FairDAG: Consensus Fairness over Concurrent Causal Design

Dakai Kang, Junchao Chen, Tien Tuan Anh Dinh[†], Mohammad Sadoghi
Exploratory Systems Lab, University of California, Davis
†Deakin University

ABSTRACT

The rise of cryptocurrencies like Bitcoin and Ethereum has driven interest in blockchain technology, with Ethereum's smart contracts enabling the growth of decentralized finance (DeFi). However, research has shown that adversaries exploit transaction ordering to extract profits through attacks like front-running, sandwich attacks, and liquidation manipulation. This issue affects both permissionless and permissioned blockchains, as block proposers have full control over transaction ordering. To address this, a more fair approach to transaction ordering is essential.

Existing fairness protocols, such as POMPE [52] and THEMIS [29], operate on leader-based consensus protocols, which not only suffer from low throughput but also allow adversaries to manipulate transaction ordering. To address these limitations, we propose FAIRDAG-AB and FAIRDAG-RL, which leverage DAG-based consensus protocols.

We theoretically demonstrate that FAIRDAG protocols not only uphold fairness guarantees, as previous fairness protocols do, but also achieve higher throughput and greater resilience to adversarial ordering manipulation. Our deployment and evaluation on CloudLab further validate these claims.

1 INTRODUCTION

The emergence of cryptocurrencies, including Bitcoin [38] and Ethereum [9] sparked people's interest in blockchain technology. Later, the deployments of smart contracts on Ethereum facilitated the realization of decentralized Finance (DeFi) [6, 7, 42, 51]. To date, the market capitalization of DeFi has reached 70 billion. However, numerous studies [14, 39, 40, 48] have shown that adversaries manipulate transaction order, extracting profits and harming other participants. Typical manipulation mechanisms include front-running, back-running, sandwich attacks, liquidation manipulation, and time-bandit attacks.

Such an "order manipulation crisis" exists in both permissionless blockchains e.g., Ethereum, and permissioned blockchains e.g., PBFT [12] and HotStuff [50], because the block proposers have full control on the selection and ordering of transactions within blocks. A malicious block proposer, such as the leader in Ethereum or PBFT, can set the transaction order within a block to maximize personal profits [14]. The transaction ordering, which can be arbitrarily manipulated by malicious block proposers, is unfair to other participants. In contrast, a fairer way to order transactions is needed.

The key to achieving fair ordering is to prevent the proposers from dominating the transactions within blocks, as existing fairness protocols do [29, 30, 32, 37, 52]. In these protocols, each block does not contain a list of transactions but a set of local orderings signed digitally, where each local ordering represents the sequence in which the signer participant would like the transactions to be ordered. After committing the blocks, a final transaction ordering can be generated based on the local orderings. Fairness properties can be ensured if the committed local orderings include a sufficient number of contributions from correct participants not controlled by the adversary.

Two representative fairness protocols are POMPE [52] and THEMIS [29]. Pompe guarantees *Ordering Linearizability*, which guarantees that transaction T_1 is ordered before T_2 if every correct participant receives T_1 before any correct participant receives T_2. However, a malicious leader can still manipulate the order between any two transactions that arrive at the participants at a similar time, even if every participant receives T_1 before T_2. Then, Themis introduces and enforces a stronger fairness notion, *γ-batch-order-fairness*, which ensures that if a γ proportion of correct participants receive T_1 before T_2, then T_1 will be ordered *no later* than T_2.

We observe that a good fairness protocol should achieve the following goals:

G1 **Limiting Adversarial Manipulation.** A well-designed fairness protocol should restrict the influence of malicious participants on the final transaction ordering, thereby preserving fairness guarantees.
G2 **Minimal Correct Participants.** To mitigate the influence of adversarial participants, fairness protocols aggregate local orderings from correct participants. Given the same number of malicious actors, an effective fairness protocol should ensure fairness properties while relying on the minimal number of correct participants.
G3 **High Performance.** As a specialized class of BFT protocols, fairness protocols should strive for high performance, including high throughput and low latency, similar to traditional BFT protocols.

Unfortunately, existing fairness protocols [29, 30, 52] rely on leader-based consensus protocols, such as PBFT [12] and HotStuff [50], to commit blocks, which hinders their ability to fully achieve fairness objectives. Leader-based consensus protocols rely on a single leader to aggregate local orderings. A malicious leader, however, can manipulate transaction ordering by selectively filtering out unfavorable local orderings. Additionally, even in the absence of adversary, a single leader may become a performance bottleneck if the workload exceeds its capacity.

To address the challenges posed by underlying leader-based consensus protocols, we propose running fairness protocols on top of DAG-based protocols [15, 28, 45], which offer the following features:

- **Leaderless High-Throughput Design**: DAG-based protocols employ a leaderless, multi-proposer design, allowing every participant to propose a block. This approach enhances system throughput.

- **Validity**: Blocks in DAG-based protocols reference blocks from other replicas, forming a *Directed Acyclic Graph (DAG*. Such a design ensures that blocks from correct participants will eventually be committed by all correct participants.

The *leaderless high-throughput design* eliminates the bottleneck caused by a single leader (**G3**). The *validity* reduces the adversary's ability to selectively filter out unfavorable local orderings (**G1**), and thus lowers the requirement on the number of correct participants (**G2**).

Then, we propose FairDAG that runs fairness protocols on top of DAG-based protocols. In FairDAG, each participant proposes its local ordering as a block and broadcasts the blocks, and a final transaction ordering is deterministically generated based on the blocks committed by the DAG-based protocols. By processing committed local orderings and generating the final ordering in different ways, we introduce FairDAG-AB with *absolute ordering* and FairDAG-RL with *relative ordering*, each achieving distinct fairness protocols.

In this paper, our main contributions are:

(1) We propose FairDAG-AB, a DAG-based *absolute* fairness protocol that guarantees fairness property *Ordering Linearizability*. Compared to Pompe, FairDAG-AB achieves higher performance and more effectively limits the adversary's ability to manipulate transaction order.
(2) We propose FairDAG-RL, a DAG-based *relative* fairness protocol that guarantees fairness property *γ-batch-order-fairness*. Compared to Themis, FairDAG-AB not only achieves higher performance and further limits the adversary's ability to manipulate transaction order, but also reduces the requirement for the number of correct participants.
(3) We theoretically demonstrate that FairDAG-AB protocols better limit the adversary's manipulation of transaction ordering compared to previous leader-based fairness protocols.
(4) We conduct experiments that verify the advantages of FairDAG-RL over Themis in terms of fairness.
(5) We evaluate the performance of FairDAG-AB and FairDAG-RL by comparing them with other fairness protocols. The results confirm our claim that FairDAG-AB achieves better performance.

2 BACKGROUND

FairDAG-AB and FairDAG-RL run fairness protocols on top of DAG-based consensus protocols. In this section, we introduce the definitions of three fairness properties and the DAG-based consensus protocols. Starting from this section, our discussion is framed within the context of BFT protocols, where the participants are replicas.

2.1 Receive-Order-Fairness

Definition 2.1. Receive-Order-Fairness: For any two transactions T_1 and T_2 that are both in the local orderings of correct replicas, if all correct replicas receive T_1 before T_2, then T_1 will be ordered *before* T_2.

$R_1 : \{T_1, T_2, T_3, T_4\}$
$R_2 : \{T_2, T_3, T_4, T_1\}$
$R_3 : \{T_3, T_4, T_1, T_2\}$
$R_4 : \{T_4, T_1, T_2, T_3\}$

$$\begin{tikzcd}
 & T_2 \arrow[dr] & \\
T_1 \arrow[ur] & & T_3 \arrow[dl] \\
 & T_4 \arrow[ul] &
\end{tikzcd}$$

Figure 1: Condorcet Cycle

It is impossible to achieve *Receive-Order-Fairness* due to the existence of a *Condorcet Cycle*. In Figure 1, we illustrate this with an example where *Receive-Order-Fairness* cannot hold. We assume there are four replicas, three of which are correct and one is Byzantine, but which one is Byzantine is unknown. The replicas are indexed as R_1, R_2, R_3, R_4, and there are 4 transactions indexed as T_1, T_2, T_3, T_4. As shown on the left side of the figure, each replica R_i receives the 4 transactions in the order $T_i, \ldots, T_4, T_1, \ldots, T_{i-1}$. Since any replica can be Byzantine, all ordering preferences endorsed by at least three nodes are preserved as edges in a directed graph, as shown on the right side of the figure. This results in the formation of a *Condorcet Cycle*.

No matter which transaction we start from and trace a transaction order along the edges, due to not knowing which replica is Byzantine, *Receive-Order-Fairness* cannot always be guaranteed for all transaction pairs. For example, if the final transaction ordering is selected as $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$ and R_1 is Byzantine, then T_4 will be ordered later than T_1 even if all correct replicas receive T_4 before T_1.

2.2 Ordering Linearizability

Ordering Linearizability is a fairness property by Pompe [52] and our work FairDAG-AB. To achieve the fairness property, each replica R assigns monotonically increasing *ordering indicators (oi)* to transactions. The final transaction ordering is then determined based on the ordering indicators. We denote by ois_i^C the set of ordering indicators for T_i from correct replicas. The definition of *Ordering Linearizability* is:

Definition 2.2. Ordering Linearizability: For any two transactions T_1 and T_2, if every ordering indicator in ois_1^C is smaller than every ordering indicator in ois_2^C, i.e., $\forall oi_1 \in ois_1^C, \forall oi_2 \in ois_2^C, oi_1 < oi_2$, then T_1 is ordered before T_2.

For example, we have four replicas that receive four transactions in different orders shown in Figure 2. R_1, R_2 and R_3 are correct replicas while R_4 is Byzantine. For transactions T_1 and T_4, ois_1^C is $\{2, 1, 1\}$ while ois_4^C is $\{3, 4, 4\}$. Every ordering indicator in ois_1^C is smaller than every ordering indicator in ois_4^C. In Pompe and our work FairDAG-AB that guarantee *Ordering Linearizability*, whatever the ordering indicators from R_4 are, T_1 will be ordered before T_4.

2.3 γ-Batch-Order-Fairness

In Section 2.1 we have shown that it is impossible to guarantee *Receive-Order-Fairness*. However, a weaker variant, *γ-Batch-Order-Fairness* can be guaranteed by Themis[29] and our work FairDAG-RL.

$R_1 : \{(T_2, 1), (T_1, 2), (T_4, 3), (T_3, 4)\}$
$R_2 : \{(T_1, 1), (T_3, 2), (T_2, 3), (T_4, 4)\}$
$R_3 : \{(T_1, 1), (T_2, 1), (T_3, 3), (T_4, 4)\}$
$R_4 : \{(T_1, 1), (T_2, 2), (T_3, 3), (T_4, 4)\}$

Figure 2: T_1 will be ordered before T_4 if *Ordering Linearizability* holds regardless of the local ordering from R_4.

$R_1 : \{T_0, T_1, T_2, T_3, T_4, T_5\}$
$R_2 : \{T_0, T_2, T_3, T_4, T_1, T_5\}$
$R_3 : \{T_0, T_3, T_4, T_1, T_2, T_5\}$
$R_4 : \{T_0, T_4, T_1, T_2, T_3, T_5\}$
Final Ordering: $\{T_0, T_1, T_2, T_3, T_4, T_5\}$

(a)

$$\begin{tikzcd}\boxed{b_1 : \{T_0\}} \arrow[d] \\ \boxed{b_2 : \{T_1, T_2, T_3, T_4\}} \arrow[d] \\ \boxed{b_3 : \{T_5\}}\end{tikzcd}$$

(b)

Figure 3: A final ordering of six transactions that satisfies γ-Batch-Order-Fairness with $\gamma = 1$, $\mathbf{n} = 4$ and $\mathbf{f} = 1$.

We say that a transaction T_2 is **dependent** on a transaction T_1, i.e., $T_1 \longrightarrow T_2$, if T_1 must be ordered before T_2. Due to the existence of *Condorcet cycles*, there are **cyclic dependent** transactions.

Definition 2.3. A batch S of transactions is *cyclic dependent* if for any two transactions T_1, T_2 in S, there is a list of transactions $T_1, T_{m_1}, T_{m_2}, ..., T_{m_k}, T_2$ such that:

- $\forall j \le k$, T_{m_j} is dependent on $T_{m_{j-1}}$, i.e., $T_{m_{j-1}} \longrightarrow T_{m_j}$;
- $T_1 \longrightarrow T_{m_1}$; $T_{m_k} \longrightarrow T_2$.

Each final transaction ordering can be split into non-overlapping slices $S_1, S_2, ...$, each of which is a *maximal cyclic dependent batch* such that for any i, S_i, and S_{i+1} cannot be combined into a *cyclic dependent batch*.

We say that a transaction T_1 is ordered **no later** than T_2 if T_1 is in the same or an earlier batch than T_2. Thus, we derive the definition of *γ-Batch-Order-Fairness*:

Definition 2.4. *γ-Batch-Order-Fairness*: For any two transactions T_1 and T_2, if γ fraction of correct replicas, i.e., $\gamma(\mathbf{n} - \mathbf{f})$ correct replicas, receive T_1 before T_2, then T_1 is ordered *no later* than T_2.

For example, as shown in Figure 3 (a), we have four replicas receiving six transactions in different orders, and a final transaction ordering is generated. By splitting the final ordering into three *cyclic dependency* batches (the details related to adding dependencies between transactions will be explained in Section 6), for each transaction pair, the fairness property *γ-Batch-Order-Fairness* holds. For instance, T_0 is received earlier than T_1 by $\gamma(\mathbf{n} - \mathbf{f}) = 3$ correct replicas, the fairness property holds as T_0 is ordered in an earlier batch than T_1.

2.4 DAG-based Consensus Protocols

A DAG-based BFT consensus protocol operates through a series of rounds. In each round r, every participant proposes a block as a DAG vertex. Each vertex references at least $\mathbf{n} - \mathbf{f}$ vertices proposed in the previous round $r - 1$, forming edges in the DAG. The causal history of a vertex consists of all the vertices to which it has a path. For every k rounds, there is a randomly selected or predetermined leader vertex (e.g., in Tusk, $k = 2$). Leader vertices are committed in ascending order of rounds, and the vertices in their causal history are ordered deterministically using a predefined method. Many DAG-based protocols use reliable broadcast protocols (RBC) to disseminate these vertices. With the RBC and well-designed commit rules, DAG protocols guarantee three important properties, even in *asynchronous network*:

- **Agreement**: if a correct replica commits a vertex v, then every other correct replica eventually commits v.
- **Total Order**: if a correct replica commits v before v', then every correct replica commits v before v'.
- **Validity**: if a correct replica broadcasts a DAG vertex v, then each correct replica will commit eventually v.

Compared to leader-based protocols, the multi-proposer design of DAG-based protocols has higher throughput. But the RBC and commit rules also incur higher commit latency.

3 SYSTEM MODEL

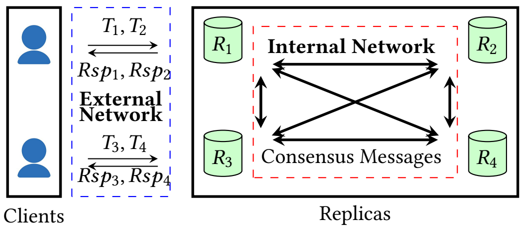

Figure 4: Network topology with Clients, Replicas, and Networks. Rsp_i represents a client response for transaction T_i, including the execution results of T_i. Change arrow color. No gap for external network.

3.1 Clients

Clients generate and submit transactions to replicas, then await execution results in response. There is no assumption that clients always behave correctly; they may exhibit arbitrary or malicious behavior.

3.2 Replicas

In the system, there are a total of $\mathbf{n}$ replicas and an *adaptive adversary* capable of corrupting up to $\mathbf{f}$ replicas during execution. The corrupted replicas, referred to as Byzantine or malicious replicas, may exhibit arbitrary malicious behavior.

Regarding the fairness of transaction ordering, Byzantine replicas can falsify their local orderings and attempt to filter out unfavorable ones. In contrast, correct replicas strictly adhere to the protocols, honestly presenting local orderings in the sequence in which they receive client transactions.

Different fairness protocols guarantee various fairness properties and exhibit different levels of tolerance to corruption, i.e., different minimal ratios between $\mathbf{n}$ and $\mathbf{f}$. FairDAG-AB and Pompe require

$\mathbf{n} \geq 3\mathbf{f}$; Themis requires $\mathbf{n} > \frac{(2\gamma+2)\mathbf{f}}{2\gamma-1}$; and FairDAG-RL requires $\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{2\gamma-1}$, $\frac{1}{2} < \gamma \geq 1$.

3.3 Authentication

We assume *authenticated communication*, where Byzantine replicas may impersonate each other but cannot impersonate correct replicas. To ensure authenticated communication, we employ *digital signatures* [27]. Additionally, we use d_i to denote the message digest of a transaction T_i, which is computed using a *secure collision-resistant cryptographic hash function* [27].

3.4 Network

We classify the network into two categories: *internal network* and *external network*. As shown in Figure 4, the *internal network* involves communication exclusively among replicas, while the *external network* refers to communication between clients and replicas.

Non-adversarial External Network: No assumptions are made regarding the synchrony of the external network. However, the *external network* is assumed to be *non-adversarial*, meaning there is no adversary with full control over the network. If such an adversary existed, they could manipulate transaction arrival times and ordering, thereby undermining any fairness guarantees.

The *internal network* can operate in either an *asynchronous* or *partial synchronous* manner.

Asynchronous Internal Network: Messages are not dropped and are eventually delivered to all replicas. However, there is no assumption about the message delivery time.

Partially Synchronous Internal Network: There is an unknown *Global Stabilization Time (GST)* such that after *GST* there is a message delay bound Δ, i.e., a message sent at time t will be delivered by $max(t, GST) + \Delta$.

4 OVERVIEW

Previous fairness protocols, such as Pompe and Themis, run on top of leader-based consensus protocols. Though the final ordering is generated based on local orderings from a quorum of replicas, the single leader has control over selecting local orderings, undermining the fairness of the final ordering. Moreover, a faulty or slow leader is prone to be the performance bottleneck of the system.

In FairDAG, to mitigate the possible negative impacts from the leader, we run fairness protocols on top of leaderless DAG protocols rather than leader-based protocols. FairDAG consists of two layers: *DAG Layer* and *Fairness Layer*. The *DAG Layer* is responsible for disseminating (Figure 5 (a,b)) and committing (Figure 5 (c)) local orderings in which replicas receive transactions. The *Fairness Layer*, taking the committed local orderings as input (Figure 5 (d)), runs fair-ordering mechanisms (Figure 5 (e)) to generate a final transaction ordering (Figure 5 (f)) satisfying fairness properties.

4.1 DAG Layer

In DAG layer, we run existing protocols, such as Tusk, DAG-Rider and Bullshark, with necessary modifications that make them suitable for guaranteeing fairness properties.

Directed-Acylic-Graph

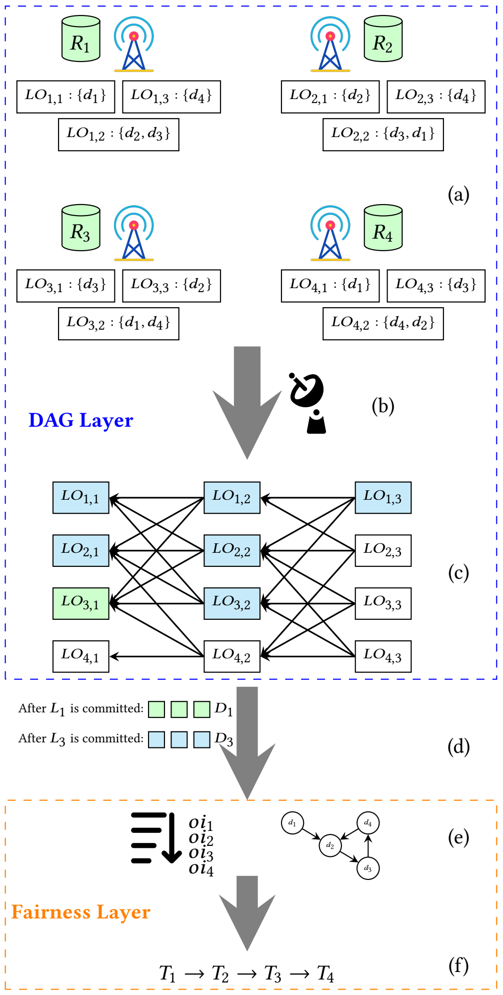

Figure 5: Overview Architecture of FairDAG: (a) Each replica reliable broadcast blocks containing its local ordering. (b) Each replica receives blocks delivered through Reliable Broadcast. (c) Each replicas forms a local view of the DAG using received blocks and reference links, where different colors represent causal history of different committed leader vertices. (d) Local orderings in D_r are used as input of the fairness layer after L_r is committed. (e) Transaction are ordered based on the committed local orderings using fair-ordering mechanisms.

DAG protocols proceed round by round, in which all replicas propose new blocks concurrently. Each replica proposes a new block per round r, which should refer to blocks in the previous rounds. Thus, taking the blocks as **vertices** and the reference links as **edges**, a Directed-Acyclic-Graph (DAG) is formed. A vertex of round r can refer to at least $\mathbf{n}-\mathbf{f}$ vertices in round $r-1$ and at most $\mathbf{f}$ vertices of rounds lower than $r-1$. We name the reference links to vertices of round $r-1$ **strong edges** and the reference links to vertices of lower rounds **weak edges** .

We denote by $v_{i,r}$ the block proposed by replica R_i in round r. A valid block $v_{i,r}$ should contain:

- *strong_edges*: at least $\mathbf{n} - \mathbf{f}$ *strong edges*.
- *weak_edges*: at most $\mathbf{f}$ *weak edges*.

An example DAG is shown in Figure 5 (c), in which the squares are vertices (blocks), the black arrows between vertices are strong edges and the blue arrows are weak edges. By default, we require a valid vertex $v_{i,r}$ to contain a strong edge to $v_{i,r-1}$ if $r > 0$.

Proposing a Vertex

In traditional DAG protocols, each vertex contains a list of transactions, the order of which is decided by the replica that proposes. However, in FairDAG, to generate a final ordering of transactions that reflects the opinions of most replicas and satisfies the fairness properties, each vertex proposed by replica R contains a piece of LO_R, which is a local ordering of transactions ordered based on the time that R receives the transactions.

Fairness protocols can be partitioned into two categories, with local orderings of different types:

- **Absolute**: Each replica assigns monotonically increasing ordering indicators, e.g., sequence number or timestamp, to the transactions, forming a local ordering.
- **Relative**: Each replica assigns monotonically increasing sequence numbers to the transactions, forming a local ordering.

To reduce communication overhead, for transactions within the local orderings, each vertex only needs to include the digests (hashes) of the transactions rather than complete contents.

Thus, for *relative-ordering* protocols such as FairDAG-RL, as shown in Figure 5 (a), the local ordering within each vertex is a list of transaction digests, e.g., $\{d_1, d_2, d_3\}$; for *absolute-ordering* protocols such as FairDAG-AB, the local ordering within each vertex is a list of transaction digests with ordering indicators, e.g., $\{(d_1, 1), (d_2, 3), (d_3, 5)\}$.

Forming a DAG

As shown in Figure 5 (a) and (b), each replica broadcasts its vertices and receives vertices from other replicas through reliable broadcast mechanisms [15].

Then, as shown in Figure 5 (c), each replica forms a local view of the DAG using the vertices sent and received. The DAG protocols guarantee that all replicas eventually have consistent local views of the DAG.

Committing DAG Vertices

In DAG protocols, rounds are split into waves. Each wave has one or multiple leader vertices. We denote by L_{r_i} a *committed* leader vertex of round r_i and by C_{r_i} the causal history of L_{r_i}, containing all vertices that L_{r_i} have a path to, including L_{r_i} itself. A leader vertex is *committed* if certain conditions are met. The DAG protocols guarantee that all correct replicas would commit the leader vertices in a consistent round-increasing order $(L_{r_1}, L_{r_2}, ...)$, in which $C_{r_i} \subset C_{r_{i+1}}$ holds for any i.

We say that a vertex v is *dependent* on a committed leader vertex L_{r_i}, if $v \in C_{r_i}$ and $\forall j < i, v \notin C_{r_j}$. We denote by D_{r_i} the set of DAG vertices that are dependent on L_{r_i}. With the consistent order of committed leader vertices, we can partition the DAG into non-overlapping parts $(D_{r_1}, D_{r_2}, ...)$.

For example, as shown in Figure 5 (c), we assume committed leader vertex L_1 to be the green vertex with $LO_{3,1}$ and L_3 to the blue vertex with $LO_{1,3}$. Then the green vertices represent D_1, while the blue vertices represent D_3.

As shown in Figure 5 (d), every time a leader vertex L_r is committed, the fairness layer takes DAG part D_r as input to generate a final ordering of transactions. Using D_r as input can ensure that the final ordering reflects the opinions of most non-faulty replicas because DAG protocols guarantee that each D_r contains local orderings from at least $\mathbf{n} - \mathbf{f}$ distinct replicas, at least $\mathbf{f} + 1$ of which are from correct replicas.

4.2 Fairness Layer

Taking local orderings within the committed causal history as input, different types of fairness protocols use different mechanisms to generate final orderings of transactions.

As shown by the left half of Figure 5 (e), with **absolute** fairness protocols, each transaction gets an assigned ordering indicator calculated on the basis of local orderings contained in the input DAG parts. The transactions are then ordered according to their assigned ordering indicators, generating the final ordering (Figure 5 (f)). The relative fairness protocols guarantee the *Ordering Linearizability* property.

As shown by the right half of Figure 5 (e), with **relative** fairness protocols, dependency graphs of transactions are formed on the basis of local orderings contained in the input DAG parts. Within the dependency graphs, each node is the transaction, and the edge direction between each pair of nodes is determined by their positions in the local orderings. Finally, a final ordering is formed by finding *Hamilton Paths* within the graphs (Figure 5 (f)). The relative fairness protocols guarantee the *γ-Batch-Order-Fairness* property.

5 FAIRDAG-AB DESIGN PRINCIPLES

FairDAG-AB is a **absolute** fairness protocol. Each transaction gets an *assigned ordering indicator* based on local ordering indicators from a quorum of replicas. Transactions are then ordered based on their assigned ordering indicators.

5.1 Transaction Dissemination

In FairDAG-AB, clients broadcast their transactions to all replicas rather than sending transactions to only one replica. Clients do so to eliminate a single leader's impact on the final ordering. Otherwise, for example, a replica can back-run certain transactions by deliberately delaying the propagation of some transactions.

5.2 FairDAG-AB Vertex

Operating on top of DAG consensus protocols, each FairDAG-AB replica constructs and reliably broadcasts a DAG vertex upon the fulfillment of specific conditions outlined by the DAG protocols. As an absolute fairness protocol, as described in 4.1, besides the necessary information for forming a DAG, each FairDAG-AB vertex contains the information of client transactions that it has received since the last time it broadcast a DAG vertex:

- *dgs*: a list of transaction digests.
- *ois*: a list of *monotonically increasing* ordering indicators corresponding with *dgs*.

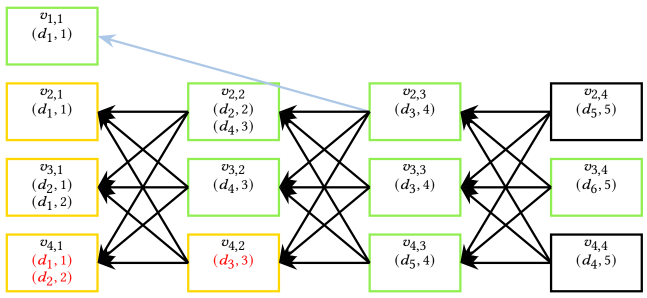

Figure 6: Example of calculating AOI and LPAOI.

5.3 Managing Ordering Indicators

To calculate the assigned ordering indicators, which transactions will be ordered based on, we have designed an *Ordering Indicator Manager*.

We say that replica R has **seen** an ordering indicator oi if oi is in a vertex that has been reliably broadcast and added into R's local view of the DAG. And we say R has **committed** an ordering indicator oi if oi is in a committed vertex in R's local view of the DAG. A replica R gives each transaction digest d a corresponding ordering indicator manager object, containing the following information:

- *seen_ois*: vector of ordering indicators of d that R has seen.
- *committed_ois*: vector of ordering indicators of d that R has committed.
- *LPAOI*: lowest possible assigned ordering indicator of d.
- *AOI*: assigned ordering indicator of d.

FairDAG-AB orders transactions based on *AOI* calculated from *committed_ois* and determines which transactions are safe to be ordered based on the *LPAOI* of transactions (see details later in Section XXX). The *seen_ois* is used for *LPAOI* calculation. Note: Both vectors are indexed from 1 to **n**, and all values within an *ordering indicator manager* object are initialized to be ∞. Following in this paper, when we say a transaction has *committed_ois* values, *seen_ois* values, *LPAOI* or *AOI*, we imply that it has non-∞ values.

Each replica R maintains two maps, a set and a vector in memory.

- *txns_wo_assigned_oi*: mapping from transactions digests **without** *AOI* to their *ordering indicator manager* objects.
- *txns_w_assigned_oi*: mapping from transactions digests **with** *AOI* to their *ordering indicator manager* objects.
- *ordered_txns*: a set of digests of transactions that are already ordered.
- *highest_ois*: the highest ordering indicators received from each replica, initialized to be 0 and indexed from 1 to **n**.

Managing Ordering Indicators

Upon a DAG vertex $v_{i,r}$ is reliably broadcast and added into R's local view of the DAG, R updates $highest\_ois[i]$ to $hoi_{i,r} + 1$, where $hoi_{i,r}$ is the highest ordering indicator within $v_{i,r}$. Also, for each transaction digest d within $v_{i,r}$ without an *AOI*, i.e., $d \notin ordered\_txns$ and $d \notin txns\_w\_assigned\_oi$, R updates the $seen\_ois[i]$ of $txns\_wo\_assigned\_oi[d]$.

Upon a DAG leader vertex L_r is committed and then D_r is input, R updates the *committed_ois* of the transactions within D_r using the contained ordering indicators, again, skipping transactions with *AOI*.

```
In-memory Variables
$txns\_w\_assigned\_oi := \{\}$, $txns\_wo\_assigned\_oi := \{\}$
$ordered\_txns := \{\}$
$highest\_oi\_list := [\,]$, $local\_highest\_oi = 0$
$dgs := [\,]$, $ois := [\,]$
$current\_round$, $replica\_id$, $local\_timer$

Client Thread (processing transactions sent from clients) :
event Upon receiving a transaction $T$ do
    if T is valid then
        $oi := local\_timer.GetNextOI()$
        $ois.push(oi)$
        $local\_highest\_oi := oi$
        $dgs.push(T.digest)$

DAG Layer Thread (forming the DAG) :
event Upon $R$ is ready to propose a DAG vertex do
    $v := DAGVertex(replica\_id, current\_round)$
    $v.dgs := dgs$; $v.ois := ois$
    $dgs.clear()$; $ois.clear()$
    $v.hoi := local\_highest\_oi$
    Reliably Broadcast $v$

event Upon receiving a valid DAG vertex $v_{i,r}$ do
    $highest\_oi\_list[i] := v_{i,r}.hoi + 1$
    for $d, oi \in v_{i,r}.dgs, v_{i,r}.ois$ do
        $txns\_wo\_assigned\_oi[d].seen\_ois[i] := oi$

event Upon committing a DAG leader vertex $L_r$ do
    Send $D_r$ to Ordering Layer

Fairness Layer Thread (Ordering transactions) :
event Upon $D_r$ is received do
    for $v \in D_r$ do
        $i := v.replica\_id$
        for $d, oi \in v.dgs, v.ois$ do
            if $d \notin ordered_nodes$ and $d \notin ordered\_txns$ then
                $txns\_wo\_assigned\_oi[d].committed\_ois[i] := oi$
    $LPAOI_{min} := \infty$
    for $d \in txns\_wo\_assigned\_oi$ do
        if $|\{oi | oi \in d.committed\_ois \wedge oi \neq \infty\}| \geq$ n − f then
            $d.AOI := sorted(d.committed\_ois)[$f + 1$]$
            $txns\_w\_assigned\_oi.add(d)$
            $txns\_wo\_assigned\_oi.remove(d)$
        else
            for $i := 1, 2, ...,$ n do
                $lp\_ois[i] := min(d.seen\_ois[i], highest\_oi\_list[i])$
            $d.LPAOI := sorted(lp\_ois)[$f + 1$]$
            $LPAOI_{min} := min(LPAOI_{min}, d.LPAOI)$
    for $d \in sorted\_by\_aoi(txns\_w\_assigned\_oi)$ do
        if $d.AOI < LPAOI_{min}$ then
            execute the transaction of $d$
            $order\_txns.add(d)$
            $txns\_w\_assigned\_oi.remove(d)$
        else
            break
```

Figure 7: FairDAG-AB Algorithm.

Calculating Assigned Ordering Indicator

After that, R calculates *AOI* for transactions in *txns_wo_assigned_oi* that have at least $\mathbf{n} - \mathbf{f}$ non-∞ *committed_ois* values. The *AOI* of d is the $(\mathbf{f}+1)$-th lowest value of its *committed_ois*. Then, the transactions that get an *AOI* are moved from *txns_wo_assigned_oi* to *txns_w_assigned_oi*. *Note: Since the calculation described above only applies to transaction digests in txns_wo_assigned_oi. Thus, the AOI of d is fixed once calculated, even though there would be more committed ordering indicators of d.*

Example 5.1. Shown in the Figure 6, we assume that vertex $v_{4,2}$ and $v_{3,4}$ are two committed leader vertices, L_2 and L_4. The blue vertices represent D_2 and the green vertices represent D_4. The values selected as *AOI* values are marked red. Within D_2, d_1 is the only transaction digest with at least $\mathbf{n} - \mathbf{f}$ non-∞ *committed_ois* values: $\{\infty, 1, 2, 1\}$, the $(\mathbf{f}+1)$-th lowest of which is 1. Then the *AOI* of d_1 is 1. Although there is an *seen* ordering indicator 1 in $v_{1,1}$ and $v_{1,1}$ is in D_4, it is not used for calculating the *AOI* of d_1 because the value is fixed after L_2 gets committed.

Within D_4, there are 3 ordering indicators for d_2, $\{1, 2, 3\}$, then the *AOI* of d_2 is 2. Similarly, the *AOI* of d_3 is 3. For the other transactions, there are insufficient *committed_ois* values in D_4.

5.4 Global Ordering

To guarantee the *Ordering Linearizability* property, we need to ensure that the transactions are ordered and executed based on *AOI*. Thus, after getting a *AOI*, a transaction digest d cannot be ordered until it is guaranteed that no other unordered transaction has or could have a lower *AOI* than T.

Calculating Lowest Possible Assigned Ordering Indicator

To guarantee the fairness property, after calculating *AOI* for the eligible transaction digests, for the others that remain in *txns_wo_assigned_oi*, we calculate *LPAOI* for each of them and find the minimal *LPAOI* among them.

For each transaction digest d, a vector *lp_ois* is created by merging its *seen_ois* and the vector *highest_ois*, where $lp\_ois[i]$ is the $min(seen\_ois[i], highest\_ois[i])$. Then, the *LPAOI* of d is the $(\mathbf{f}+1)$-th lowest value in *lp_ois*.

After calculating *LPAOI* of all digests in *txns_wo_assigned_oi*, we pick out the minimal value $LPAOI_{min}$ and set it as a threshold. That is, only transaction digests with a *AOI* lower than $LPAOI_{min}$ can be ordered and executed. Then we sort the transaction digests based on *AOI*, order and execute them one by one and stop when an *AOI* reaches the threshold. For all successfully ordered transactions, we move them from *txns_w_assigned_oi* to *ordered_txns*.

Example 5.2. As shown in Figure 6, transaction digests d_1, d_2, d_3 have *AOI* values 1, 2, 4. With the *highest_ois* being $(2, 6, 6, 6)$ and its *seen_ois* being $(\infty, 3, 3, 5)$, d_4 has *lp_ois* being $(2, 3, 3, 5)$ and then gets its *LPAOI* value 3. Similarly, we can get the *LPAOI* values of d_5, d_6 to be 4, 5. Then the $LPAOI_{min}$ is 3. Thus, we can order and execute d_1 and d_2. And d_3 cannot be ordered because the *AOI* is higher than d_3 $LPAOI_{min}$, which implies that d_4 might get a lower *AOI* than d_3.

6 FAIRDAG-RL

FairDAG-RL is a **relative** fairness protocol. Dependency graphs are constructed with transaction digests being added as nodes. Then the fairness layer decides edge directions between nodes based on the local orderings committed in the DAG layer. Finally, Hamilton paths within the dependency graphs are found and extracted as the final transaction ordering, satisfying *γ-Batch-Order-Fairness*, $\mathbf{n} \geq \frac{\mathbf{f}(2\gamma+1)}{2\gamma-1}$, $\frac{1}{2} < \gamma \leq 1$.

6.1 Transaction Dissemination and FairDAG-RL Vertex

In FairDAG-RL, clients broadcast their transactions to all replicas, which is the same as in FairDAG-AB. Upon the fulfillment of specific conditions outlined by the DAG protocols, each replica forms and broadcasts a vertex. FairDAG-RL vertices can be viewed as a special form of FairDAG-AB vertices, where the *ois* in a FairDAG-RL vertex is a list of monotonically increasing sequence numbers. We define the *committed local ordering* of a replica R as the part of local ordering in the DAG vertices that are from R and are committed in DAG layer.

6.2 Dependency Graph Construction

Every time the DAG layer outputs a committed leader vertex L_r, the fairness layer of FairDAG-RL constructs a new dependency graph and updates the existing dependency graph with the local orderings in D_r.

First, FairDAG-RL checks if any transaction nodes are eligible for being added as graph nodes. Then, based on the local orderings, the weights between each node pair are updated, where $Weight(d_1, d_2)$ represents the number of local orderings in which d_1 appears before d_2. Finally, we add a directed edge between a pair of nodes if the weight reaches a threshold.

Each dependency graph node of transaction digest d contains the following information:

- *type* : the type of the node, which is initialized as *blank* and changed when it is added into a graph.
- *committed_ois*: vector of sequence numbers of d that R has committed. All values are initialized as ∞.
- *committed_rounds*: vector of rounds that R committed the sequence numbers. All values are initialized as ∞.
- G: the graph that the node is added into.

Each dependency graph contains the following information:

- *nodes*: a set of added transaction digest nodes.
- *edges* : a set of directed edges between the nodes.
- *weight* : mapping from node pairs to their weights.
- *compared* : mapping from node pairs to replica ids of local orderings in which two nodes have been compared.

In memory, each replica keeps the following global information:

- *ordered_nodes*: mapping from transaction digests to ordered nodes.
- *graphs*: a list of graphs ordered by the round of their corresponding leader vertices.

Adding Nodes

$$D_2 : \begin{array}{l} R_1 : \{d_0, d_1, d_2, d_5, d_3\} \\ R_2 : \{d_0, d_2, d_3, d_4, d_5\} \\ R_3 : \{d_0, d_4, d_1, d_6\} \\ R_4 : \{d_0, d_4, d_1, d_2\} \end{array} \qquad D_4 : \begin{array}{l} R_1 : \{d_4, d_6\} \\ R_2 : \{d_1, d_6\} \\ R_3 : \{d_3, d_2, d_5, d_7\} \\ R_4 : \{d_3, d_5, d_6, d_7\} \end{array}$$

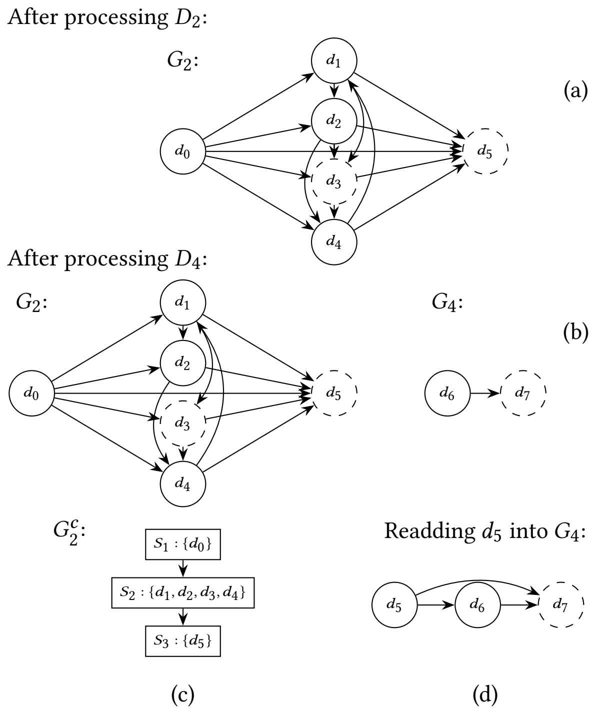

Figure 8: Constructing dependency graphs and finalizing transaction order with $\mathbf{n} = 4, \gamma = 1, \mathbf{f} = 1$.

Upon a committed leader vertex L_r is output, replica R creates a new dependency graph G_r and adds G_r to *graphs*.

For each transaction digest d contained in D_r, we denote by $node(d)$ the corresponding node of it. The *committed_ois* and *committed_rounds* of *node* are updated according to D_r if $node(d) \notin ordered\_nodes$. And we keep track of the updated nodes using a set *updated_nodes*.

We denote by $ap(d, r)$ the number of sequence numbers of d that R has committed by round r, i.e., $ap(d, r) := |\{i | node(d).committed\_rounds[i] \leq r\}|$. Next, each updated *blank* $node(d)$ is classified into different types based on $ap(d, r)$:

- *solid*, if $ap(d, r) \geq \mathbf{n} - \mathbf{f}$.
- *shaded*, if $\frac{\mathbf{n}-\mathbf{f}}{2} \leq ap(d, r) < \mathbf{n} - \mathbf{f}$.
- *blank*, if $ap(d, r) < \frac{\mathbf{n}-\mathbf{f}}{2}$.

Once *type* of $node(d)$ is changed from *blank* to *solid* or *shaded*, then $node(d)$ is added into G_r. By conducting classification to only nodes that were *blank*, it is guaranteed that each node is added into at most one dependency graph.

Updating Weights between Nodes

```
In-memory Variables
ordered_nodes := {}
graphs := []
current_round, replica_id

Client Thread (processing transactions sent from clients) :
event Upon receiving a transaction $T$ do
    if T is valid then
        $oi := GetNextSeqNum()$
        $ois.push(oi)$
        $dgs.push(T.digest)$

DAG Layer Thread (forming the DAG) :
event Upon $R$ is ready to propose a DAG vertex do
    $v := DAGVertex(replica\_id, current\_round)$
    $v.dgs := dgs$; $v.ois := ois$
    $dgs.clear()$; $ois.clear()$
    Reliably Broadcast $v$

event Upon committing a DAG leader vertex $L_r$ do
    Send $D_r$ to Ordering Layer
```

Figure 9: FairDAG-RL Algorithm.

After classifying and adding nodes into G_r. We update the weights between nodes based on the local orderings in D_r. We process the vertices one by one in a round-increasing order.

For each vertex $v_{i,r}$ from replica R_i in D_r, we traverse the *dgs*. For each *unordered* transaction digest d in *dgs*, we compare $node(d)$ with all other nodes in the same dependency graph G by the value of $comitted\_ois[i]$. For each other node $node(d_2)$, if it has not been compared with $node(d)$, then we mark this pair as compared for replica i, i.e., insert i into both $G.compared[(d, d_2)]$ and $G.compared[(d_2, d)]$. And the weight between d and d_2 is updated as follows:

- increase $G.weight[(d, d_2)]$ by 1 if $node(d).comitted\_ois[i] < node(d_2).comitted\_ois[i]$;
- increase $G.weight[(d_2, d)]$ by 1, otherwise.

While updating the weights, we use a set *addable_edges* to keep track of the node pairs between which an edge can be added. If any of the weight values reaches the threshold $\frac{\mathbf{n}-\mathbf{f}}{2}$, we insert the pair (d, d_2) into *addable_edges*.

Adding Edges

For each node pair (d, d_2) in *addable_edges*, if there is no edge between $node(d)$ and $node(d_2)$, we decide the edge direction by comparing $G.weight(d, d_2)$ and $G.weight(d_2, d)$:

- Add edge $e(d, d_2)$ pointing from $node(d)$ to $node(d_2)$, if $G.weight(d, d_2) \geq G.weight(d_2, d)$;
- Add edge $e(d_2, d)$ pointing from $node(d_2)$ to $node(d)$, otherwise.

Figure 8 shows one example of how FairDAG-RL constructs dependency graphs. As shown in Figure 8 (a), after processing D_2, G_2 is constructed with 6 nodes. Nodes d_0, d_1, d_2 and d_4 are classified as *solid* (solid circles), while nodes d_3 and d_5 are *shaded* (dashed circles). There is an edge between each pair of nodes except between d_3 and

```
Fairness Layer Thread (Ordering transactions) :
event Upon $D_r$ is received do
    $G_r := NewGraph()$
    $graphs.Push(G_r)$
    $updated\_nodes := \{\}$
    for $v \in D_r$ do
        $i := v.replica\_id$
        for $d, oi \in v.dgs, v.ois$ do
            if $node(d) \notin ordered\_nodes$ then
                $node(d).committed\_ois[i] := oi$
                $node(d).committed\_rounds[i] := i$
                $updated\_nodes.insert(d)$

    for $d \in updated\_nodes$ do
        if $node(d).type = blank$ then
            $ap(d, r) := |\{i | node(d).committed\_rounds[i] \le r\}|$
            if $ap(d, r) \ge$ n − f then
                $node(d).type = solid$
            else if $ap(d, r) \ge \frac{n-f}{2}$ then
                $node(d).type = shaded$
            if $node(d) \ne blank$ then
                $G_r.nodes.add(node(d))$

    $addable\_edges := \{\}$
    for $v \in D_r$ do
        $i := v.replica\_id$
        for $d, oi \in v.dgs, v.ois$ do
            if $node(d) \notin ordered\_nodes$ then
                $G := node(d).G$
                $d\_oi\_i := node(d).committed\_ois[i]$
                for $node(d_2) \in G.nodes$ do
                    if $d\_oi\_i < node(d_2).committed\_ois[i]$ then
                        $G.weights[(d, d_2)] := G.weights[(d, d_2)] + 1$
                    else
                        $G.weights[(d_2, d)] := G.weights[(d_2, d)] + 1$
                    if $G.weights[(d, d_2)] \ge \frac{n-f}{2} \lor G.weights[(d_2, d)] \ge \frac{n-f}{2}$ then
                        $addable\_edges.insert((d, d_2))$

    for $(d, d_2) \in addable\_edges$ do
        $G := node(d).G$
        if $G.weights[(d, d_2)] \ge G.weights[(d_2, d)]$ then
            $G.edges.add(e(d, d_2))$
        else
            $G.edges.add(e(d_2, d))$

    OrderFinalization()
```

Figure 10: Constructing Dependency Graph in FairDAG-RL

d_5, because neither $G_2.weights[(d_3, d_5)]$ nor $G_2.weights[(d_5, d_3)]$ reaches the threshold $\frac{n-f}{2} = 1$. As shown in Figure 8 (b), after processing D_4, G_4 is constructed with 2 nodes. And the edge between $node(d_3)$ and $node(d_5)$ is added as $G_2.weights[(d_3, d_5)]$ reaches the threshold.

6.3 Ordering Finalization

A *tournament* graph is a graph such that there is an edge between each pair of nodes. After adding edges, we check if the graphs are *tournaments* in a round-increasing order. If a graph is a *tournament*, we finalize the ordering of transactions within it as follows and then check the next graph until encountering a *non-tournament* graph.

```
Fairness Layer Thread (Ordering transactions) :
function OrderFinalization() do
    while $G_r := graphs.Front()$ do
        if $G_r$ is a tournament then
            $graphs.Pop()$
            $G_r^c := Trajan\_SCC(G_r)$
            $[S_1, S_2, ..., S_s] := Topologically\_Sorted(G_r^c)$

            $last := \max\{j \mid \exists node \in S_j, node.type = solid\}$
            for $j = 1, 2, ..., last$ do
                $p_j := Hamilton\_Path(S_j)$
                Append $p_j$ to final ordering
                for $node(d) \in p_j$ do
                    $ordered\_nodes.add(node(d))$

            $G_{r'} := graphs.Front()$
            for $j = last + 1, last + 2, ..., s$ do
                for $node(d) \in S_j$ do
                    $ap(d, r') := |\{i \mid node(d).committed\_rounds[i] \le r'\}|$
                    if $ap(d, r') \ge$ n − f then
                        $node(d).type = solid$
                    else if $ap(d, r') \ge \frac{n-f}{2}$ then
                        $node(d).type = shaded$
                    $G_{r'}.nodes.add(node(d))$
                    for $node(d_2) \in G_{r'}.nodes$ do
                        Calculate $G_{r'}.weights[(d, d_2)]$
                        Calculate $G_{r'}.weights[(d_2, d)]$
                        if $G_{r'}.weights[(d, d_2)] \ge \frac{n-f}{2}$
                          $\lor G_{r'}.weights[(d_2, d)] \ge \frac{n-f}{2}$ then
                            if $G_{r'}.weights[(d, d_2)] \ge G_{r'}.weights[(d_2, d)]$
                            then
                                $G_{r'}.edges.add(e(d, d_2))$
                            else
                                $G_{r'}.edges.add(e(d_2, d))$
        else
            break
```

Figure 11: Finalizing Transaction Ordering in FairDAG-RL

Condensing Dependency Graph

In graph theory, a *strongly connected component (SCC)* is a maximal subset of nodes such that for every pair of nodes $(node_1, node_2)$ in the subset, there exists a directed path from $node_1$ to $node_2$ and a directed path from $node_2$ to $node_1$.

For a *tournament* dependency graph G_r, we condense it into G_r^c by finding all *SCCs* with *Tarjan's SCC algorithm.* Figure 8 (c) shows G_2^c the condensed graph of G_2.

Ordering Finalization

It is guaranteed that after condensation, there will be one and only one topological sorting of the *SCCs* in G_r^c, which we denote by $S_1, S_2, ..., S_s$, where S_j represents the j-th *SCC* in the topological sorting.

We denote by the S_{last} the last *SCC* that contains a *solid* node. Then, for each S_j such that $j \le last$, we find a *Hamilton path* p_j

of nodes in S_j and append p_j to the final transaction ordering. In Figure 8 (c), S_2 is S_last of G_2^c.

Readding Shaded Nodes

For the *SCCs* behind S_j, the nodes are all *shaded* nodes. We add these nodes into the next graph $G_{r'}$. For each $node(d)$, we run the following steps:

(1) Classify $node(d)$ based on $ap(d, r')$;
(2) Calculate the weights between $node(d)$ and all existing nodes in $G_{r'}$.
(3) Add edges between $node(d)$ and other nodes if the weight reaches threshold $\frac{\mathbf{n-f}}{2}$.

In Figure 8 (d), $node(d_5)$ is readded into G_4, being classified as a *solid* node. And two edges pointing from $node(d_5)$ are added.

6.4 Ordering Dependency

After illustrating how to construct dependency graphs and finalize transaction ordering in FairDAG-RL, we now discuss *ordering dependencies* in FairDAG-RL. For any transaction T_1 and T_2 with digests d_1 and d_2, either T_1 is dependent on T_2 or T_2 is dependent on T_1. As the transactions are ordered based on the edges in the dependency graphs, intuitively, if an edge $e(d_1, d_2)$ exists, then T_2 is dependent on T_1.

However, for two transactions that are in different replicas, deciding *ordering dependency* is more complicated. We denote by $weight[(d_2, d_1)]_{max}$ the maximal number of the local orderings in which T_2 is ordered before T_1. Even though $node(d_2)$ is in an earlier dependency graph, it is possible that $weight[(d_2, d_1)]_{max} < \frac{\mathbf{n-f}}{2}$, which implies that edge $e(d_2, d_1)$ cannot exist even if the two nodes are in the same dependency graph. Then, if $weight[(d_2, d_1)]_{max} < \frac{\mathbf{n-f}}{2}$, T_2 is dependent on T_1.

To endorse the transactions that are added as nodes earlier, if T_1 is in a earlier dependency graph and $weight[(d_1, d_2)]_{max} \geq \frac{\mathbf{n-f}}{2}$, which implies that edge $e(d_1, d_2)$ could exist, we also say that T_1 is also dependent on T_2.

Definition 6.1. We say that T_2 with digest d_2 is dependent on T_1 with digest d_1 if:

- $weight[(d_2, d_1)]_{max} < \frac{\mathbf{n-f}}{2}$; **or**
- edge $e(d_1, d_2)$ exists; **or**
- $weight[(d_1, d_2)]_{max} >= \frac{\mathbf{n-f}}{2}$ and $node(d_1)$ is in a earlier dependency graph.

7 COMPARISON REGARDING FAIR TRANSACTION ORDERING

In this section, we demonstrate how FairDAG-AB and FairDAG-RL outperform Pompe and Themis in limiting the adversary's manipulation of transaction ordering. We achieve this by comparing how these protocols perform under adverse conditions, such as those caused by Byzantine replicas or an asynchronous network.

7.1 Pompe and Themis Sketch

First, we briefly go through the design of Pompe and Themis that run on top of leader-based protocols like PBFT and HotStuff.

Pompe. Unlike FairDAG-AB, which first commits local orderings before determining assigned ordering indicators, Pompe calculates assigned ordering indicators before consensus. Each client transaction T is submitted to one replica, which is responsible for gathering at least $\mathbf{n} - \mathbf{f}$ local ordering indicators as proof and computing the assigned ordering indicator for T. Once the assigned ordering indicator is generated, the replica broadcasts it along with the corresponding proof.

Pompe operates in a round-based manner, where each round corresponds to a distinct, non-overlapping time slot. Periodically, the leader replica collects transactions with assigned ordering indicators that fall within the most recent time slot from at least $n - f$ replicas, along with their proofs. These transactions and proofs are then batched into a block, upon which replicas run PBFT/HotStuff consensus.

Themis. In Themis, a single leader replica is responsible for collecting local orderings from at least $\mathbf{n} - \mathbf{f}$ replicas. These local orderings are then batched into a block. Once the blocks are committed, dependency graphs are constructed.

Themis employs different threshold values compared to FairDAG-RL. Specifically, in Themis, the thresholds for classifying nodes as *solid* or *shaded*, as well as for adding an edge, are $n - 2\mathbf{f}$, $\frac{\mathbf{n-2f}}{2}$, and $\frac{\mathbf{n-2f}}{2}$, respectively. In contrast, FairDAG-RL sets these thresholds at $\mathbf{n} - \mathbf{f}$, $\frac{\mathbf{n-f}}{2}$, and $\frac{\mathbf{n-f}}{2}$, respectively.

7.2 Selectively Filtering Out Orderings

We now analyze how an adversary can manipulate transaction ordering in Pompe and Themis by selectively filtering out unfavorable local ordering. Additionally, we demonstrate how FairDAG-AB and FairDAG-RL mitigate these vulnerabilities, ensuring a more resilient and fair transaction ordering process.

In Pompe, a malicious replica can exploit the assigned ordering indicators broadcast by other replicas to selectively choose $\mathbf{n-f}$ local ordering indicators for the transactions it manages. This selective inclusion allows the adversary to manipulate the relative ordering of transactions with close assigned ordering indicators.

Furthermore, if at least $\mathbf{n} - 2\mathbf{f}$ correct replicas do not receive the assigned ordering indicator for a transaction T before a malicious leader replica collects assigned ordering indicators, the leader can exclude T from the new block. Since the leader is only required to collect from $\mathbf{n} - \mathbf{f}$ replicas, it can selectively choose $\mathbf{n} - 2\mathbf{f}$ correct replicas and $\mathbf{f}$ malicious replicas, thereby delaying the final ordering position of T.

In Themis, a malicious leader can exclude local orderings from $\mathbf{f}$ correct replicas, thereby for a transaction T there might be only $\mathbf{n} - 2\mathbf{f}$. committed local orderings containing it. Therefore, Themis has lower thresholds compared to FairDAG-AB, which increases the possibility for malicious replicas to manipulate the transaction ordering.

The aforementioned issues are effectively addressed by FairDAG-AB and FairDAG-RL. The adversary's ability to selectively exclude unfavorable local orderings is significantly weakened due to the following factors:

- The *Validity* property of DAG-based protocols ensures that local orderings proposed by correct replicas will eventually be committed.
- Local orderings are more likely to be included in the causal history of a committed leader vertex, as they are referenced by DAG vertices of higher rounds.
- The leader vertex is selected at random, ensuring that it is correct with a probability of at least $\frac{\mathbf{n}-\mathbf{f}}{\mathbf{n}} > \frac{2}{3}$.

7.3 Delaying Transaction Dissemination

In Pompe, since each client transaction is sent to a single replica, a malicious replica can execute a back-running attack by intentionally delaying its dissemination to other replicas. Consequently, the transaction receives a higher assigned ordering indicator, causing it to be placed later in the final ordering.

FairDAG-AB effectively mitigates this issue, as transactions are broadcast to all replicas, and each replica independently generates and disseminates its local ordering indicators.

7.4 Crashed Leader and Asynchronous Network

In Pompe and Themis, if the leader crashes, a recovery mechanism must be initiated to elect a new leader, introducing an additional delay of $O(\Delta)$ before the protocol can resume normal operation. Moreover, in Pompe, each round corresponds to a distinct, non-overlapping time slot. If the designated leader crashes, transactions with assigned ordering indicators falling within that time slot cannot be committed or ordered, resulting in potential transaction loss or indefinite delays. In Themis, if the leader crashes, replicas have to resend their local orderings to the new leader, resulting in additional overhead.

If the network operates under asynchronous conditions where messages can experience indefinite delays, additional overhead will be introduced, similar to the overhead incurred during leader crashes.

FairDAG-AB and FairDAG-RL address the aforementioned issues through their leaderless design, inherent to DAG-based consensus protocols, and the Reliable Broadcast Communication (RBC) mechanism, which ensures the *Validity* property even in asynchronous settings.

8 CORRECTNESS PROOF

In this section, we will prove the safety, liveness, and fairness properties of FairDAG-AB and FairDAG-RL. Derived from [15, 28, 45], in the context of FairDAG, the DAG-Layer has the following properties:

- **Agreement**: if a correct replica commits D_r of leader vertex L_r, then every other correct replica eventually commits D_r.
- **Total Ascending Order**: if a correct replica commits D_r before $D_{r'}$, then $r < r'$ and no correct replica commits $D_{r'}$ before D_r.
- **Validity**: if a correct replica broadcasts a DAG vertex v, then eventually each correct replica will commit a leader vertex L_r such that $v \in D_r$.

Combining the **Agreement** and **Total Ascending Order**, we have the following lemma:

Lemma 8.1. *If a correct replica R commits a series of leader vertices $L\_list^R = L^R_{r_j}, L^R_{r_{j+1}}, \ldots$ and every other correct replica R' will eventually commit $L\_list^{R'} = L^{R'}_{r_j}, L^{R'}_{r_{i+1}}, \ldots$, such that $\forall j,\, r_j < r_{j+1}$; $L^R_{r_j} = L^{R'}_{r_j}$; and $C^R_{r_j} = C^{R'}_{r_j}$.*

8.1 Safety

Lemma 8.2. *In* FairDAG-AB, *if a correct replica R assigns transaction T with digest d* assigned ordering indicators *$d.AOI^R$, then every other correct replica R' will eventually assign T with $d.AOI^{R'}$ such that $d.AOI^R = d.AOI^{R'}$.*

Proof. According to FairDAG-AB algorithm, we know that a correct replica deterministically calculates $d.AOI$ based on C_r, the causal history of the lowest-round leader vertex L_r such that after the replica commits L_r, there are at least $\mathbf{n} - \mathbf{f}$ *committed_ois* of d in C_r.

Then, we prove the lemma by contradiction. We assume that R calculates $d.AOI^R$ based on the causal history C^R_r of leader vertex L^R_r. If R' never assigns T with an *assigned ordering indicator*, then R' never commits C^R_r. If R' assigns T with an *assigned ordering indicator* different from T', then R' must have committed some different $C'^{R'}_r$. Both cases contradict 8.1. □

Lemma 8.3. *In* FairDAG-AB*, after D_r is committed and processed, for a transaction T with digest d that has no* assigned ordering indicator *but a* lowest possible assigned ordering indicator *$d.LPAOI_r$, if T eventually gets an* assigned ordering indicator *$d.AOI$, then $d.AOI \geq d.LPAOI_r$.*

Proof. We denote by $highest\_oi\_list_r$ and $d.seen\_ois_r$, respectively, the $highest\_oi\_list$ and $d.seen\_ois$ after D_r is committed. According to the FairDAG-AB algorithm, $d.LPAOI^r$ is the $(\mathbf{f}+1)$-th lowest value of lp_ois_r where $lp\_ois_r[i] := min(d.seen\_ois_r[i], highest\_oi\_list_r[i])$, $1 \leq i \leq \mathbf{n}$. As the DAG grows, each new $d.committed[i]$ will not be smaller than $min(d.seen\_ois_r[i], highest\_oi\_list_r[i])$. Thus, as $d.AOI$ is the $\mathbf{f}+1$-th lowest value of $d.committed\_ois[i]$, where $\forall i, d.committed\_ois[i] \geq lp\_ois_r[i]$, it holds true that $d.AOI \geq d.LPAOI_r$. □

Lemma 8.4. *In* FairDAG-AB*, for any two transactions T_1 and T_2 with digests d_1 and d_2, if $d_1.AOI < d_2.AOI$, then d_1 will be ordered before d_2 in the final ordering.*

Proof. We denote by or_j the round such that transaction digest d_j is ordered, getting an *assigned ordering indicator* lower than $LPAOI_{min}$, i.e., $d_j.AOI < LPAOI_{min}$.

Now we prove by the lemma by contradiction. Assuming that d_2 is ordered before d_1, then obviously, $or_1 \geq or_2$.

- If $or_1 = or_2$, then according to FairDAG-AB algorithm, d_1 and d_2 are ordered based on assigned ordering indicator. Thus, with a lower AOI, d_1 would be ordered before d_2, contradicting with our assumption.
- If $or_1 > or_2$, then by the definition of $LPAOI_{min}$ we know that $d_2.AOI < LPAOI_{min} \leq d_1.LPAOI_{or_2}$. Also, from Lemma 8.3 we know that eventually d_1 will get $d_1.AOI \geq d_1.LPAOI_{or_2}$. Thus, we have $d_1.AOI > d_2.AOI$, which contradicts the fact that $d_1.AOI < d_2.AOI$.

In summary, it holds true that if $d_1.AOI < d_2.AOI$, then d_1 will be ordered before d_2 in the final ordering. □

Theorem 8.5. *(FairDAG-AB SAFETY) In FairDAG-AB, if a correct replica orders transaction T at position p in the final ordering, then every correct replica will eventually order T at position p.*

Proof. From Lemma 8.2 we know that every correct replica will eventually assign the same *assigned ordering indicator* to T. From Lemma 8.4 we know that all transactions with *assigned ordering indicators* are ordered in an ascending order of *AOI*. Combining the two claims above, we conclude that every correct replica will eventually order T at the same position in the final ordering. □

Theorem 8.6. *(FairDAG-RL SAFETY) In FairDAG-RL, if a correct replica orders transaction T at position p in the final ordering, then every correct replica will eventually order T at position p.*

Proof. After committing a leader vertex, each correct replica uses a deterministic method to construct dependency graphs and finalize transaction order. Combining this with Lemma 8.1, we know that safety holds for FairDAG-RL. □

8.2 Liveness

We claim the following assumption holds, which is necessary for the liveness property.

Assumption. If a correct replica R receives transaction T, then every correct replica will eventually receive transaction T.

Lemma 8.7. *In FairDAG-AB, if a transaction T with transaction d is received by correct replicas, then T will eventually get an* assigned ordering indicator.

From Assumption 8.2 we know that all correct replicas will receive T will propose a DAG vertex containing d and a corresponding ordering indicator. From the **Validity** of DAG layer, we know that all these DAG vertices will be committed. Thus, d will get at least $\mathbf{n} - \mathbf{f}$ *committed_ois* from correct replicas and then get an *assigned ordering indicator.*

Lemma 8.8. *For transactions T_1 and T_2 with digests d_1 and d_2, if only T_1 has an* assigned ordering indicator *and T_2 has an LPAOI lower than $d_1.AOI$, i.e., $d_2.AOI = \infty \wedge d_2.LPAOI < d_1$, then eventually,*

- *either T_2 gets an* assigned ordering indicator*;*
- *or $d_2.LPAOI$ becomes larger than $d_1.AOI$.*

Proof. If T_2 is received by correct replicas, from Lemma 8.7, we know that eventually T_2 gets an *assigned ordering indicator.*

Next, we discuss the case that T_2 is received by only faulty replicas. Since $d_2.LPAOI$ is the $(\mathbf{f} + 1)$-th lowest value of *lp_ois*, $\mathbf{n} - \mathbf{f}$ values of which are the *highest_oi_list* values from correct replicas, thus, $d_2.LPAOI$ is not greater than *highest_oi_list* value from at least one correct replica. Due to the **Valifity** property of the DAG layer, DAG keeps growing and the *highest_oi_list* value from each correct replica will eventually be higher than $d_1.AOI$. □

Theorem 8.9. *(FairDAG-AB Liveness) If a transaction T with digest d is received by correct replicas, then T will eventually be ordered.*

Proof. From Lemma 8.7 we know that T will eventually get an $d.AOI$. From Lemma 8.8 we know that eventually there will be no other transaction that has an *LPAOI* lower than $d.AOI$. Thus, eventually it will be satisfied that $d_1.AOI < LPAOI_{min}$, and then T will be ordered. □

Lemma 8.10. *In FairDAG-RL, each dependency graph G will eventually become a tournament.*

Proof. Since only *solid* and *shaded* nodes can be added into a dependency graph, for each $node(d)$ in G, there are at least $\frac{\mathbf{n}-\mathbf{f}}{2} > \mathbf{f} + 1$ local orderings that contain d. Thus, d will be received by all correct replicas. And then eventually, due to the **Validity** property of the DAG layer, there will be a round r such that $ap(d, r) \geq \mathbf{n} - \mathbf{f}$.

Thus, for each pair of nodes $node(d_1)$ and $node_2$ in G, there will be a round r such that $ap(d_1, r) \geq \mathbf{n} - \mathbf{f}$ and $ap(d_2, r) \geq \mathbf{n} - \mathbf{f}$. Thus, at least one of $G.weights[(d_1, d_2)]$ and $G.weights[(d_2, d_1)]$ will reach the threshold $\frac{\mathbf{n}-\mathbf{f}}{2}$. Then, eventually, there will be an edge between each pair of nodes in G, i.e., G will be a tournament. □

Theorem 8.11. *(FairDAG-RL Liveness) If a transaction T with digest d is received by correct replicas, then T will eventually be ordered.*

Proof. Due to the **Validity** property of the DAG layer, there will eventually be a round r such that $ap(d, r) \geq \mathbf{n} - \mathbf{f}$, and then $node(d)$ will be added into a dependency graph G.

From Lemma 8.10 we know that G will eventually be a tournament. If $node(d)$ is *solid*, the T will be ordered. If $node(d)$ is *shaded*, the T might be ordered. Even if *shaded* $node(d)$ has to be added into the next dependency graph, eventually $node(d)$ will be added as a *solid* node and T will be ordered. □

8.3 FairDAB-AB: Ordering Linearizability

To prove ordering linearizability, we introduce the following denotations:

- $d.\_ois^C$: the ordering indicators of d from correct replicas.
- $d.low\_oi^C$: the lowest value in $d.\_ois^C$.
- $d.high\_oi^C$: the highest value in $d.\_ois^C$.

Lemma 8.12. *For each transaction T with digest d, $d.low\_oi^C \leq d.AOI \leq d.high\_oi^C$.*

Proof. As the $d.AOI$ is the $(\mathbf{f} + 1)$-th lowest value of a subset of $d.committed\_ois$ with at least $\mathbf{n} - \mathbf{f} \geq 2\mathbf{f} + 1$ values. Among the $\mathbf{f} + 1$ lowest values of the subset, at least one is in $d.\_ois^C$. Thus, $d.low\_oi^C \leq d.AOI$. Similarly, among the $\mathbf{f} + 1$ highest values of the subset, at least one is from a correct replica. Thus, $d.AOI \leq d.high\_oi^C$. □

Theorem 8.13. *(Ordering Linearizability) For two transactions T_1 and T_2 with digests d_1 and d_2, if $d_1.high\_oi^C < d_2.low\_oi^C$, then T_1 will be ordered before T_2 in the final ordering.*

Proof. From Lemma 8.12 we know that $d_1.AOI \leq d_1.high\_oi^C$ and $d_2.low\_oi^C \leq d_2.AOI$. Thus, $d_1.AOI < d_2.AOI$. From Lemma 8.4 we know that transactions are ordered based on *AOI* values. Thus, T_1 will be ordered before T_2 in the final ordering. □

8.4 FairDAG-RL: γ-Batch-Order-Fairness

Lemma 8.14. *For any two transactions T_1 and T_2 with digests d_1 and d_2, if $\gamma(\mathbf{n}-\mathbf{f})$ correct replicas receive T_1 before T_2, then $\frac{\mathbf{n}-\mathbf{f}}{2} > weight[(d_2, d_1)]_{max}$.*

Proof. As $\gamma(\mathbf{n}-\mathbf{f})$ correct replicas receive T_1 before T_2, then $weight[(d_2, d_1)]_{max} \geq \mathbf{f}+(1-\gamma)(\mathbf{n}-\mathbf{f})$. As $\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{(2\gamma-1)}$, $\frac{1}{2} < \gamma \leq 1$, we have:

$$\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{(2\gamma-1)} \iff (2\gamma-1)(\mathbf{n}-\mathbf{f}) > 2\mathbf{f} \iff$$
$$(2\gamma-2)(\mathbf{n}-\mathbf{f})+\mathbf{n}-\mathbf{f} > 2\mathbf{f} \iff \mathbf{n}-\mathbf{f} > 2\mathbf{f}+(2-2\gamma)(\mathbf{n}-\mathbf{f}) \iff$$
$$\frac{\mathbf{n}-\mathbf{f}}{2} > \mathbf{f}+(1-\gamma)(\mathbf{n}-\mathbf{f}) \iff \frac{\mathbf{n}-\mathbf{f}}{2} > weight[(d_2, d_1)]_{max}$$

□

Combing Lemma 8.14 with the definition of *ordering dependency*, we have:

Lemma 8.15. *For any two transactions T_1 and T_2 with digests d_1 and d_2, if $\gamma(\mathbf{n}-\mathbf{f})$ correct replicas receive T_1 before T_2, then T_2 is dependent on T_1, i.e., $T_1 \rightarrow T_2$.*

Lemma 8.16. *For any two transactions T_1 and T_2 with digests d_1 and d_2 in a tournament dependency graph G, if $\gamma(\mathbf{n}-\mathbf{f})$ correct replicas receive T_1 before T_2, then after condensing G and topologically sorting the* SCCs, *$node(d_1)$ is in the same or an earlier* SCC *than $node(d_2)$.*

Proof. According to graph theory, we know that a tournament dependency graph G can be condensed into a graph with multiple *SCCs*, and after topologically sorting the *SCCs*, there is a unique list $S_1, S_2, ..., S_s$. It holds that for any two *SCCs* S_a and S_b, if $a < b$, then $\forall node(d_a) \in S_a, \forall node(d_b) \in S_b$, edge $e(d_a, d_b)$ exists in G.

We prove the lemma by contradiction. Assuming that $node(d_2)$ is in an earlier *SCC* than $node(d_1)$, then $e(d_2, d_1)$ exists in G. However, it contradicts Lemma 8.14, from which we know $\frac{\mathbf{n}-\mathbf{f}}{2} > weight[(d_2, d_1)]_{max}$, i.e., $e(d_2, d_1)$ cannot exist. □

Lemma 8.17. *$\forall node(d_1) \in G_{r_1}, \forall node(d_1) \in G_{r_2}, r_1 < r_2$, if $node(d_1)$ is* solid, *then transaction T_2 is dependent on T_1.*

Proof. If $node(d_2)$ was not added as a node into G_{r_1}, then $ap(d_2, r_1) < \frac{\mathbf{n}-\mathbf{f}}{2}$ when $ap(d_1, r_1) \geq \mathbf{n}-\mathbf{f}$. Thus, there are more than $\frac{\mathbf{n}-\mathbf{f}}{2}$ committed local orderings in which T_1 is before T_2. Then $weight[(d_1, d_2)]_{max} \geq \frac{\mathbf{n}-\mathbf{f}}{2}$, then, combining with the fact that $node(d_1)$ is in an earlier dependency graph, T_2 is dependent on T_1.

If $node(d_2)$ was added as a node into G_{r_1} but readded into G_{r_1}, then it implies that $node(d_2)$ was in an *SCC* later than the last *SCC* that contains a *solid* node. Thus, solid $node(d_1)$ has an edge to $node(d_2)$, i.e., $e(d_1, d_2)$ exists, and then T_2 is dependent on T_1. □

Theorem 8.18. *(γ-Batch-Order-Fairness) For any two transactions T_1 and T_2 with digests d_1 and d_2, if $\gamma(\mathbf{n}-\mathbf{f})$ correct replicas receive T_1 before T_2, then T_1 will be ordered no later than T_2.*

Proof. We denote by G_{r_1} and G_{r_2} in which T_1 and T_2 are ordered, respectively. If $r_1 < r_2$, then obviously T_1 is ordered before T_2 in the final transaction ordering. If $r_1 = r_2$, then from Lemma 8.17 we know that $node(d_1)$ is in the same or an earlier *SCC* than $node(d_2)$. Thus, T_1 is ordered no later than T_2 in the final transaction ordering.

If $r_1 > r_2$, then $node(d_2)$ is *shaded*. Otherwise, assuming $node(d_2)$ is *solid*, then there are at least $\mathbf{n}-\mathbf{f}$ *committed local ordering* containing d_2 when $node(d_2)$ is added into G_{r_2}. From Lemma 8.14 we know that $G_{r_2}.weights[(d_2, d_1)] < \frac{\mathbf{n}-\mathbf{f}}{2}$, then $G_{r_2}.weights[(d_1, d_2)] > \frac{\mathbf{n}-\mathbf{f}}{2}$, and edge $e(d_1, d_2)$ exists. Therefore, $node(d_1)$ should be ordered in G_{r_2} as it has a path to a *solid* node, contradicting the fact that $r_1 > r_2$. Thus, $node(d_2)$ must be *shaded* in G_{r_2}. Hence, there is some *solid* node $node(d_s)$ such that there is a path p_2 from $node(d_2)$ to $node(d_s)$. We denote by p_1 the path in G_{r_1} from the first ordered node to $node(d_1)$. Combining the following information:

- from Lemma 8.17 we know that all transactions on p_1, including $T-1$, are dependent on transaction T_s of $node(d_s)$;
- from Lemma 8.15 we know that T_2 is dependent on T_1.
- along path p_2, each transaction is dependent on the previous one, from T_s to T_2.

Thus, the transactions on p_1 and p_2 form a *cyclic dependent batch* b. For the transactions that are ordered in G_{r_2} later than T_s, they are in the same *SCC* as T_s and then can be added into b. Therefore, even if $r_1 > r_2$, T_1 and T_2 are in the same *cyclic dependent batch* in the final ordering, i.e., T_1 is ordered *no later* than T_2. □

9 EVALUATION

This section evaluates FairDAG-AB and FairDAG-RL by comparing their performance with other baseline protocols. We implement the protocols, including the baseline comparisons, using Apache ResilientDB (Incubating) [1, 20, 21]. Apache ResilientDB is an open-source incubating blockchain project that supports various consensus protocols. It provides a fair comparison of each protocol by offering a high-performance framework. Researchers can focus solely on their protocols without considering system structures such as the network and thread models. We set up our experiments on CloudLAB m510 machines with 64 vCPU and 64GB of DDR3 memory. Each replica and client runs on a separate machine.

We compare FairDAG-AB and FairDAG-RL with the following baseline protocols:

- Pbft [12]: A single-proposer consensus protocol that lacks fairness guarantees, $\mathbf{n} \geq 3\mathbf{f}+1$.
- Pompe [52]: an absolute fairness protocol running on top of Pbft, $\mathbf{n} \geq 3\mathbf{f}+1$.
- Themis [29]: a relative fairness protocol running on top of Pbft, $\mathbf{n} > \frac{\mathbf{f}(2\gamma+2)}{2\gamma-1}$.
- RCC [20]: a multi-proposer protocol that runs concurrent Pbft instances, $\mathbf{n} \geq 3\mathbf{f}+1$.
- Tusk [16], a DAG-based consensus protocol that lacks fairness guarantees in transaction ordering, $\mathbf{n} \geq 3\mathbf{f}+1$.

For Themis and FairDAG-RL, we set $\gamma = 1$ in the experiments by default. And we implement the DAG layer of FairDAG-AB and FairDAG-RL on top of a modified Tusk with weak edges.

9.1 Scalability

In the scalability experiments, we mainly measure two performance metrics:

(1) *Throughput* – the maximum number of transactions per second for which the system completes consensus and orders.

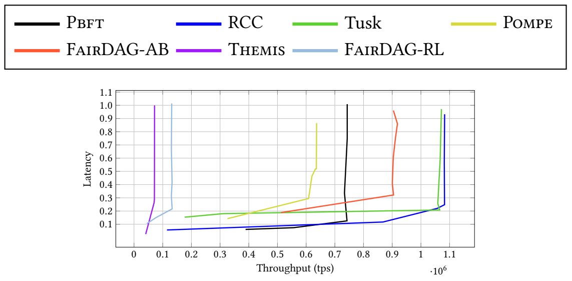

Figure 12: Throughput vs latency with f = 8.

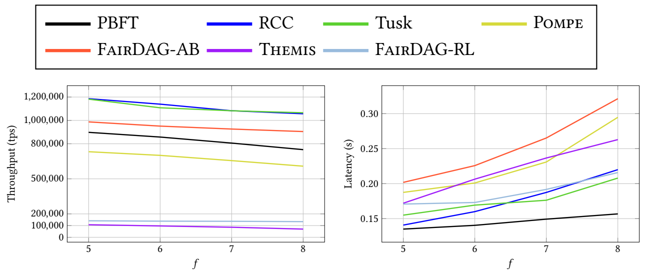

Figure 13: Performance of FairDAG-AB, FairDAG-RL and other protocols with varying f.

(2) *Client Latency* – the average duration between the time a client sends a transaction and the time the client receives **f** + 1 matching responses.

we compare the performance of different protocols with varying **f**, the maximal number of faulty replicas allowed. The value of **f** varies from 5 to 8. With the same **f**, different protocols have different replica numbers. For example, when **f** = 5, Themis has **n** = 21 replicas while other protocols have **n** = 16 replicas.

The two performance metrics are highly related to the workload that the replicas process. As shown in Figure 12 where we set **f** = 8, as the workload increases, throughput increases until the pipeline is fulfilled by the transactions. Then, after the throughput reaches the peak, latency increases as the workload increases. We define by *optimal point* the point with the lowest latency while maintaining the highest throughput. The scalability experiments are conducted at the *optimal points* of each protocol with varying **f**.

Throughput. Figure 13 shows that Tusk and RCC achieve higher throughput than other protocols because they have multiple proposers and no overhead for fairness guarantees. Due to the fairness overhead, when **f** = 5 and **f** = 8, FairDAG-AB reaches 83.5% and 84.9% throughput of Tusk, while FairDAG-RL reaches 11.9% and 12.6% throughput of Tusk.

However, compared to Pompe and Themis, the multi-proposer design of the DAG layer brings FairDAG-AB and FairDAG-RL advantages in throughput. When **f** = 5 and **f** = 8, FairDAG-AB obtains 30.2% and 52.6 higher throughput than Pompe, respectively. Similarly, FairDAG-RL reaches 43.1% and 82.3% higher throughput than Themis.

Latency. Without the fairness overhead, Tusk and Pbft, as the underlying consensus protocols, have lower latency than the fairness protocols running on top of them.

With **f** = 5 and **f** = 8, FairDAG-AB latency is 7.1% and 8.3% higher than Pompe, because Tusk, the underlying DAG consensus protocol of FairDAG-AB, has a higher commit latency than Pbft, the underlying consensus protocol of Pompe.

FairDAG-RL has a latency close to Themis when **f** = 5. As **f** grows, FairDAG-RL has a lower latency than Themis, which is 20.9% lower when **f** = 8. FairDAG-RL achieves a lower latency because Themis needs **f** more correct replicas to guarantee fairness, which causes higher overhead for both consensus and ordering. By comparing the latency of Themis with **f** = 6 and FairDAG-RL with **f** = 8, we can verify this claim. Then, with the same replica number **n** = 25, FairDAG-RL achieves a 4.6% higher latency than Themis.

9.2 Tolerance to Byzantine Behavior

In the following part, we will discuss the impact of Byzantine behaviors on the performance and final transaction ordering of the fairness protocols.

Faulty Leader Replica. In this experiment, we make the leader replica in Pbft faulty, which would trigger a view-change to replace the faulty leader. While for Tusk, as the leader vertices are randomly selected and there is no stable leader, we make a replica faulty. Figure 14 shows how the faulty leader affects the performance of Themis and Pompe. At time 7, the leader becomes faulty and stops participating in consensus. With a period without progress, Pbft timers are out within other replicas, and then a view-change is triggered to replace the faulty leader. At time 15, the view-change is complete, and the throughput of Pompe and Themis recovers to the original level. In contrast, due to the random leader vertex selection of Tusk, running on top of it, the performance of FairDAG-RL and FairDAG-AB is not affected.

Adversarial Manipulation. We conduct two experiments in which malicious replicas attempt to manipulate transaction ordering in Themis and FairDAG-RL. In the first experiment, every time a malicious replica proposes a block, it proposes the transactions in reverse order. In the second experiment, in every round, malicious replicas attempt to back-run a *victim* transaction by not including it in local orderings. In both experiments, Themis leader is malicious, excluding all local orderings from **f** correct replicas.

In the first experiment, we measure the ratio of correctly ordered pairs across different *Dist* values. For two transactions T_1 and T_2, we say that they are correctly ordered if T_1 is ordered before T_2 and $weight[(d_1, d_2)] > \frac{n}{2}$, and vice versa. The value of $Dist(T_1, T_2)$ is computed as $|weight[(d_1, d_2)] - weight[(d_2, d_1)]|$.

In the second experiment, we introduce the notion of the *victim* transaction, denoted as T_1. For each T_1, we compare its ordering only with the *bs* transactions that are closest to it in the final ordering, where *bs* is the batch size of each block. We then measure the ratio of correctly ordered pairs as a function of different $weight[(d_1, d_2)]$ values.

We conduct the experiments with **f** = 10, and vary $\mathbf{f}_a$, the actual number of malicious replicas, from 0 to 10. For example, Themis-7 denotes Themis with $\mathbf{f}_a = 7$. Since Themis and FairDAG-RL differ in their total number of replicas **n**, we normalize the x-axis values

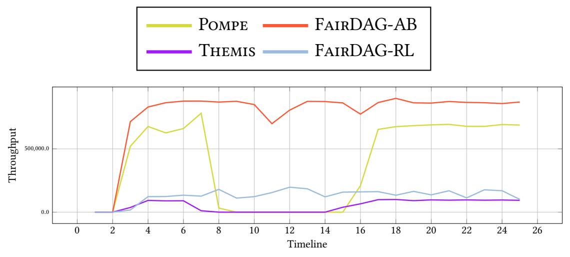

Figure 14: Real-Time Throughput with a Faulty Leader.

accordingly to ensure comparability. The results can be found in Figure 15.

As shown in Figure 15, FairDAG-RL consistently demonstrates superior resilience against adversarial ordering manipulation across all experimental settings. These results substantiate our claim that FairDAG-RL effectively mitigates selective local ordering aggregation through the properties inherent in the DAG-based consensus layer.

10 RELATED WORK

Fairness in BFT. In traditional Byzantine Fault Tolerance (BFT) research, protocols are designed to ensure both safety and liveness in the presence of malicious replicas [8, 13, 19, 22–24, 31, 35, 41]. While these protocols do not explicitly guarantee fair transaction ordering, they mitigate unfair ordering to some extent. Protocols such as HotStuff [50], which employ leader rotation in a round-robin manner [2, 17, 18, 25, 34, 47], provide each participant with the opportunity to propose a block. Leaderless approaches, including concurrent consensus protocols [20, 26, 46] and DAG-based protocols [5, 15, 28, 44, 45], enable multiple participants to propose blocks concurrently, ordering them globally through either predetermined or randomized mechanisms. Although these protocols reduce reliance on a single leader and distribute transaction ordering authority, a malicious participant can still manipulate the ordering of transactions within the blocks it proposes.

Some protocols seek to eliminate the block proposers' oligarchy over the selection and ordering of transactions within blocks by incorporating *censorship resistance* [4, 10, 36, 49]. In these protocols, transactions are encrypted and remain indecipherable until the transaction ordering is determined. However, block proposers can still engage in censorship based on metadata, such as IP addresses, or prioritize their own transactions, since they possess knowledge of the mapping from their transactions to the encrypted ciphertexts. More importantly, these censorship-resistant protocols fail to guarantee fairness, as the ordering of transactions within blocks remains fully controlled by the proposers.

Besides Pompe and Themis, there are other fairness protocols. Wendy [32] guarantees *Timed-Relative-Fairness* similar to *Ordering Linearizability*, but it relies on synchronized local clocks, which are impractical in asynchronous networks. Aequitas [30] and Quick-Order-Fairness [11] guarantee batch-order-fairness but suffer from liveness issues due to the existence of infinite *Condorcet Cycles*, which Themis solves via a *batch unspooling* mechanism. Rashnu [37]

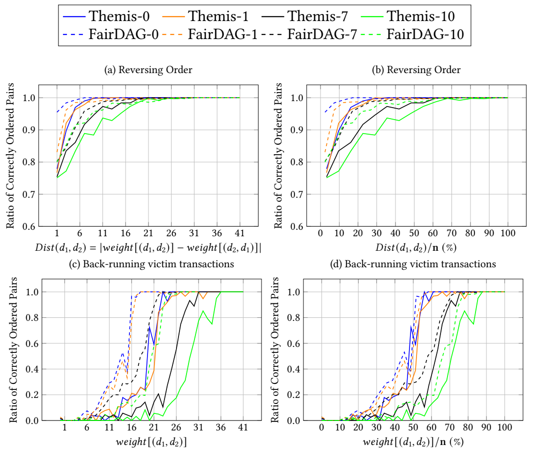

Figure 15: Fairness Quality of FairDAG-RL **and** Themis **under Byzantine attacks.**

improves Themis performance by identifying data-dependent transactions that access the same resource, guaranteeing *γ-Batch-Order-Fairness* between data-dependent transactions. However, Rashnu suffers from the same issues mentioned in Section 7 as Themis because the ordering method is unchanged.

Leaderless Protocols. A significant amount of research [3, 5, 43, 44] has been dedicated to reducing the latency in DAG-based protocols. These works employ various techniques, including *pipelining DAG waves*, *fast commit rules*, *multi-anchor*, and *uncertified DAG*, to enhance the efficiency and speed of these protocols.

Concurrent consensus protocols [20, 26, 46] represent a distinct category of leaderless consensus protocols. These protocols run multiple concurrent consensus instances independently, generating a final ordering by globally ordering the committed blocks in each instance, round by round. However, these protocols face several challenges that make them unsuitable as the underlying consensus mechanisms for fairness protocols. First, the presence of a straggler instance, which lags behind the others, can substantially decrease throughput and increase system latency [33]. Second, since the progress of different instances is not synchronized, a malicious leader of an instance could manipulate transaction ordering by delaying the block proposal until other replicas have proposed their blocks. DAG-based protocols address these issues effectively, as a replica can proceed to the next round once there are $\mathbf{n}-\mathbf{f}$ committed DAG vertices in the current round, ensuring more efficient and reliable progression.

11 CONCLUSION

In this paper, we introduced FairDAG-AB and FairDAG-RL, two fairness protocols designed to operate atop DAG-based consensus protocols. Through theoretical demonstration and experimental evaluation, we show that: unlike previous fairness protocols, FairDAG-AB and FairDAG-RL not only uphold fairness guarantees but also achieve superior performance while more effectively constraining adversarial manipulation of the final transaction ordering.

REFERENCES

[1] 2024. Apache ResilientDB (Incubating). https://resilientdb.incubator.apache.org/

[2] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. 2018. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR, abs/1803.05069* (2018).

[3] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2024. Shoal++: High throughput dag bft can be fast! *arXiv preprint arXiv:2405.20488* (2024).

[4] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. 2018. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 55–65.

[5] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *CoRR* abs/2310.14821 (2023).

[6] Paddy Baker and Omkar Godbole. 2020. Ethereum Fees Soaring to 2-Year High: Coin Metrics. *CoinDesk* (2020). https://www.coindesk.com/defi-hype-has-sent-ethereum-fees-soaring-to-2-year-high-coin-metrics

[7] Christopher Brookins. 2020. DeFi Boom Has Saved Bitcoin From Plummeting. *Forbes* (2020). https://www.forbes.com/sites/christopherbrookins/2020/07/12/defi-boom-has-saved-bitcoin-from-plummeting/

[8] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018).

[9] Vitalik Buterin. 2013. Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/.

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

[11] Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. 2022. Quick order fairness. In *International Conference on Financial Cryptography and Data Security*. Springer, 316–333.

[12] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[13] Junchao Chen, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. 2024. Thunderbolt: Causal Concurrent Consensus and Execution. *arXiv preprint arXiv:2407.09409* (2024).

[14] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2019. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234* (2019).

[15] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 34–50. https://doi.org/10.1145/3492321.3519594

[16] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3492321.3519594

[17] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International conference on financial cryptography and data security*. Springer, 296–315.

[18] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-Commit Proofs: Defeating Livelock in BFT. https://eprint.iacr.org/2021/1308

[19] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 568–580. https://doi.org/10.1109/DSN.2019.00063

[20] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1392–1403. https://doi.org/10.1109/ICDE51399.2021.00124

[21] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (2020), 868–883. https://doi.org/10.14778/3380750.3380757

[22] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2023. Dissecting bft consensus: In trusted components we trust!. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 521–539.

[23] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol. In *Proceedings of the 21st International Conference on Extending Database Technology*. Open Proceedings, 157–168. https://doi.org/10.5441/002/edbt.2018.15

[24] Jelle Hellings, Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2022. On the Correctness of Speculative Consensus. arXiv:2204.03552 [cs.DB] https://arxiv.org/abs/2204.03552

[25] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. 2025. HotStuff-1: Linear Consensus with One-Phase Speculation. *Proceedings of the ACM on Management of Data (SIGMOD)* 3, 3 (June 2025). https://doi.org/10.1145/3725308

[26] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2024. SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024*. IEEE.

[27] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.). Chapman and Hall/CRC.

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

[29] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2023. Themis: Fast, strong order-fairness in byzantine consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 475–489.

[30] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-fairness for byzantine consensus. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*. Springer, 451–480.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4 (2009), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[32] Klaus Kursawe. 2020. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 25–36.

[33] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Ivan Beschastnikh, Yinqian Zhang, Mohammad Sadoghi, and Chen Feng. 2024. Orthrus: Accelerating Multi-BFT Consensus through Concurrent Partial Ordering of Transactions. *arXiv preprint arXiv:2501.14732* (2024).

[34] Dahlia Malkhi and Kartik Nayak. 2023. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive* (2023).

[35] Tejas Mane, Xiao Li, Mohammad Sadoghi, and Mohsen Lesani. 2024. AVA: Fault-tolerant Reconfigurable Geo-Replication on Heterogeneous Clusters. *arXiv preprint arXiv:2412.01999* (2024).

[36] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 31–42.

[37] Heena Nagda, Shubhendra Pal Singhal, Mohammad Javad Amiri, and Boon Thau Loo. 2024. Rashnu: Data-Dependent Order-Fairness. *Proceedings of the VLDB Endowment* 17, 9 (2024), 2335–2348.

[38] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[39] Kaihua Qin, Liyi Zhou, Pablo Gamito, Philipp Jovanovic, and Arthur Gervais. 2021. An empirical study of defi liquidations: Incentives, risks, and instabilities. In *Proceedings of the 21st ACM Internet Measurement Conference*. 336–350.

[40] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2021. Attacking the defi ecosystem with flash loans for fun and profit. In *International conference on financial cryptography and data security*. Springer, 3–32.

[41] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2022. RingBFT: Resilient Consensus over Sharded Ring Topology. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. OpenProceedings.org, 298–311.

[42] Fabian Schär. 2021. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review* (2021).

[43] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving the Latency of DAG-based BFT. Cryptology ePrint Archive, Paper 2024/472.

[44] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT latency and robustness. *arXiv preprint arXiv:2306.03058* (2023).

[45] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 2705–2718.

[46] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. http://arxiv.org/abs/1906.05552

[47] Xiao Sui, Sisi Duan, and Haibin Zhang. 2022. Marlin: Two-Phase BFT with Linearity. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 54–66. https://doi.org/10.1109/DSN53405.2022.00018

[48] Ye Wang, Patrick Zuest, Yaxing Yao, Zhicong Lu, and Roger Wattenhofer. 2022. Impact and user perception of sandwich attacks in the defi ecosystem. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–15.

[49] Shaokang Xie, Dakai Kang, Hanzheng Lyu, Jianyu Niu, and Mohammad Sadoghi. 2025. Fides: Scalable Censorship-Resistant DAG Consensus via Trusted Components. *arXiv preprint arXiv:2501.01062* (2025).

[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 347–356. https://doi.org/10.1145/3293611.3331591

[51] Dirk A Zetzsche, Douglas W Arner, and Ross P Buckley. 2020. Decentralized finance. *Journal of Financial Regulation* 6, 2 (2020), 172–203.

[52] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 633–649.