain Protocol for Financial Institutions. In *23rd Int. Conf. on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[64] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 17–30.

[65] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. 2017. SmartPool: Practical Decentralized Pooled Mining. In *USENIX Security Symposium*. USENIX, 1409–1426.

[66] Dahlia Malkhi and Michael K Reiter. 1998. Secure and scalable replication in Phalanx. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*. IEEE, 51–58.

[67] Dahlia Malkhi and Pawel Szalachowski. 2022. Maximal Extractable Value (MEV) Protection on a DAG. *arXiv preprint arXiv:2208.00940* (2022).

[68] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. 2015. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM SIGSAC, 680–691.

[69] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. 31–42.

[70] Louise E Moser, Peter M Melliar-Smith, Priya Narasimhan, Lauren A Tewksbury, and Vana Kalogeraki. 1999. The Eternal system: An architecture for enterprise applications. In *Int. Enterprise Distributed Object Computing Conf. (EDOC)*. IEEE, 214–222.

[71] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. 2018. On the challenges of building a BFT SCADA. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 163–170.

[72] Rafael Pass and Elaine Shi. 2017. Fruitchains: A fair blockchain. In *symposium on Principles Of Distributed Computing (PODC)*. ACM, 315–324.

[73] Rafael Pass and Elaine Shi. 2017. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31 Int. Symposium on Distributed Computing*. 6.

[74] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *Symposium on Security and Privacy (SP)*. IEEE, 198–214.

[75] Tom Roeder and Fred B Schneider. 2010. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)* 28, 2 (2010), 1–54.

[76] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[77] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Int. Conf. on Management of Data*. ACM, 105–122.

[78] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2009. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems* 21, 4 (2009), 452–465.

[79] Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. 2021. Adding Fairness to Order: Preventing Front-Running Attacks in BFT Protocols using TEEs. In *Int Symp on Reliable Distributed Systems (SRDS)*. IEEE, 34–45.

[80] David Yakira, Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, and Ronen Tamari. 2021. Helix: A Fair Blockchain Consensus Protocol Resistant to Ordering Manipulation. *IEEE Transactions on Network and Service Management* 18, 2 (2021), 1584–1597.

[81] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating agreement from execution for byzantine fault tolerant services. *Operating Systems Review (OSR)* 37, 5 (2003), 253–267.

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

[83] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling blockchain via full sharding. In *SIGSAC Conf. on Computer and Communications Security*. ACM, 931–948.

[84] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without Byzantine oligarchy. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 633–649.

[85] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. 2021. High-frequency trading on decentralized on-chain exchanges. In *Symposium on Security and Privacy (SP)*. IEEE, 428–445.

[86] Lidong Zhou, Fred Schneider, Robbert VanRenesse, and Zygmunt Haas. 2002. Secure distributed on-line certification authority. US Patent App. 10/001,588.

[87] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. 2002. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 329–368.