Order-Fairness for Byzantine Consensus

Mahimna Kelkar* Fan Zhang Steven Goldfeder Ari Juels

Cornell University, Cornell Tech, and IC3

February 27, 2020

Abstract

Decades of research in both cryptography and distributed systems has extensively studied the problem of state machine replication, also known as Byzantine consensus. A consensus protocol must satisfy two properties: *consistency* and *liveness*. These properties ensure that honest participating nodes agree on the same log and dictate when fresh transactions get added. They fail, however, to ensure against adversarial manipulation of the actual *ordering* of transactions in the log. Indeed, in leader-based protocols (almost all protocols used today), malicious leaders can directly choose the final transaction ordering.

To rectify this problem, we propose a third consensus property: *transaction order-fairness*. We initiate the first formal investigation of order-fairness and explain its fundamental importance. We provide several natural definitions for order-fairness and analyze the assumptions necessary to realize them.

We also propose a new class of consensus protocols called Aequitas[1]. Aequitas protocols are the first to achieve order-fairness in addition to consistency and liveness. They can be realized in a black-box way using existing broadcast and agreement primitives (or indeed using any consensus protocol), and work in both synchronous and asynchronous network models.

*Corresponding Author: mahimna@cs.cornell.edu

[1]Aequitas (IPA pronunciation: /ˈae̯.kʷi.taːs/) is the Roman personification of fairness.

Contents

1 Introduction **3**
1.1 Our Contributions . . . 4
1.2 Related Work . . . 7

2 Definitions, Framework, and Preliminaries **9**
2.1 Protocol Execution Model . . . 9
2.2 Execution Environments . . . 11
2.3 The State Machine Replication Abstraction . . . 13

3 Building Blocks **15**
3.1 Set Byzantine Agreement . . . 15
3.2 FIFO Broadcast . . . 17

4 Defining Fair Ordering **18**
4.1 Condorcet paradox and the impossibility of fair ordering. . . . 19
4.2 Environments that support receive order-fairness . . . 20
4.3 Towards weaker definitions for order-fairness . . . 20

5 Overview of the Aequitas protocols **21**
5.1 The Finalization Stage . . . 25

6 The Synchronous Aequitas protocol **26**
6.1 Protocol Description . . . 27
6.2 Protocol Pseudocode . . . 28
6.3 Consistency Proof . . . 30
6.4 Liveness Proof . . . 31
6.5 Block Order-Fairness Proof . . . 31

7 The Asynchronous Aequitas protocol **31**
7.1 Protocol Pseudocode . . . 32
7.2 Consistency Proof . . . 33
7.3 Liveness Proof . . . 34
7.4 Block Order-Fairness Proof . . . 34

8 Other results **34**
8.1 Leader-Based Aequitas Protocols . . . 34
8.2 Adding Order-Fairness to Any Consensus Protocol . . . 35
8.3 Send Order-Fairness . . . 35

1 Introduction

The abstraction of state machine replication has been investigated in cryptography and distributed systems literature for the past three decades. At a high level, the goal of a state machine replication protocol is for a set of nodes to agree on an ever-growing, linearly ordered log of messages (transactions). Two properties need to be satisfied by such a protocol: (1) *Consistency* - all honest nodes must have the same view of the agreed upon log — that is, they must output messages in the same order; and (2) *Liveness* - messages submitted by clients are added to the log within a reasonable amount of time. In this paper, we will use the terms *state machine replication* and *consensus*[2] interchangeably.

Unfortunately, neither consistency nor liveness says anything about the actual ordering of transactions in the final log. A protocol that ensures that all nodes agree on the same ordering is deemed consistent regardless of how the ordering is generated. This leaves room for the definition to be satisfied even if an adversary directly chooses the actual transaction ordering, which is discomforting considering that the ordering is often easy to manipulate [10]. Moreover, in all existing protocols that rely on a designated "leader" node (e.g., [16, 35, 45]), which includes most used in practice, an adversarial leader may choose to propose transactions in any order.

In this paper, we formulate a new property for byzantine consensus which we call *order-fairness*. Intuitively, order-fairness denotes the notion that if a large number of nodes receive a transaction tx_1 before another one tx_2, then this should somehow be reflected in the final ordering.

Importance of fair transaction ordering. The need for a notion of fair transaction ordering is immediately clear when looking at financial systems. Here, the execution order can determine the validity and/or profitability of a given transaction. Suppose Bob has $0, and two transactions are initiated: tx_0, which sends $5 from Alice to Bob, and tx_1, which sends $5 from Bob to Carol. If tx_0 is sequenced before tx_1, then both transactions are valid; the opposite ordering invalidates tx_1. Manipulation of transaction ordering is a well known phenomenon on Wall Street [33], but recent work has shown it to also be commonplace in consensus-based systems such as permissionless blockchains. A recent paper by Daian et al. [21], for example, reports rampant adversarial manipulation of transactions in the Ethereum network [24] by bots extracting upwards of USD 6M in revenue from unsophisticated users.

Comparison to validity in Byzantine agreement. Beyond its critical practical importance, we believe that order-fairness is a key missing theoretical concept in existing consensus literature. To underscore this point, consider Byzantine agreement [31], or *single-shot* agreement, another well-studied problem in consensus literature. For Byzantine agreement, each node starts with a single value within a set $\mathcal{V}$. The goal is for all nodes to agree on the same value. *Validity* requires that if all honest nodes start with the same value v, then the agreed upon value should also be v.

The property of *order-fairness* is a natural analog of validity formulated for the consensus problem, i.e., extension of Byzantine agreement to multiple rounds. If all honest nodes start with the belief that a transaction tx_1 precedes another transaction tx_2, by natural analogy with validity, the final output log should sequence tx_1 before tx_2. Consequently, we maintain that *order-fairness* is a natural property of independent theoretical interest in the consensus literature.

[2]The term "consensus" has been used in systems literature for a number of related primitives, including "single-shot" consensus. However, in this paper, we use "consensus" to refer to the problem of "state machine replication."

1.1 Our Contributions

The main contributions of our paper are three-fold: (1) First, we investigate a natural notion of fair transaction ordering and show why it is impossible to realize. (2) Second, we investigate slightly weaker notions of fair ordering that are intuitive yet achievable. Still, we find that no existing consensus protocol achieves them. (3) Third, we introduce a new class of consensus protocols that we refer to as Aequitas. Aequitas protocols achieve fair transaction ordering while also providing the usual consistency and liveness. We discuss Aequitas protocols in both synchronous and asynchronous settings.

Defining order-fairness and impossibility results. To model the consensus protocol, we use an approach similar to prior work by Pass et al. [40, 41], wherein protocol nodes *receive* transactions from clients and need to *output* or *deliver* them in a way that satisfies consistency and liveness. We detail our model in Section 2. In this model, we provide the first formalization of the property of order-fairness. We start with a natural definition based on when transactions are received by nodes.

Definition 1.1 (Receive Order-Fairness, informal; formalized in Definition 4.1)**.** If sufficiently many (at least γ-fraction) nodes receive a transaction tx before another transaction tx', then all honest nodes must output tx before tx'.

While Definition 1.1 is intuitive, it turns out that it is impossible to achieve unless we assume very strong synchrony properties and/or a non-corrupting adversary. This result draws from a surprising connection with voter preferences in social choice theory. To highlight this using a simple example, consider three nodes, A, B, and C, that each receive 3 transactions, x, y, and z. A receives them in the order $[x, y, z]$, B in the order $[y, z, x]$ and C in the order $[z, y, x]$. Notice that a majority of nodes have received (x before y), (y before z) and (z before x)! This scenario, often called the Condorcet paradox [19], can cause a non-transitive global ordering *even when* all local orderings are transitive. This is problematic for the notion of receive order-fairness. Theorem 1.2 gives an informal description of our impossibility result.

Theorem 1.2 (Impossibility of receive order-fairness, informal; formalized in Theorem 4.4)**.** *Consider a system with n nodes where the external network (between users and protocol nodes) is either asynchronous or the maximum delay δ is at least n rounds. Then, no protocol can achieve all of consistency, liveness, and receive order-fairness.*

Given this impossibility result, we consider a natural relaxation of receive order-fairness that we call *block order-fairness.* To see the primary difference between the two definitions, we look at two transactions, tx and tx', where sufficiently many nodes have received tx before tx'. While receive order-fairness requires that tx be output "*before*" tx', block order-fairness relaxes this to "*before or at the same time as.*" We refer to transactions delivered at the same time as being in the same "block."

Definition 1.3 (Block Order-Fairness, informal; formalized in Definition 4.7)**.** If sufficiently many nodes (at least γ-fraction) receive a transaction tx before another transaction tx', then no honest node can deliver tx in a block after tx'.

This small relaxation allows us to evade the Condorcet paradox by a simple trick: placing paradoxical orderings into the same "block." We emphasize that block order-fairness does not mean that transactions are partially ordered. Consistency still requires that all nodes output transactions in the same order (within the same block or not). The only difference is that unfair ordering of a set of transactions in our definition without blocks is now, with the use of blocks, considered fair, provided that these transactions appear in the same block.

Further, we note that while receive order-fairness is impossible to achieve (as pointed out informally in Theorem 1.2 and formalized later in the paper in Theorem 4.4), block order-fairness is not and we provide protocols that guarantee it. We would also like to highlight that our proposed Aequitas protocols actually make minimal use of this relaxation. In particular, they achieve the stronger notion of receive order-fairness except when non-transitive preferences are observed.

Aequitas: Achieving order-fairness. We present a new class of consensus protocols, Aequitas, that achieve block order-fairness, in addition to providing the usual consistency and liveness properties. Aequitas protocols make use of two basic primitives in a black-box way: (1) FIFO Broadcast (FIFO-BC), which is a basic extension of standard reliable broadcast [27]; and (2) Set Byzantine Agreement (Set-BA), which we define in Section 3 and can be achieved from Byzantine agreement.

We note that these are weak primitives and any standard consensus protocol (that achieves consistency and liveness) can also be used to build the FIFO-BC and Set-BA primitives. This results in an interesting observation: The Aequitas technique provides a generic compiler that takes any standard consensus protocol and converts it into one that also provides order-fairness.

At a high level, Aequitas protocols proceed in three major stages. Each transaction tx goes through these stages before being delivered.

1. **Gossip:** Nodes gossip transactions in the order that they are received. That is, each node gossips its *local* transaction ordering.

   For this purpose, we use the FIFO broadcast primitive (FIFO-BC). FIFO-BC guarantees that broadcasts by an honest node are delivered by other honest nodes in the same order that they were broadcast. Even if the sender is dishonest, FIFO-BC guarantees that all honest nodes deliver messages in the same order. As a result, nodes have a consistent view of the transaction orderings of other nodes.

   We use Log_i^j to denote node i's view of the order in which node j received transactions, according to how j gossiped them. Note that if node j is malicious, Log_i^j may arbitrarily differ from the actual order in which j *received* transactions, but FIFO-BC prevents j from equivocating, i.e., any two honest nodes i and k will have consistent Log_i^j and Log_k^j. When i records enough logs Log_i^k that contain tx, we say that the "gossip phase" for tx is complete.

2. **Agreement:** Nodes agree on the set of nodes whose local orderings should be considered for deciding on the global ordering of a particular transaction.

   To elaborate, at the end of the gossip stage for a transaction tx, a node i ends up with a set U_i^{tx} of other nodes whose local orderings i has obtained. That is, $k \in U_i^{\text{tx}}$ if $\text{tx} \in \mathsf{Log}_i^k$. Note that different nodes may end up with a slightly different set U, but agreement proceeds when enough honest nodes are present in each set. Nodes perform Byzantine agreement to agree on a set L^{tx} of nodes whose ordering will be used to finalize the ordering for tx. For this, we define a new primitive Set-BA whose validity condition guarantees that if $k \in U_i^{\text{tx}}$ for all i, then $k \in L^{\text{tx}}$. It is easy to see how Set-BA can be realized by using standard Byzantine agreement to determine the inclusion of each possible value k individually.

Protocol	Style	Network	Corruption Bound†	Consistency	Liveness	Order-Fairness
$\Pi^{\mathsf{sync,lead}}_{\mathsf{Aequitas}}$	Leader	Synchronous*	$n > \frac{2f}{2\gamma-1}$	✓	✓	✓
$\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$	Leaderless	Synchronous*	$n > \frac{2f}{2\gamma-1}$	✓	✓ (Weak)	✓
$\Pi^{\mathsf{async,lead}}_{\mathsf{Aequitas}}$	Leader	Any	$n > \frac{4f}{2\gamma-1}$	✓	✓ (Eventual)	✓
$\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$	Leaderless	Any	$n > \frac{4f}{2\gamma-1}$	✓	✓ (Eventual, Weak)	✓

* Completely Synchronous Setting (See Section 2)

† $\frac{1}{2} < \gamma \leq 1$ is the order-fairness parameter (See Section 4)

Figure 1: The Aequitas protocols

3. **Finalization:** Nodes finalize the global ordering of a transaction tx using the set of local orderings decided on in the agreement stage.

   Suppose that the agreement stage for a transaction tx resulted in the set L^{tx}. Now, if there is any other transaction tx′ such that tx′ is ordered before tx in a large number of these local logs, it signifies that tx should be delivered after tx′. In other words, the finalization of tx depends on waiting until tx′ has been delivered.

   To characterize such dependencies between transactions, a node i maintains a directed graph G_i where vertices represent transactions and an edge from a to b denotes that b is waiting for a. Since nodes are building this graph on the same "data" (the set of local logs agreed upon in the agreement phase), nodes will have consistent graphs. That is, if an edge (a, b) exists in G_i, then it will also (eventually) exist in G_j, where i and j are both honest.

   We present two finalization techniques, a leader-based one and a leaderless one. For the leader-based technique, resolving any partial ordering within the graph G is delegated to a leader node. We emphasize that order-fairness is not lost. The leader is only able to choose the ordering for transactions that are not required to be ordered a certain way. We present another, leaderless technique that requires no further communication between nodes. We find that it realizes a slightly weaker notion of liveness than the standard one, even in a synchronous setting. Specifically, future transactions are required to be input to the system in order to "flush out" earlier transactions.

It is worth pointing out that the first two stages (gossip and agreement) are fairly straightforward to understand and easy to achieve. The third stage is somewhat complex, as it needs to avoid the Condorcet paradox while continuing to maintain both consistency and order-fairness.

Aequitas protocols. In summary, we present the first consensus protocols that provide order-fairness. We provide a leader-based and a leaderless protocol each for the synchronous and asynchronous settings, for a total of four protocols that follow the same general outline. These protocols all provide consistency, block order-fairness, and some form of liveness. Fig. 1 shows a comparison.

Paper organization. The rest of the paper is organized as follows. We discuss our results in the context of related work in Section 1.2. We describe our formal framework, along with other preliminaries, in Section 2. In Section 3, we provide the building blocks for our protocol constructions. Section 4 formally introduces our notion of order-fairness. Section 5 provides a general overview of our constructions; we detail our constructions for the synchronous and asynchronous settings in Sections 6 and 7 respectively. We describe some other interesting results in 8.

1.2 Related Work

While there is an extensive literature on consensus protocols, to the best of our knowledge, no previous work formally captures a notion of order-fairness like the one we introduce. The term "fairness" has been used widely in blockchain and cryptography literature, but for properties unrelated to ours.

Broadcast primitives. Byzantine broadcast, or the Byzantine Generals Problem [31], is the elementary broadcast primitive where a designated sender broadcasts a single value to a set of receiving nodes. In a Byzantine broadcast protocol with the key property of *consistency*, all honest receivers output the same value. Reliable broadcast is a continuous version of Byzantine broadcast where the sender broadcasts multiple values which must be eventually delivered by nodes if the sender is honest. Three orthogonal properties can be added onto reliable broadcast to give stronger notions. FIFO-ordering provides first-in first-out ordering on the messages broadcast by an honest sender. We refer to such a protocol as FIFO Broadcast or OARcast [27]. Local-ordering (also called causal-ordering) ensures that if a node broadcasts a message m' after receiving some other message m, then m will be ordered before m'. The total-ordering property ensures that all honest nodes deliver messages broadcast potentially by different senders in the same order. This notion is usually called *atomic broadcast* [20], which is well-known to be equivalent to the consensus problem. Adding all three properties to reliable broadcast results in the notion of Causal FIFO Atomic Broadcast which still does not provide the *order-fairness* property that we are looking for. The main problem is none of the requirements consider a global notion of FIFO ordering based on multiple senders.

Our order-fairness property does enforce such a notion according to the following idea: If enough nodes broadcast a message m before another message m', then honest nodes will respect this ordering. Adding this property to atomic broadcast results in a *new broadcast* notion, which we call "Global FIFO Atomic Broadcast." Consequently, requiring order fairness along with standard consensus properties of consistency and liveness will be equivalent to this new notion of Global FIFO Atomic Broadcast.

We note that our setup is also slightly different than earlier notions. We assume that any message broadcast by an honest node is also eventually broadcast by all honest nodes. This allows us to redefine liveness in terms of being broadcast by enough nodes. This also means that identical messages broadcast by different nodes can now be delivered together as a single message. Global FIFO ordering is defined on the ordering of these messages. Note that it no longer makes sense to talk about (single source) FIFO order or causal order as identical messages, potentially broadcast at different positions by different nodes, are now delivered as a single message.

Consensus protocols. Hundreds of Byzantine fault tolerant consensus protocols have been proposed over the years, with PBFT [16] being perhaps the most well known. Multiple survey papers [7, 10] have aimed to systematize this vast literature. Many papers provide efficiency improvements while maintaining the basic leader-based structure of PBFT. That is, a *leader* or *primary* node is responsible for proposing the transactions in the current round. In such leader-based protocols ([2, 3, 5, 8, 18, 35, 43–45], just to name a few), the leader node can propose transactions in the order of its choosing. The leader is also capable of suppressing transactions, at least temporarily, until an honest node becomes the new leader. We highlight that in previously explored leader-based protocols, nodes do not know the ordering in which transactions were received by everyone. This means that a leader's proposal can only be rejected based on validity of transactions rather than the fairness of their ordering. Order-fairness is thus not achieved in existing leader-based protocols.

Some protocols provide *transaction censorship resistance*, such that malicious nodes cannot censor specific transactions based on their content. For this, in protocols like [4, 11, 37], transactions are encrypted, and the contents are revealed only once their ordering is fixed. Separately, protocols like [4, 30, 32] rely on a reputation based system to detect unfair censorship. Censorship resistance is strictly weaker than the order-fairness we consider for three reasons. First, in practice, even if transaction data is temporarily encrypted, metadata such as a user identifier or a client IP address can be used to censor a particular transaction. Second, a malicious leader can still *blindly* reorder or censor transactions based on just their ciphertext. But perhaps more importantly, a malicious leader colluding with a user will know the ciphertext corresponding to the user's transaction and can thus unfairly order this transaction before others.

Other uses of the word *fairness*. The term *fairness* has been used before in consensus literature for notions unrelated to ours. One popular use case relates to *fairness in block mining* in Proof-of-Work (PoW) blockchains, which intuitively requires that a node's mining rewards be proportional to its relative computational power. That is, no node should be able to mine *selfishly* [25] to obtain more rewards than its fair share. This fairness notion is met by protocols in [1, 32, 34, 36, 38], among others.

Another related definition considers fairness in terms of the opportunities each node gets to append transactions to the ledger. This includes both fair leader election (in leader based protocols) and fair committee election (in hybrid consensus protocols). This definition is considered in [1, 26, 29, 32, 39]. We note that even if the leader election process is fair, the current leader still has the power to manipulate transaction ordering.

Fairness has also been used in the context of "fair exchange." Fair exchange protocols provide a way for mutually distrusting parties to exchange digital goods in a secure way. This notion is completely unrelated to ours but we mention it for completeness.

Works that mention fair transaction ordering. Helix [4] alludes to fair transaction ordering, but only considers censorship resistance and fair committee election. It uses threshold encryption to choose a random set of pending transactions for inclusion in the current block. Hashgraph [6] considers our notion of receive-order fairness, but provides no formal definitions. Moreover, it *fails to realize the impossibility* of this notion of fairness resulting from the Condorcet paradox [19]. As a result, we identify an elementary attack on the Hashgraph protocol that allows an adversarial node to control transaction ordering. We describe this attack at a high level below:

In the Hashgraph algorithm, each participant maintains a directed graph (called the hashgraph) of the transactions it has received from others. Participants *sync* their transactions to others by sending the hashgraph to a randomly chosen participant at every round. The intuitive strategy of their consensus protocol is to ensure that the hashgraphs maintained by honest participants are consistent. When Alice receives a "sync" of the hashgraph from Bob, she adds all of Bob's new transactions (say including a transaction tx) and any of her own to a new event node N. She then sets the new node's parents to be the last node received from Bob, and her own last node. Alice includes a timestamp with the N which is considered to be Alice's receive-time for the transaction tx. Without going into too much detail, after N has been buried sufficiently deep in the graph, Alice considers a specific set of graph nodes in her hashgraph and computes the final timestamp for tx by taking the median of all the corresponding timestamps. Each participant ends up with the same final timestamp as they compute the median on the same set of event nodes. However, we highlight that using the median to compute the final timestamp is the actual cause of unfairness since it is prone to adversarial manipulation. To see why, consider two users transactions tx_1 and tx_2 that are sent by honest users to all the protocol participants. Suppose that all nodes receive tx_1 before tx_2 and that the network adversary lets no "sync" attempts go through before everyone receives both tx_1 and tx_2. If the receive times for tx_1 and tx_2 are sufficiently intertwined, then even a single adversarial participant can cause the median timestamp for tx_1 to become larger than the median timestamp for tx_2 which breaks fair-ordering. In Section 5, we show a simple example of why timestamp based ordering protocols in general do not work.

2 Definitions, Framework, and Preliminaries

In this section, we describe the general execution framework that we will use for expressing and analyzing consensus protocols. To define the state machine replication problem in an unconstrained setting, we adopt an approach like that of Pass and Shi [40, 41] and Chan et al. [17]. We focus on the "permissioned" setting — where the number of consensus nodes n, as well as their identities, is known a priori to all participants. While arbitrary clients can send messages to these nodes, only a fixed set of nodes will take part in the consensus protocol. We are also interested in protocols for several network settings (e.g. synchronous, partially synchronous, and asynchronous) and define constrained environments for these settings by imposing restrictions that an adversary must respect.

2.1 Protocol Execution Model

Interactive Turing Machines (ITMs). To model protocol execution, we adopt the widely used Interactive Turing Machine (ITM) approach rooted in the Universal Composability framework [12]. Informally, a protocol details how nodes interact with each other where each node is represented by an Interactive Turing Machine. As standard practice in cryptography literature [12, 13, 15], we use an environment $\mathcal{Z}(1^\kappa)$ (where κ is the security parameter) to direct the protocol execution. The environment $\mathcal{Z}$ can be thought of to represent everything that is not defined by the protocol in consideration. $\mathcal{Z}$ is also responsible for activating nodes as either *honest* or *corrupt*, providing messages as inputs to nodes, and delivering messages between nodes. This is useful to model systems where protocol inputs may come from external applications and protocol outputs may be used by external applications. To communicate with others, a node sends a message to the environment, which is then relayed to other nodes as appropriate by the environment. Honest nodes follow the protocol description while corrupt nodes are assumed to be controlled by an adversary. This adversary, denoted by $\mathcal{A}$, is able to read all inputs/messages sent to corrupt nodes and can set all outputs/messages to be sent. The adversary also decides when messages sent over the network get delivered, of course subject to any network assumptions.

Rounds. We assume that the environment $\mathcal{Z}$ maintains a global clock. The clock is a global functionality [15] that contains a simple monotonic counter which can be updated adversarially by the environment. Informally, "global" means that the clock functionality exists in the system regardless of the analyzed protocol. This modeling choice follows from Canetti et al. [14]. Whether this clock is visible to protocol nodes depends on specific network settings. In synchronous settings, this clock is visible to all nodes[3]. In the synchronous setting [22], we can therefore model protocol execution in discrete time steps or rounds. At the start of each round, each node receives a set txs of transactions from the environment $\mathcal{Z}$. Transactions are assumed to be submitted by clients, but using the environment abstraction avoids having to model clients explicitly. Rather, the environment is in charge of providing transactions as input to the nodes. Furthermore, at the end of each round, each node outputs an ordered log LOG to $\mathcal{Z}$ which intuitively represents the list of transactions ordered by the node so far. We assume that $\mathcal{Z}$ always signals the start of a new round to each node.

Rounds in the partially synchronous setting [23] work similarly to the synchronous setting.

In the asynchronous setting [9], we assume that a global clock still exists in the environment. Except now, the clock is not accessible to the protocol nodes. The environment $\mathcal{Z}$ can provide user transactions and communication messages to nodes at any time. Without loss of generality, since protocol nodes cannot read the global clock, we can assume that the clock counter is incremented every time $\mathcal{Z}$ provides new transactions or delivers messages. Note that, we use the notion of rounds in an asynchronous setting merely as a tool for our analysis. It serves no purpose in the actual protocol and any protocol that works in the asynchronous setting should not rely on the current time. Throughout the paper, we may use the terms "time" and "round" interchangeably.

Notational conventions. We use κ to denote the security parameter. $\mathcal{N}$ denotes the set of protocol nodes. For a protocol Π, $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$ represents the random variable for all possible execution traces of Π w.r.t. adversary $\mathcal{A}$ and environment $\mathcal{Z}$. The possible executions arise from any randomness used by honest nodes, adversarially controlled nodes, and the environment. Any view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$ is a fully specified instance of an execution trace. That is, a particular view can be thought of as the joint view of all nodes (including all inputs, outputs, random coins etc.) during an execution. We use $\mathsf{view} \leftarrow\!\!\$\ \mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$ to denote randomly sampling an execution. $|\mathsf{view}|$ denotes the number of rounds in view.

A function $\mathsf{negl}(\cdot)$ is *negligible* if for every polynomial $p(\cdot)$, there exists a constant $\kappa_0 \in \mathbb{N}$, such that $\mathsf{negl}(\kappa) \leq \frac{1}{p(\kappa)}$ for all $\kappa \geq \kappa_0$. We use $\mathsf{negl}(\kappa)$ to denote a function that is negligible as a function of κ.

Corruption Model. Since we are concerned only with the permissioned setting, we consider environments $\mathcal{Z}$ that do not spawn any more nodes after an initial spawn. In particular, $\mathcal{Z}$ spawns a set of nodes, numbered from 1 to n without loss of generality at the start. It never spawns any additional nodes. At any point, $\mathcal{A}$ can ask $\mathcal{Z}$ to corrupt a particular node for which $\mathcal{Z}$ sends a `corrupt` signal to that node. When this happens, the internal state of that node gets exposed to $\mathcal{A}$ and $\mathcal{A}$ henceforth fully controls the node. $\mathcal{A}$ gets full control over all corrupt nodes, including the ability to control their messages and outputs.

A node is said to be *honest* in a given view if it is never under adversarial control, otherwise it is said to be *corrupt* or *byzantine*. Note that once a node is corrupted, it cannot become honest at a later point. In our general model, we assume that the adversary can corrupt nodes dynamically. That is, nodes can be corrupted at any point during the protocol's execution. We use a corruption parameter f to denote the maximum number of nodes that $\mathcal{A}$ can corrupt.

[3] In [14], it is emphasized that honest parties do not talk directly to the clock functionality. We can circumvent this restriction by having the environment send the current time counter to each node as input every round.

Communication and Network Model. As mentioned before, the environment $\mathcal{Z}$ provides transactions sent by users as inputs to nodes and also handles communication between nodes. We assume that a node can broadcast a message to any subset of recipients through an authenticated channel. Furthermore, we assume that the adversary $\mathcal{A}$ cannot modify messages sent by honest nodes but can reorder or delay messages, possibly constrained by the specific setting. We also assume the existence of a public-key infrastructure (PKI) — each node has a public key registered with the PKI. The public key of a node is given to all nodes on initialization by the environment. Note that in a PKI, digital signatures can be used to prove the identity of the message sender. Digital signatures can be realized from a PKI, as a global signing functionality $\mathcal{G}^{\Sigma}_{\text{sign}}$ (parameterized by a signature scheme Σ). We refer the reader to [41] for further details. For our paper, we can abstract away the actual implementation of signatures since in a PKI, without adding any communication overhead, we can assume that $\mathcal{Z}$ simply reveals the identity of the sender when forwarding a message to the recipient(s). We note that this is equivalent to working in the $\mathcal{G}^{\Sigma}_{\text{sign}}$-hybrid world where nodes query the global functionality $\mathcal{G}^{\Sigma}_{\text{sign}}$ when they need to sign messages.

We differentiate between two networks in our model - an *internal* network for communication between nodes and an *external* network for how external users send transactions to nodes. We emphasize that $\mathcal{A}$ is only in charge of scheduling message delivery for the internal network. The external network may reside in other parts of the application (not relevant to the consensus protocol) and is managed by $\mathcal{Z}$ (and possibly by some other network adversary). However, we may abstract specific timing properties from the external network to prove our results.

Depending on the network delay properties, we consider the synchronous setting [22] (where the network delay bound is known), the partially synchronous setting [23] (where the network delay bound is finite but unknown), and the asynchronous setting [9] (where the network delay is unbounded).

2.2 Execution Environments

Network Assumptions. First, we formally define the different network assumptions for both the external and internal networks. We assume that clients submit transactions to the system by sending them to all the nodes. As mentioned before, we do not explicitly model clients, but rather have transactions input by the environment. Any network assumptions are modeled as restrictions imposed on the environment.

External Network. The external network models the communication channel between the system users and the protocol nodes. Any assumptions on the external network can be thought of as assumptions on how the environment acts. By a synchronous external network, we mean that any transaction that is received (from the environment) by a node reaches all other nodes within a known time. This is formally defined in Definition 2.1.

Definition 2.1 (External Synchronous Setting)**.** We say that $(\mathcal{A}, \mathcal{Z})$ respects $\Delta_{\mathsf{ext}} = (\mathsf{full}, \delta)$ ext-synchrony w.r.t. protocol Π if for every $\kappa \in \mathbb{N}$ and view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, the following conditions hold: (1) $\mathcal{Z}$ provides δ as a public parameter to all nodes upon spawning; (2) If $\mathcal{Z}$ provides an input message m to a node in the txs set at time t, then at any time $t' \geq t + \delta$, all other nodes will also have received message m as input.

For the partially synchronous setting, we assume that the delay bound δ exists but is unknown to the nodes. Partial synchrony in the external network is defined similar to the synchronous setting, except now, $\mathcal{Z}$ does not provide the parameter δ to the nodes upon spawning. We use $\Delta_{\mathsf{ext}} = (\mathsf{partial}, \delta)$ to denote the partially synchronous setting. For the asynchronous setting, we only assume that transactions are not dropped by the network — they eventually get delivered to all the nodes. However, we make no assumptions on the actual delivery time. We use $\Delta_{\mathsf{ext}} = (\mathsf{none}, \infty)$ to denote an asynchronous network.

Internal Network. The internal network represents the network between nodes and is usually the standard network considered for consensus problems. For the internal network, synchrony is the assumption that any message sent by a node reaches the recipient(s) in a known, finite time δ. Definition 2.2 formalizes this synchrony assumption. The partially synchronous and asynchronous settings are defined similarly to the corresponding notions for the external network. We use $\Delta_{\mathsf{int}} = (\mathsf{partial}, \delta)$ and $\Delta_{\mathsf{int}} = (\mathsf{none}, \infty)$ to denote a partially synchronous internal network and an asynchronous internal network respectively.

Definition 2.2 (Internal Synchronous Setting)**.** We say that $(\mathcal{A}, \mathcal{Z})$ respects $\Delta_{\mathsf{int}} = (\mathsf{full}, \delta)$ int-synchrony w.r.t. protocol Π if for every $\kappa \in \mathbb{N}$ and view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, the following conditions hold: (1) $\mathcal{Z}$ provides δ as a public parameter to all nodes upon spawning; (2) If an honest node sends a message at time t, then at any time $t' \geq t + \delta$, all recipient(s) will have received the message.

Network nomenclature. We say that the network is *completely synchronous* (resp. *completely asynchronous*) if both the external and the internal network are synchronous (resp. asynchronous). We say that the internal network (resp. the external network) is *instant synchronous* if $\delta_{\mathsf{int}} = (\mathsf{full}, 0)$ (resp. $\delta_{\mathsf{ext}} = (\mathsf{full}, 0)$). We use not-async to denote both the synchronous setting (full) and the partially synchronous setting (partial).

We formalize the permissioned setting next.

Permissioned Setting. We can express the "permissioned" or "classical" environment by placing the following constraints on $(\mathcal{A}, \mathcal{Z})$: In the permissioned setting, we require that the environment $\mathcal{Z}$ spawn all nodes upfront and not spawn any new nodes during the protocol execution. Furthermore, all nodes know the identity of all other nodes in the protocol. Without loss of generality, we can assume that the initial nodes spawned by $\mathcal{Z}$ are numbered from 1 to n. We define such a permissioned environment in Definition 2.3.

Definition 2.3 (Classical Permissioned Environment)**.** We say that $(\mathcal{A}, \mathcal{Z})$ respects $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-classical execution w.r.t. a protocol Π if it respects Δ_{int} int-synchrony, Δ_{ext} ext-synchrony and for every $\kappa \in \mathbb{N}$ and view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, the following conditions hold: (1) $\mathcal{Z}$ spawns a set of nodes numbered from 1 to n at the start of the protocol and never spawns any nodes later; (2) $\mathcal{Z}$ does not corrupt more than f nodes; (3) $\mathcal{Z}$ provides all nodes the parameters (n, f) upon spawning; (4) $\mathcal{Z}$ also provides all nodes any other public parameters upon spawning. This includes the node identities as well as any public keys.

Notation. For all constraints on $(\mathcal{A}, \mathcal{Z})$, when the context is clear, we may choose to exclude which protocol we are referring to. For example, we may simply write $(\mathcal{A}, \mathcal{Z})$ respects $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-

classical execution. For the remainder of the paper, we will only consider $(\mathcal{A}, \mathcal{Z})$ that respect classical execution.

2.3 The State Machine Replication Abstraction

In the state machine replication or consensus problem, a set of nodes try to agree on a growing, linearly ordered log. At the start of each round, $\mathcal{Z}$ may provide a set txs of transactions to protocol nodes. At any time, nodes may also choose to deliver transactions by outputting a log to $\mathcal{Z}$. The LOG can be thought of as a totally ordered sequence where each element is an ordered set of transactions. We refer to the set of transactions at an index of the LOG as a "block". The LOG represents the set of transactions committed by a node so far.

Transaction nomenclatures. When discussing the trajectory of a transaction, several related terms are used in literature which may be confusing. We say that a transaction tx *received* by a node when it is given as input to the node by $\mathcal{Z}$. A transaction tx is *delivered* or *committed* by a node when it is included in a LOG output by the node to $\mathcal{Z}$.

Notation for the ordered log. Suppose that $\mathcal{T}$ denotes the space of all possible transactions. Let LOG_i represent the most recent log output by node i to the environment i.e. LOG_i represents the totally ordered list of transactions that node i has delivered so far. For two logs LOG and LOG', we define a relation $\preceq$ which intuitively signifies a "prefix" notion. $\mathsf{LOG} \preceq \mathsf{LOG}'$ stands for "LOG is a prefix of LOG'". We assume that for any x, we have $x \preceq x$ and $\emptyset \preceq x$.

$\mathsf{LOG}[p]$ denotes the p^{th} element in LOG. $\mathsf{LOG}(m)$ denotes the number p such that $\mathsf{LOG}[p]$ contains m.

The security of a state machine replication protocol can now be defined as follows:

Definition 2.4 (Security of state machine replication [41])**.** We say that a protocol Π satisfies consistency (resp. $(T_{\text{warmup}}, T_{\text{confirm}})$-liveness w.r.t. $(\mathcal{A}, \mathcal{Z})$ if there exists a negligible function $\mathsf{negl}(\cdot)$ such that for any $\kappa \in \mathbb{N}$, consistency (resp. $(T_{\text{warmup}}, T_{\text{confirm}})$-liveness) is satisfied except with $\mathsf{negl}(\kappa)$ probability over the choice of $\mathsf{view} \leftarrow\!\!\$\ \mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$ where negl is negligible in κ.

For a particular view, we define the properties as below:

- **(Consistency)** A view satisfies consistency if the following holds:
  - *Common Prefix.* If an honest node i outputs LOG to $\mathcal{Z}$ at time t and an honest node j outputs LOG' to $\mathcal{Z}$ at time t', then it holds that either $\mathsf{LOG} \preceq \mathsf{LOG}'$ or $\mathsf{LOG}' \preceq \mathsf{LOG}$.
  - *Future Self Consistency.* If a node that is honest between times t and t', outputs LOG at time t and LOG' at time $t' \geq t$ to the environment $\mathcal{Z}$, then it holds that $\mathsf{LOG} \preceq \mathsf{LOG}'$.
- **(Liveness)** A view satisfies $(T_{\text{warmup}}, T_{\text{confirm}})$-liveness if the following holds: At a time t such that $T_{\text{warmup}} < t < |\mathsf{view}|$, if an honest node either received a transaction m from $\mathcal{Z}$ or output m in its log to $\mathcal{Z}$, then for any honest node i and any time $t' \geq t + T_{\text{confirm}}; t' < |\mathsf{view}|$, it holds that m is in the log output by node i at time t'.

Here, T_{confirm} and T_{warmup} are polynomial functions in the security parameter κ, the number of nodes n, the corruption parameter f, the maximum network delay bounds as defined in Δ_{ext} and Δ_{int} (for synchronous and partially synchronous networks only), as well as the actual network delay. T_{warmup} is the protocol's warmup time, until which point liveness need not be satisfied. T_{confirm} is the maximum time it takes for a transaction (input after the warmup time) to be delivered by all honest nodes.

Note that the actual network delay is required as a parameter only for completely asynchronous networks. When the network is not asynchronous, the actual network delay is bounded by the maximum delay parameter. In such cases, the polynomials T_{confirm} and T_{warmup} can be bounded by replacing the actual network delay by the appropriate delay bound. While this is true, synchronous protocols where T_{confirm} does not depend on the maximum delay bound but rather on the actual network delay can confirm transactions much faster. The term *responsive* [39] is used to refer to such protocols.

Liveness in asynchronous networks. In the asynchronous setting, we assume that the network delay is an *unbounded* polynomial [39] in the security parameter. Equivalently, there does not exist a concrete polynomial T_{confirm} that serves as the liveness bound. Rather, we require that as long as the environment eventually delivers messages, honest nodes eventually include transactions in their logs. Note that since the environment eventually delivers all messages before the protocol execution finishes, all transactions input by the environment should be delivered by a *live* protocol.

- **(Asynchronous / Eventual Liveness)** A view satisfies $(\mathsf{none}, T_{\text{warmup}})$-eventual liveness if the following holds: At a time t such that $t > T_{\text{warmup}}$, if an honest node either received a transaction m from $\mathcal{Z}$ or output m in its log to $\mathcal{Z}$, then for any honest node i, at the end of protocol execution, it holds that m is in the log output by node i.

Weak liveness. The standard definition of liveness of a transaction tx (from Definition 2.4) is independent of what happens in the rest of the protocol's execution. Sometimes however, it is enough for a protocol to be live only if transactions continue to be received by the system. For example, a transaction tx will only be delivered if there is some transaction that is received by all nodes δ time after tx. Intuitively, later transactions will cause earlier ones to be "flushed out" of the system. We note that this subtle distinction between the two liveness definitions is rarely considered in the literature. We found that some leaderless protocols (i.e. those that are not based on a leader node) like [6, 42] implicitly ignore this distinction. In this paper however, we will call this notion "weak-liveness." Despite the subtle technical difference, we note that in any real world system, it should be as good as standard liveness. For a particular view, we define weak-liveness below.

- **(Weak Liveness)** A view satisfies $(\delta_{\mathsf{weak}}, T_{\text{warmup}}, T_{\text{confirm}})$-weak-liveness if the following holds: At a time t such that $t > T_{\text{warmup}}$, if an honest node either received a transaction m from $\mathcal{Z}$ or output m in its log to $\mathcal{Z}$ and if another transaction m' is such that it was first received by a node at time $t_{\mathsf{flush}} > t + \delta_{\mathsf{weak}}$, then for any honest node i and any time $t' \geq t_{\mathsf{flush}} + T_{\text{confirm}}; t' < |\mathsf{view}|$, it holds that m is in the log output by node i at time t'.

We also define *weak eventual liveness*, which provides a version of weak liveness for the asynchronous setting.

- **(Weak Eventual Liveness)** A view satisfies $(\mathsf{none}, T_{\text{warmup}})$-weak eventual liveness if the following holds: Suppose that at a time t such that $t > T_{\text{warmup}}$, an honest node either received a transaction m from $\mathcal{Z}$ or output m in its log to $\mathcal{Z}$. Let T be the set of transactions that were received by some node no later than when m was received. If another transaction tx' was first received by nodes after all nodes received all transactions in T, then for any honest node i, at the end of protocol execution, it holds that m is in the log output by node i.

3 Building Blocks

We start by describing some useful primitives that will form the foundation for designing our fair ordering consensus protocols. More specifically, we will utilize two primitives: (1) Set Byzantine Agreement (Set-BA); and (2) FIFO Broadcast (FIFO-BC).

Subroutines and composition. We follow the standard conventions to enable secure composition when considering multiple instantiations of the same protocol. Each instance of a protocol is spawned with a session identifier sid. We use $\Pi[\mathsf{sid}]$ to denote the instance of protocol Π with session id sid. Each protocol may take inputs from and return outputs to an environment. Note that this "environment" may be different for any subroutines called. For example, when a calling process p, forks an instance of a protocol Π, p is taken to be part of the environment for Π and handles its inputs and outputs.

3.1 Set Byzantine Agreement

Definitions. In a (poly) Set Byzantine Agreement protocol (Set-BA), participating nodes will try to agree on a set of values. At the start of the protocol, each node receives the identities of all participating nodes, the parameters n and f, the network parameters, as well as any other public parameters from $\mathcal{Z}$. Each node i in the set $\mathcal{P}$ of participating nodes also receives a set $U_i \subseteq S$ as input from $\mathcal{Z}$. The set S is also known to all nodes and its description is polynomial in κ. At the end of the protocol, each honest node $j \in \mathcal{P}$ outputs a set of the agreed upon values O_j.

Definition 3.1 (Security of Set-BA)**.** A Set-BA protocol Π_{sba} satisfies agreement, inclusion validity, and exclusion validity w.r.t. $(\mathcal{A}, \mathcal{Z})$ if for all $\kappa \in \mathbb{N}$, the following properties hold except with negligible probability over the random choice of $\mathsf{view} \leftarrow_{\$} \mathsf{EXEC}^{\Pi_{\text{sba}}}(\mathcal{A}, \mathcal{Z}, \kappa)$.

- **(Agreement)** If two honest nodes i and j output the sets O_i and O_j respectively, then $O_i = O_j$.
- **(Inclusion Validity)** If an element is in the input sets of all nodes, then it will also be in the output sets of all honest nodes. That is, if $c \in U_i$ for all $i \in \mathcal{P}$, then $c \in O_j$ for all honest j.
- **(Exclusion Validity)** If an element is not in any input set, then it is not in any honest output set. That is, if $c \notin U_i$ for all $i \in \mathcal{P}$, then $c \notin O_j$ for all honest j.

Lemma 3.2. *Consider any set Byzantine agreement protocol* Π_{sba} *that satisfies agreement, inclusion validity, and exclusion validity (w.r.t* $(\mathcal{A}, \mathcal{Z})$*). Except for a negligible number of views,* Π_{sba} *also satisfies the following:*

- **(Honest Proposal)** *If an honest node outputs the set O, then for every $c \in O$, there exists $i \in \mathcal{P}$ such that i is honest and $c \in U_i$.*

Informally, all values in the agreed upon set must have been proposed by some honest node

Proof. The proof is straightforward. We ignore the negligible "bad" views and let view be a execution of Π_{sba} where agreement, inclusion validity, and exclusion validity are all satisfied. Suppose that there was a value c in the output agreed upon by honest nodes even though it was not in any honest node's input set. Now, to an honest node, this protocol execution is indistinguishable from the world where none of the malicious nodes had c in their input set either. Equivalently, in this world, c was in the agreed upon output in Π_{sba} even when no node was given it as input by $\mathcal{Z}$. This contradicts the exclusion validity property of Π_{sba}. □

Set Agreement from Binary Byzantine Agreement (BBA**).** We show how $\mathsf{Set\text{-}BA}$ can easily be realized from a BBA protocol. Recall that in a BBA protocol, each node i starts with an initial value $b_i \in \{0, 1\}$ and outputs a bit out_i when the protocol ends. The goal is for all honest players to output the same bit. A secure BBA protocol Π_{BBA} needs to satisfy two properties in all except a negligible number of executions —

- **(Agreement)** $out_i = out_j$ for all honest nodes i and j.
- **(Validity)** If all honest nodes start with the same initial value b, then $out_i = b$ for all honest nodes i.

Let Π_{BBA} be a BBA protocol that satisfies both agreement and validity. We can now construct a protocol Π_{sba} from the BBA protocol Π_{BBA} that satisfies the $\mathsf{Set\text{-}BA}$ security properties. Suppose that Π_{sba} needs to be instantiated with the session id sid. We now describe the protocol Π_{sba} for a node i:

1. For each $s \in S$, if $s \in U_i$, node i forks a new instance of $\Pi_{\text{BBA}}[(\mathsf{sid}, s)]$ with input 1; otherwise it forks an instance $\Pi_{\text{BBA}}[(\mathsf{sid}, s)]$ with input 0.
2. Collect the outputs of all Π_{BBA} instances. Let $out(s)$ denote the output of $\Pi_{\text{BBA}}[(\mathsf{sid}, s)]$. Construct the set $O = \{s \in S \mid out(s) = 1\}$ and output it.

Lemma 3.3. *If Π_{BBA} satisfies the* BBA *security properties for $(\mathcal{A}, \mathcal{Z})$, then Π_{sba} satisfies agreement, inclusion validity, and exclusion validity.*

Proof. The proof follows in a straightforward way from the security of Π_{BBA}. One crucial point to mention is that Π_{sba} forks only a polynomial number of instances of Π_{BBA} since S is $\mathsf{poly}(\kappa)$. □

Other Properties. To analyze other useful characteristics of a $\mathsf{Set\text{-}BA}$ protocol, we define two additional properties, liveness and α-validity. Liveness describes how long it takes for nodes to reach agreement while α-validity can be used to determine how easy it is for an adversary to make honest nodes agree on a non-majority value. Formally, we say that a protocol Π_{sba} satisfies $T^{\text{sba}}_{\text{confirm}}$-liveness (respectively α_{sba}-validity) if the properties as described below are satisfied except for a negligible number of executions.

- **($T^{\text{sba}}_{\text{confirm}}$-Liveness)** All honest nodes output in at most $T^{\text{sba}}_{\text{confirm}}$ rounds after all honest nodes have input their starting value.

  When the network is asynchronous, we define liveness in the same way as for state machine replication.

- **(α_{bba}-Validity)** If c is present in the initial sets of at least α_{bba} fraction of all nodes, then $c \in O_i$ for all honest nodes i.

$T^{\text{sba}}_{\text{confirm}}$ is a polynomial in κ, n, f and the internal network delay.

3.2 FIFO Broadcast

Single source FIFO (first in, first out) broadcast (also called Ordered Authenticated Reliable broadcast or OARcast in [27]) is a broadcast primitive in which all honest nodes in the protocol need to deliver messages in the same order as they were broadcast by the sender. In one instantiation of a FIFO broadcast protocol, we consider a single designated sender who broadcasts a sequence of messages to all other nodes. If the sender is honest, each honest node must deliver the messages in the same order as they were broadcast. If the sender is dishonest, all honest nodes must deliver messages in the same order as each other; except now, this order may may be different than the one broadcast by the sender. When composing several FIFO broadcast primitives together with different senders, FIFO order is maintained for each individual sender but different honest nodes may deliver messages from different senders in different orders.

Definitions. At the start of the FIFO Broadcast ($\mathsf{FIFO\text{-}BC}$) protocol, each node receives the appropriate public parameters from the environment. At any time, the designated sender may also receive as input a message m from the environment. At any time, nodes can choose to deliver messages.

Definition 3.4 (Security of ($\mathsf{FIFO\text{-}BC}$))**.** A $\mathsf{FIFO\text{-}BC}$ protocol Π_{fifocast} satisfies liveness, agreement, and FIFO-order w.r.t. $(\mathcal{A}, \mathcal{Z})$ if for all $\kappa \in \mathbb{N}$, the following properties hold except with negligible probability over the random choice of $\mathsf{view} \leftarrow\!\$\ \mathsf{EXEC}^{\Pi_{\text{fifocast}}}(\mathcal{A}, \mathcal{Z}, \kappa)$.

- **($(T^{\text{fifocast}}_{\text{warmup}}, T^{\text{fifocast}}_{\text{confirm}})$-Liveness)** If the sender is honest and receives a message m as input in round $r > T^{\text{fifocast}}_{\text{warmup}}$, or if an honest node delivers m in round $r > T^{\text{fifocast}}_{\text{warmup}}$, then all honest nodes will have delivered m by round $r + T^{\text{fifocast}}_{\text{confirm}}$.

  Eventual liveness in asynchronous networks is defined in the same way as for state machine replication.

- **(Agreement)** If an honest node delivers a message m before m', then no honest node delivers m' unless it has already delivered m.

- **(FIFO-Order)** If the sender is honest and is input a message m before m', then no honest node delivers m' unless it has already delivered m.

$T^{\text{fifocast}}_{\text{confirm}}$ is a polynomial in κ, n, f and the internal network delay.

Notation. Let $\Pi_{\text{fifocast}}[(\mathsf{sid}, j)]$ denote the instance of the protocol Π_{fifocast} where node j is the designated sender. In a consensus protocol that invokes $\Pi_{\text{fifocast}}[(\mathsf{sid}, j)]$, we assume that each node i keeps track of the messages delivered (i.e. messages broadcast by node j) in a local log $\mathsf{Log}_i^{(\mathsf{sid},j)}$. This represents node i's view of broadcasts from node j in the session sid. When the session id is clear from context, we may also write the local log simply as Log_i^j. Two local logs Log and Log' are called "equal until tx", denoted by $\approx_{\text{tx}}$, if they are equivalent until the occurrence of tx. $\mathsf{Log}[p]$ denotes the p^{th} element in Log. $\mathsf{Log}(m)$ denotes the number p such that $\mathsf{Log}[p]$ contains m. Consequently, $\mathsf{Log}(m) < \mathsf{Log}(m')$ signifies that m appears before m' in Log.

$\mathsf{FIFO\text{-}BC}$ **from Reliable Broadcast.** Reliable broadcast is a basic broadcast primitive where a designated sender broadcasts messages to a set of nodes. Honest nodes will only deliver those messages that were broadcast, and will eventually deliver all messages broadcast by an honest sender. Reliable broadcast can be considered a "continuous" version of single shot byzantine broadcast or the byzantine generals problem [31]. Ho et al. [27] show how FIFO broadcast can be achieved using reliable broadcast even in asynchronous networks. The intuition is simple: sequence numbers are added to the messages broadcast by the sender in a reliable broadcast protocol. An honest node does not deliver a message with sequence number k until it has delivered a message with sequence number $k-1$. We refer the reader to [27] for the detailed construction.

4 Defining Fair Ordering

We formally define fair ordering in this section. As it turns out, providing a definition that is achievable by protocols, yet intuitive, is not trivial. Some natural definitions are not achievable except under strong assumptions. We use this section to also go through these definitions that led to our final definition.

(Attempt 1) – Send order-fairness. A strawman approach is to require ordering to be in terms of when transactions were *sent* by clients. For instance, if a transaction tx_1 was sent by a client before another transaction tx_2 (possibly by another client), then tx_1 should appear before tx_2 in the agreed upon log. Not surprisingly, this can lead to several problems: most importantly, there needs to be a trusted way to timestamp a transaction at the client side. We discuss the possibility of achieving it in practice using trusted hardware in Section 8.3.

(Attempt 2) – Receive order-fairness. The challenges of send order-fairness suggest it would be more prudent to define fair ordering in terms of when the consensus nodes actually *receive* transactions. Intuitively, "receive order" means that the fair ordering is defined by looking at when *enough* nodes receive a particular transaction. For instance, if sufficiently many nodes receive a transaction tx_1 before another transaction tx_2, then tx_1 must appear before tx_2 in the final log. "Sufficiently many" is parameterized using γ.

Definition 4.1 (Receive order-fairness, restatement of Definition 1.1)**.** For a view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, define receive order-fairness as follows:

- A view satisfies $(\gamma, T_{\text{warmup}})$-(receive-)order-fairness if the following holds: For any two transactions m and m', let η be the number of nodes that received both transactions between times

T_{warmup} and $|\mathsf{view}|$. If at least $\gamma\eta$ of those nodes received m before m' from $\mathcal{Z}$, then for all honest nodes i, i does not deliver m' unless it has previously delivered m.

A protocol Π satisfies $(\gamma, T_{\text{warmup}})$ (receive) order-fairness w.r.t $(\mathcal{A}, \mathcal{Z})$ if there is a negligible function $\mathsf{negl}(\cdot)$ such that for any $\kappa \in \mathbb{N}$, the order-fairness property is satisfied except with probability $\mathsf{negl}(\kappa)$ over $\mathsf{view} \leftarrow_{\$} \mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$.

4.1 Condorcet paradox and the impossibility of fair ordering.

The Condorcet paradox [19], or the "voting paradox", is a result in social choice theory that shows how some situations can lead to non-transitive collective voting preferences even if the preferences of individual voters are transitive. To illustrate how this applies to fair ordering, let us look at a simple example:

Example 4.2. Suppose that there are 3 nodes: A, B, and C. In the protocol execution 3 transactions, tx_1, tx_2, and tx_3 are sent by clients to all the nodes.

- Node A receives transactions in the order $\text{tx}_1, \text{tx}_2, \text{tx}_3$
- Node B receives transactions in the order $\text{tx}_2, \text{tx}_3, \text{tx}_1$
- Node C receives transactions in the order $\text{tx}_3, \text{tx}_1, \text{tx}_2$

Now, 2 nodes (A and C) received tx_1 before tx_2, 2 nodes (A and B) received tx_2 before tx_3, and 2 nodes (B and C) received tx_3 before tx_1. It is easy to see that no protocol can satisfy fair ordering for $\gamma \leq \frac{2}{3}$, since such a protocol would have to include tx_1 before tx_2; tx_2 before tx_3; and tx_3 before tx_1 in its final log.

Theorem 4.3 extrapolates this observation to a system with n consensus nodes.

Theorem 4.3. *Consider any $n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}}$ where Δ_{ext} is either (none, ∞) or $(\mathsf{not\text{-}async}, \delta_{\mathsf{ext}} \geq n)$. Let $\gamma \leq \frac{n-1}{n}$. If a consensus protocol Π satisfies $(T_{\text{warmup}}, T_{\text{confirm}})$-liveness w.r.t. all $(\mathcal{A}, \mathcal{Z})$ that respect $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-classical execution, then it cannot also satisfy $(\gamma, T_{\text{warmup}})$-receive-order-fairness (from Definition 4.1).*

Proof. The proof takes inspiration from the counterexample in Example 4.2. Denote the nodes in the system by the numbers 1 to n. We show a specific environment $\mathcal{Z}$ in which no protocol can achieve receive order-fairness. Suppose that clients submit n transactions tx_1 to tx_n. Further, suppose that node 1 receives the transactions in the order $\text{tx}_1, \text{tx}_2, \cdots, \text{tx}_n$ and any node $i \neq 1$ receives the transactions in the order $\text{tx}_i, \cdots, \text{tx}_n, \text{tx}_1, \cdots, \text{tx}_{i-1}$.

Now, it is straightforward to see that all nodes except node 2 received tx_1 before tx_2, all nodes except node 3 received tx_2 before tx_3 and so on. Finally, all nodes except node 1 received tx_n before tx_1. This means that any consensus protocol that provides order-fairness for $\gamma \leq \frac{n-1}{n}$ must order tx_1 before tx_2, $\cdots$, tx_{n-1} before tx_n, and tx_n before tx_1 which is a contradiction. □

Following the previous result, one would think that receive order-fairness might still be possible for $\gamma = 1$. Unfortunately, a simple followup theorem shows this to be impossible for even a single corrupt node.

Theorem 4.4. *Consider any $n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}}$ where $f \geq 1$ and where Δ_{ext} is either (none, ∞) or $(\mathsf{not\text{-}async}, \delta_{\mathsf{ext}} \geq n)$. Let $\gamma \leq 1$. If a consensus protocol Π satisfies consistency and $(T_{\text{warmup}}, T_{\text{confirm}})$ liveness w.r.t. all $(\mathcal{A}, \mathcal{Z})$ that respect $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-classical execution, then it cannot also satisfy $(\gamma, T_{\text{warmup}})$ receive order-fairness.*

Proof. The case for $\gamma < 1$ is handled by Theorem 4.3. To show the result for $\gamma = 1$, first suppose that a protocol Π satisfied consistency, $(T_{\text{warmup}}, T_{\text{confirm}})$ liveness and $(1, T_{\text{warmup}})$ receive order-fairness w.r.t all $(\mathcal{A}, \mathcal{Z})$. Consider an adversary $\mathcal{A}_0$ that corrupts a single node. Specifically $\mathcal{A}_0$ corrupts node N at the start of a protocol's execution and immediately crashes it. That is, N cannot send or receive any messages from other nodes. Now for any $\mathcal{Z}_0$, if Π satisfies $(1, T_{\text{warmup}})$ receive order-fairness w.r.t. $(\mathcal{A}_0, \mathcal{Z}_0)$, it must also satisfy $(\frac{n-1}{n}, T_{\text{warmup}})$ receive order-fairness since Π does not know the ordering of transactions received by node N. But from the proof of Theorem 4.3, we saw an environment where $\gamma = \frac{n-1}{n}$ receive order-fairness cannot be achieved. This means that Π cannot achieve $(1, T_{\text{warmup}})$ receive order-fairness while also achieving consistency and liveness in all $(\mathcal{A}, \mathcal{Z})$. □

4.2 Environments that support receive order-fairness

We find that the Condorcet paradox can be circumvented in a few ways by assuming specific network properties.

External synchrony assumption. The primary reason for the impossibility of fair-ordering is that different nodes may receive the same client transaction several rounds apart, resulting in non-transitive collective ordering. Suppose that $\Delta_{\mathsf{ext}} = (\mathsf{full}, \delta)$ where $\delta \leq 1$ (e.g. an instant synchronous external network). Then, any client transaction that a node receives will reach all other nodes within 1 round. This implies that if some node receives transactions tx_1, tx_2 and tx_3 in that order, then no node can receive tx_3 before tx_1. It is now straightforward to see how this circumvents the Condorcet paradox.

Non-corrupting adversary and $\gamma = 1$. If the adversary does not corrupt any nodes, and its power is restricted to influencing network delays, we find that it is possible to achieve receive order-fairness for $\gamma = 1$. In this setting, a single leader can receive the transaction orderings from individual nodes, and decide on a final ordering that preserves receive order-fairness.

4.3 Towards weaker definitions for order-fairness

We give two natural relaxations of the original definition. The first is *approximate receive order-fairness* (or simply *approximate order-fairness*) while the second is *block receive order-fairness* (or simply *block order-fairness*). For approximate order-fairness, we only look at unfairness in the ordering of two transactions if they were received sufficiently apart in time. We emphasize that approximate order-fairness only makes sense in synchronous and partially synchronous settings. On the other hand, for block order-fairness, we choose to ignore the ordering within a block while considering fair ordering. Notably, this allows us to circumvent the Condorcet paradox by aggregating any transactions with non-transitive orderings into the same block. This is reasonable to consider even in asynchronous environments. First, we look at approximate order-fairness. For a given view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, we define the property below.

Definition 4.5 (Approximate Order-Fairness)**.** A view satisfies $(\gamma, T_{\text{warmup}}, \xi)$-approximate order-fairness if the following holds: For any two transactions m and m', let η be the number of nodes that received both transactions between times T_{warmup} and $|\mathsf{view}|$. If at least $\gamma\eta$ of those nodes received m more than ξ rounds before m' from $\mathcal{Z}$, then for all honest nodes i, i does not deliver m', unless it has previously delivered m.

A protocol Π satisfies $(\gamma, T_{\text{warmup}}, \xi)$-approximate order-fairness w.r.t $(\mathcal{A}, \mathcal{Z})$ if there is a negligible function $\mathsf{negl}(\cdot)$ such that for any $\kappa \in \mathbb{N}$, the above property is satisfied except with probability $\mathsf{negl}(\kappa)$ over $\mathsf{view} \leftarrow\$ \mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$.

Quickly, we notice a protocol that satisfies $(T_{\text{warmup}}, T_{\text{confirm}})$-liveness, also satisfies $(1, T_{\text{warmup}}, \xi)$ approximate order-fairness for any $\xi \geq T_{\text{confirm}}$. Clearly, if a transaction tx_2 was received after tx_1 was delivered by all nodes, then tx_2 will be delivered after tx_1. Moreover, we also find that if $\xi < T_{\text{confirm}}$, then any protocol that satisfies $(\gamma, T_{\text{warmup}}, \xi)$ approximate order-fairness must also satisfy $(\gamma, T_{\text{warmup}})$ receive order-fairness.

Theorem 4.6. *Consider any $n, f \geq 1, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}}$. Let $\Delta_{\mathsf{int}} = (\mathsf{not\text{-}async}, \delta_{\mathsf{int}})$ and $\Delta_{\mathsf{ext}} = (\mathsf{not\text{-}async}, \delta_{\mathsf{ext}} \geq 1)$. Also consider $\gamma \leq 1$ and $\xi < T_{\text{confirm}}$. If a protocol Π achieves consistency, $(T_{\text{warmup}}, T_{\text{confirm}})$-liveness, and $(\gamma, T_{\text{warmup}}, \xi)$-approximate order-fairness. w.r.t. all $(\mathcal{A}, \mathcal{Z})$ that respect $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-classical execution, then it also satisfies $(\gamma, T_{\text{warmup}})$-receive-order-fairness w.r.t all $(\mathcal{A}', \mathcal{Z}')$ that respect $(n, f, \Delta'_{\mathsf{int}}, \Delta'_{\mathsf{ext}})$-classical execution where $\Delta'_{\mathsf{int}} = (\mathsf{not\text{-}async}, \delta'_{\mathsf{int}} = \frac{\delta_{\mathsf{int}}}{\xi})$ and $\Delta'_{\mathsf{ext}} = (\mathsf{not\text{-}async}, \delta'_{\mathsf{ext}} = \frac{\delta_{\mathsf{ext}}}{\xi})$.*

Consequently, approximate order-fairness doesn't turn out to be very useful since it suffers from the same problems as the previously defined receive order-fairness. Our second proposal, block order-fairness, performs much better since it provides a way to handle any cycles in transaction ordering. For a given view in the support of $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, we state the property below.

Definition 4.7 (Block Order-Fairness)**.** A view satisfies $(\gamma > \frac{1}{2}, T_{\text{warmup}})$-block-order-fairness if the following holds: For any two transactions m and m', let η be the number of nodes that received both transactions between times T_{warmup} and $|\mathsf{view}|$. If at least $\gamma\eta$ of those nodes received m before m' from $\mathcal{Z}$, then for all honest nodes i, i does not deliver m at a later index than it delivers m'.

A protocol Π satisfies $(\gamma, T_{\text{warmup}})$-block-order-fairness w.r.t $(\mathcal{A}, \mathcal{Z})$ if there is a negligible function $\mathsf{negl}(\cdot)$ such that for any $\kappa \in \mathbb{N}$, the above property is satisfied except with probability $\mathsf{negl}(\kappa)$ over $\mathsf{view} \leftarrow\$ \mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$.

5 Overview of the Aequitas protocols

We provide a general overview of our Aequitas protocols in this section. Specifically, we present four constructions. In the next two sections, we will dive deeper into the actual Aequitas constructions. Specifically, we provide four concrete protocols: $\Pi^{\mathsf{sync,lead}}_{\mathsf{Aequitas}}$, $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$, $\Pi^{\mathsf{async,lead}}_{\mathsf{Aequitas}}$ and $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$. Sections 6 and 7 describe the leaderless synchronous and asynchronous protocol designs respectively. The leader-based protocols are easier modifications to existing consensus protocols and we use Section 8.1 to discuss them.

- $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$ is a leaderless protocol that provides consistency, (weak) liveness, and block order-fairness in the completely synchronous setting.
- $\Pi^{\mathsf{sync,lead}}_{\mathsf{Aequitas}}$ is a leader-based protocol that provides consistency, liveness, and block order-fairness in the completely synchronous setting.

- $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$ is a leaderless protocol that provides consistency, eventual liveness, and block order-fairness in any setting.

- $\Pi^{\mathsf{async,lead}}_{\mathsf{Aequitas}}$ is a leader-based protocol that provides consistency, eventual liveness, and block order-fairness in any setting.

Construction overview. Aequitas protocols utilize the FIFO-broadcast (FIFO-BC) and the set byzantine agreement (Set-BA) primitives described in Section 3 in a black-box way to provide order-fairness. At a high level, Aequitas protocols function in three steps: First, a node uses a FIFO-BC protocol Π_{fifocast} to send all other nodes the transactions it has received from users. Recall that in FIFO-BC, nodes deliver messages in the same order as broadcast by an honest sender. When a node delivers a message received from another node, it gets added to its local log. To elaborate, broadcasts from node j as delivered by node i are tracked in the local log Log_i^j. Next, all nodes seek to agree on the content of these local logs so as to order the transaction tx in question. This is done using a Set-BA protocol Π_{sba}. At this point, intuitively, all honest nodes have agreed on anything that will be used to compute the ordering for tx. To decide on the final ordering for tx, we provide two options for the finalization step — a leader based one and a leaderless one.

For the finalization step in the leader-based protocol, a designated leader proposes an extension to the current chain. Since other nodes have all the relevant transaction orderings from the stages before, they can verify that the leader's proposal does not break order-fairness. If the leader's proposal is valid, nodes can deliver the proposed transactions by extending their LOG. An important difference exists between such a leader-based protocol and prior leader-based protocols: In earlier protocols, a leader could propose any ordering of its choice that would be accepted by other nodes. On the other hand, in our leader-based protocol, a malicious leader can mess with the transaction orderings only in a way that does not break the order-fairness property. For instance, if a transaction tx_1 was received before tx_2 by all nodes, a malicious proposal that puts tx_2 before tx_1 will be rejected by all the other nodes.

We propose another finalization that is leaderless and requires no further communication between nodes. It provides consistency, block order-fairness and weak liveness (from Section 2.3). Recall that "weak" denotes that liveness depends on transactions continuing to be input into the system.

We elaborate on the three major stages of our Aequitas protocols below:

- **Stage I: Gossip / Broadcast**. Each node FIFO-broadcasts transactions as they are received as input from the environment. When a node i receives a set of transactions txs from $\mathcal{Z}$, it sends txs as input to the protocol $\Pi_{\text{fifocast}}[(\mathsf{sid}, i)]$ with i as the designated sender. Note that all broadcasts can be sent in the same session sid. Different session ids need to be used only when considering composition of several protocols in the system.

  In parallel to broadcasting transactions, a node also receives and processes broadcasts from other nodes. For a node i, broadcasts sent by node j are appended to a local log Log_i^j when they get delivered to i by $\Pi_{\text{fifocast}}[(\mathsf{sid}, j)]$. Log_i^j denotes node i's view of how transactions were received by node j.

- **Stage II: Agreement on local logs**. To determine the ordering for a particular transaction tx, a node i waits until it has received tx from sufficiently many other nodes. In other words, node i waits until there are sufficiently many k such that the local log Log_i^k contains tx. When both the external and internal networks are synchronous, this can alternatively be achieved by waiting for enough *time*. The properties of FIFO-BC guarantee that if two honest nodes i and j have local logs Log_i^k and Log_j^k respectively that both contain tx, then $\mathsf{Log}_i^k \approx_{\mathrm{tx}} \mathsf{Log}_j^k$. We state this fact as Lemma 5.1. Recall that $\mathsf{Log}_i^k \approx_{\mathrm{tx}} \mathsf{Log}_j^k$ holds when Log_i^k and Log_j^k are identical until tx occurs.

Now, the next step is for all nodes to agree on which local logs to use to determine the ordering for tx. For a node i, let U_i^{tx} denote the set of nodes k such that Log_i^k contains tx. Node i starts an instance of the protocol $\Pi_{\mathrm{sba}}[(\mathsf{sid}, \mathrm{tx})]$ and provides it the input U_i^{tx}. Upon the completion of the Set-BA protocol, all honest nodes receive the same set L^{tx}. Intuitively, Set-BA is used to agree which nodes' orderings should be used to determine the final ordering for transaction tx. Recall that Lemma 3.2 guarantees that if a $k \in L^{\mathrm{tx}}$, then there is some honest node j such that $\mathrm{tx} \in \mathsf{Log}_j^k$. This, along with the liveness property for FIFO-BC ensures that all honest nodes will eventually receive tx broadcast by node $k \in L^{\mathrm{tx}}$ (even if k is malicious).

Finally, we note that at the end of the agreement phase, every honest node has agreed on a set of nodes L^{tx} whose transaction orderings should be used to determine the final ordering for the transaction tx in consideration. We say that a node i has received the agreed logs for tx if for all $k \in L^{\mathrm{tx}}$, it holds that $\mathrm{tx} \in \mathsf{Log}_i^k$.

- **Stage III: Finalization.** To decide on the final ordering for tx, we provide two options for the finalization step: a leader based one and a leaderless one. For both the leader-based and leaderless finalizations, nodes first build a graph that represents any ordering dependencies between transactions. A node i maintains a directed graph G_i where vertices represent transactions and edges represent ordering dependencies. We refer to G_i as the "dependency graph" or the "waiting graph" maintained by i. After the agreement stage for a transaction tx is completed, the informal technique is to use the local logs to see if some other transaction might have come before. If there is another transaction tx' that appears before tx in sufficiently many local logs (e.g., $n - f$ times), then i adds an edge from tx' to tx in G_i. Intuitively, an edge $(a, b) \in G_i$ denotes that the finalization of b is "waiting" for a to be delivered. Since the same L^{tx} is used by all honest nodes, if an edge (a, b) exists in G_i, then it will at some point exist in G_j, when nodes i and j are both honest. However, G_i is neither guaranteed to be complete nor acyclic. There may exist two vertices in G_i that will never have an edge between them. Moreover, the Condorcet paradox can still create cycles in G_i.

  - **Finalization via leader-based proposal.** $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,lead}}$ and $\Pi_{\mathsf{Aequitas}}^{\mathsf{async,lead}}$ both use a leader-based approach to finalize transactions in the graph. For this, any leader-based consensus protocol can be run along with the gossip and agreement stages above. When a designated leader proposes and broadcasts a new block, instead of just checking the syntactical validity of transactions, each node i checks that the proposal does not conflict with any required order-fairness in the graph G_i. That is, node i checks that for any transaction tx in the proposed block, if $(\mathrm{tx}', \mathrm{tx})$ is in G_i, then either tx' has already been delivered or tx' is also in the current proposed block.

In such a protocol, we allow the leader node to choose the transaction ordering but only as long as order-fairness is still satisfied. For transactions among which there is no clear winner, the leader may choose any ordering.

– **Finalization via local computation.** $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$ and $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$ both use a leaderless approach to finalize transactions in the graph and require no further communication. At a high level, to order transactions tx_1 and tx_2 between whom there in no edge in G_i, the protocol will wait until tx_1 and tx_2 have a common descendant, with the final ordering being based on which transaction vertex has the most descendants. We prove that any other graph vertex that is a descendant of only one of tx_1 and tx_2 is present in G_i when node i makes the decision for ordering tx_1 and tx_2. Intuitively, this will ensure that all honest nodes will order tx_1 and tx_2 the same way. We defer the details to Sections 5.1, 6 and 7.

We highlight that the above description is a simplified one. As described, it is not sufficient to avoid the Condorcet paradox. Furthermore, adversarial transactions could result in a node waiting for unbounded periods of time which is certainly not ideal. The actual technique to get around these obstacles is quite nuanced and we dedicate Section 5.1 to its details.

Lemma 5.1. *If two honest nodes i and j have local logs Log_i^k and Log_j^k respectively where k is any other node such that both logs contain a transaction* tx*, then $\mathsf{Log}_i^k \approx_{\text{tx}} \mathsf{Log}_j^k$.*

Proof. This result follows directly from the agreement property of FIFO-BC. □

Before getting into the details of the finalization step, we take a step back to understand why it turns out to be quite non-trivial. We look at a simple strawman protocol based on transaction timestamping that looks intuitive and analyze why it does not work.

The problem with timestamp-based ordering. Consider a simple synchronous protocol $\Pi_{\text{timestamp}}$ that works as follows:

1. When an honest node i receives a transaction tx from $\mathcal{Z}$ in round t, it assigns tx the timestamp t and broadcasts (tx, t) to all other nodes.

2. Upon waiting for $\delta_{\mathsf{ext}} + T_{\text{confirm}}$ rounds where δ_{ext} is the network delay bound for the external network and T_{confirm} is the liveness polynomial for the broadcast primitive, nodes reach agreement on the set of timestamps T to use to calculate the final timestamp for tx.

3. Calculate the final timestamp for tx as the median of all the timestamps in T. We represent this final timestamp by $\mathsf{final}(\text{tx})$

Notice how the first two steps almost perfectly resemble the gossip and agreement phases. The finalization step is also surprisingly simple, but unfortunately can lead to easy manipulation of final timestamps by a single adversary. To see why, consider 5 nodes, A, B, C, D and E, where E is malicious and two transactions, tx_1 and tx_2. tx_1 is received by nodes $A, \ldots, E$ at rounds $1, 1, 4, 4, 2$ while tx_2 is received by the nodes at rounds $2, 2, 5, 5, 3$. Now, all nodes have received tx_1 before tx_2 and consequently, $\mathsf{final}(\text{tx}_1) < \mathsf{final}(\text{tx}_2)$ should hold. However, notice how E can invert the ordering of the final timestamps simply by switching around its own timestamps for tx_1 and tx_2. E can make $\mathsf{final}(\text{tx}_1) = 3$ and $\mathsf{final}(\text{tx}_2) = 2$ which results in a timestamp of 3 for tx_1 (median of (1, 1, 3, 4, 4)) and 2 for tx_2 (median of (2, 2, 2, 5, 5)), and thus an unfair ordering.

5.1 The Finalization Stage

We describe the general theme of the finalization stage here.

Ordering two transactions. For a pair of transactions tx and tx′, how does a node i choose which one to deliver first? Suppose that the agreement phases for tx and tx′ result in the outputs L^{tx} and $L^{\mathrm{tx}'}$. Define $l_{(\mathrm{tx},\mathrm{tx}')}$ as below.

$$l_{(\mathrm{tx},\mathrm{tx}')} = \left|\left\{k \in V^{\mathrm{tx}} \cup V^{\mathrm{tx}'} \mid \mathsf{Log}_i^k(\mathrm{tx}) \le \mathsf{Log}_i^k(\mathrm{tx}')\right\}\right|$$

$l_{(\mathrm{tx},\mathrm{tx}')}$ denotes the number of logs Log_i^k where tx was ordered at or before tx′. Now, if $l_{(\mathrm{tx},\mathrm{tx}')}$ is "small," it means that a large number of nodes have received tx′ before tx. This means that the finalization for tx should wait until tx′ has been delivered. This provides a partial ordering between any two transactions. We defer the details to when we describe the actual Aequitas constructions.

Additional notation. Let tx $\lhd_i$ tx′ represent that i is waiting to deliver tx′ before proceeding with the finalization phase for tx. Lemma 5.2 shows that $l_{(\mathrm{tx},\mathrm{tx}')}$ and $l_{(\mathrm{tx}',\mathrm{tx})}$ cannot both be "small". That is, both tx and tx′ will not wait for each other or equivalently at most one of tx $\lhd_i$ tx′ and tx′ $\lhd_i$ tx will be true.

Lemma 5.2. $l_{(\mathrm{tx},\mathrm{tx}')} + l_{(\mathrm{tx}',\mathrm{tx})} \ge \left|L^{\mathrm{tx}} \cup L^{\mathrm{tx}'}\right|$

Proof. Let $X = L^{\mathrm{tx}} \cup L^{\mathrm{tx}'}$. For any $k \in X$, at least one of $\mathsf{Log}_i^k(\mathrm{tx}) \le \mathsf{Log}_i^k(\mathrm{tx}')$ and $\mathsf{Log}_i^k(\mathrm{tx}') \le \mathsf{Log}_i^k(\mathrm{tx})$ is true. k is therefore counted in either $l_{(\mathrm{tx},\mathrm{tx}')}$ or $l_{(\mathrm{tx}',\mathrm{tx})}$ which proves the required result. □

Adversarial transactions. The calculation of $l_{(\mathrm{tx},\mathrm{tx}')}$ needs to wait for the agreement phases of both tx and tx′ to finish. Now, if an adversarial node FIFO-broadcasts a transaction $\mathrm{tx}_{\mathrm{fake}}$ claiming it to be a real user transaction, then the ordering between $\mathrm{tx}_{\mathrm{fake}}$ and a real transaction tx cannot be calculated since the agreement phase for $\mathrm{tx}_{\mathrm{fake}}$ will never finish. So that this does not happen, the protocol needs to ensure that at least one honest node has received $\mathrm{tx}_{\mathrm{fake}}$ before tx (from $\mathcal{Z}$). Specifically, in the finalization stage for tx, for some other transaction tx′, a node will first check whether at least one honest node has definitely received tx′. Since an honest node has received tx′, it implies that all honest nodes will eventually receive tx as well from $\mathcal{Z}$. Moreover, the agreement stage for tx′ will eventually finish and tx will not be stuck waiting for such a tx′.

Non-transitive waiting. The Condorcet paradox can still result in non-transitive waiting. It is still possible for transactions $\mathrm{tx}_1, \mathrm{tx}_2$, and tx_3 such that $\mathrm{tx}_1 \lhd \mathrm{tx}_2$; $\mathrm{tx}_2 \lhd \mathrm{tx}_3$; and $\mathrm{tx}_3 \lhd \mathrm{tx}_1$. The way we get around this is by delivering such transactions at the same time—by placing them in the same block.

Graph based approach. Instead of a separate thread waiting for the resolution of each transaction, representing the "waiting" between transactions as a graph provides a nice way to modularize the protocol. Suppose that each node i maintains a directed graph $G_i = (G_i.V, G_i.E)$ where $G_i.V$ denotes the set of vertices and $G_i.E$ denotes the set of edges in G_i. Each vertex represents a transaction and an edge from y to x (equiv. $(y, x) \in G_i.E$) represents that x is waiting on y i.e. $x \lhd_i y$. When the agreement phase for a transaction tx completes, i does the following:

- Add tx to the graph G_i if it does not already exist.
- For all transactions tx' such $\text{tx} \lhd_i \text{tx}'$, first, if tx' does not exist in the graph, add a new vertex. Then, add the edge (tx', tx) to G_i.

As mentioned before, G_i may not be acyclic. In order to deal with the Condorcet paradox, we consider the *strongly connected components* of G_i. Recall that a subgraph G' of a directed graph G is called strongly connected if every vertex in G' can reach every other vertex in G'. A strongly connected component is a maximal strongly connected subgraph.

Intuitively, all transactions in a strongly connected component will be delivered in the same block. A cycle that exists in G_i (due to non-transitivity of transactions) will be entirely contained in the same strongly connected component. On the other hand, if a transaction does not need to wait on any other one, then it will be in a strongly connected component by itself. We can collapse G_i into a new graph G_i^* where each strongly connected component is represented as a single vertex. G_i^* is also called the *condensation* of G_i. Each vertex in G_i^* will now denote a set of transactions. We note that G_i^* will now be acyclic.

Graph Notation. Since a vertex in G_i contains a single transaction, we may use a transaction and its corresponding vertex interchangeably when referring to the vertex in G_i. Let $\mathsf{TXS}_i(v)$ be the set of transactions for a vertex $v \in G_i^*.V$. Let $\mathsf{SCC}_i(v)$ denote the strongly connected component of G_i that contains the vertex v. $\mathsf{SCC}_i(v)$ also denotes the corresponding vertex in the condensation graph G_i^*.

Ordering incomparable vertices in G_i^* and breaking ties. As mentioned before, not all pairs of vertices in G_i^* are connected by an edge. This only gives a partial ordering for delivering transactions. We still need a way to totally order vertices in G_i^*. In the leader-based version of the finalization step, we delegate this responsibility to the leader node. We elaborate on the techniques used in the leaderless version in Sections 6 and 7.

Delivering a transaction. For the leaderless protocols, the set of transactions $\mathsf{TXS}_i(v)$ corresponding to the vertex $v \in G_i^*.V$ can be delivered in the LOG output to $\mathcal{Z}$ when it is not waiting for any other transaction and it is preferred over any other transaction that it is incomparable to in the graph. For this, care must be taken to ensure that the set of transactions that tx is incomparable with is the same when all honest nodes are deciding to deliver tx. There are subtle differences between the synchronous and asynchronous protocols, which we elaborate further in Sections 6 and 7.

6 The Synchronous Aequitas protocol

We describe $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$, the leaderless Aequitas protocol for the completely synchronous setting. By "complete synchrony," we mean that both the external and internal networks are synchronous. For this section, we assume that $(\mathcal{A}, \mathcal{Z})$ respects $\Delta_{\mathsf{ext}} = (\mathsf{full}, \delta_{\mathsf{ext}})$ ext-synchrony and $\Delta_{\mathsf{int}} = (\mathsf{full}, \delta_{\mathsf{int}})$ int-synchrony.

To build the $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$ protocol, we assume a secure FIFO-BC protocol Π_{fifocast} (from Definition 3.4) and a Set-BA secure protocol Π_{sba} (from Definition 3.1) that work for any $(\mathcal{A}, \mathcal{Z})$ that respect $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-classical execution. Let $(T^{\text{fifocast}}_{\text{warmup}}, T^{\text{fifocast}}_{\text{confirm}})$ and $T^{\mathsf{Set\text{-}BA}}_{\text{confirm}}$ denote the liveness parameters for Π_{fifocast} and Π_{sba} respectively. We note that any bound for the number of corruptions f will be at least as restrictive as bounds required by Π_{fifocast} and Π_{sba}.

6.1 Protocol Description

The $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$ protocol follows much of the same general techniques from Section 5. The gossip and agreement stage take place exactly as described there. In the gossip stage, a node i forks an instance of $\Pi_{\text{fifocast}}[(\mathsf{sid}, i)]$ and uses it to broadcast transactions as they are received from $\mathcal{Z}$. After broadcasting a transaction tx, it waits until the broadcasts from all honest nodes would have arrived. Let U_i^{tx} denote the set of nodes k such that $\text{tx} \in \mathsf{Log}_i^k$. Note that all honest nodes are present in U_i^{tx}. In the agreement stage, i forks an instance of $\Pi_{\text{sba}}[(\mathsf{sid}, \text{tx})]$ to agree on a set L^{tx} indicating the nodes whose logs to use to order tx.

For the finalization stage, we now present the remaining details that were deferred from Section 5.1. Please refer to Section 5 for any notation.

Building the "waiting" graph G_i. Recall that each node i builds a graph G_i where vertices are transactions and edges denote ordering dependencies between transactions. For two transactions tx and tx′, an edge (tx′, tx) is added to G_i if $l_{(\text{tx},\text{tx}')} \leq \left|L^{\text{tx}} \cup L^{\text{tx}'}\right| - \gamma n + f$. Each node i also maintains the condensation graph G_i^* where each strongly connected component in G_i is condensed to a single vertex.

Ordering incomparable vertices in G_i^*. As mentioned before, not all two vertices in G_i^* are connected by an edge leading to only a partially ordering. Suppose that v and v' are two vertices that are currently not comparable. To determine which vertex to output first, we wait until they have a common descendant. Lemma 6.1 shows that any transaction in v' cannot have been received more than $2\delta_{\mathsf{ext}}$ rounds after a transaction in v. Consequently, any transaction that first arrives after $4\delta_{\mathsf{ext}}$ rounds will become a common descendant for both v and v' in G_i^*. We can now see where the notion of weak-liveness comes up. A common descendant for v and v' will exist if there is a transaction that arrives sufficiently late in order to "flush out" the earlier ones.

Now, we order the vertex with the higher number of descendants first. We note that once a common descendant arrives, any other transaction that arrives will also be a descendant of both v and v'. In other words, the vertex with the higher number of descendants will become fixed allowing for a consistent ordering across protocol nodes.

Lemma 6.1. *Let v_1 and v_2 be two vertices in G_i^* that do not have an edge between them. Let r_{first} denote the time when any transaction in $\mathsf{TXS}(v_1)$ was first received by a node. Let r_{last} denote the time when any transaction in $\mathsf{TXS}(v_2)$ was last received by a node. Then $r_{\mathit{last}} - r_{\mathit{first}} \leq 2\delta_{\mathsf{ext}}$.*

Proof. The proof is straightforward. Suppose that $\text{tx}_{\mathsf{first}}$ was received by some node at time r_{first}. Then, all nodes have received $\text{tx}_{\mathsf{first}}$ as input by time $r_{\mathsf{first}} + \delta_{\mathsf{ext}}$. Similarly, suppose that $\text{tx}_{\mathsf{last}}$ was received last by some node at time r_{last}. Then, no node has received $\text{tx}_{\mathsf{first}}$ as input before time $r_{\mathsf{last}} - \delta_{\mathsf{ext}}$. Since there is not an edge from v_1 to v_2, it cannot be the case that all nodes received $\text{tx}_{\mathsf{first}}$ before $\text{tx}_{\mathsf{last}}$. Therefore, $r_{\mathsf{last}} - r_{\mathsf{first}} \leq 2\delta_{\mathsf{ext}}$. □

Breaking ties. We use an a priori known ordering relation to break any ties that arise (eg. two vertices with equal number of descendants). In particular, we let Ord be a binary relation on $2^{\mathcal{T}} \times 2^{\mathcal{T}}$

that is known a priori to all nodes. $2^{\mathcal{T}}$ represents the power set of $\mathcal{T}$. The relation is defined on sets of transactions (rather than individual transactions only) since we may deliver several transactions at once. We assume that Ord is supplied to all nodes on initialization by $\mathcal{Z}$. We will use this function to deterministically break ties between two sets of transactions when neither should clearly come before the other. For two sets S_1 and S_2, $(S_1, S_2) \in \mathsf{Ord}$ implies that all nodes agree S_1 should come before S_2 if there is no clear winner. Ord can also be used to order transactions in the same block. In general, the Ord relation only needs to satisfy two properties:

- $\forall (a, b) \in 2^{\mathcal{T}} \times 2^{\mathcal{T}}; a \neq b$, exactly one of (a, b) and (b, a) is in Ord
- $\forall a, b, c \in 2^{\mathcal{T}}$, if $(a, b) \in \mathsf{Ord}$ and $(b, c) \in \mathsf{Ord}$ then $(a, c) \in \mathsf{Ord}$

We note Ord can be defined using a simple alphabetical or ascending order.

Delivering transactions. The transactions $\mathsf{TXS}_i(v)$ of a vertex v in G_i^* can be delivered when:

- v is a source vertex i.e. it has no incoming edge. This ensures that v is not waiting on any other transaction to be delivered first.
- $2\delta_{\mathsf{ext}}$ rounds have passed since v was added to the graph. This ensures that any other vertex v' that v is incomparable to, is also present in the graph.
- For any other source vertex v', v has a common descendant with v' and either has more descendants or has an equal number of descendants and $(\mathsf{TXS}_i(v), \mathsf{TXS}_i(v')) \in \mathsf{Ord}$ holds. This ensures that every node will order v before v'.

Bound on f. Suppose that $(\gamma, \cdot)$ order-fairness needs to be realized. This implies that if γn nodes receive transactions in a particular order, it must be reflected in the final ordering. Since f nodes can be adversarial, the output must be the same even if $\gamma n - f$ of those orderings are seen. Now, as we don't want a bi-directed edge to be added to G_i (this can lead to an unbounded length cycle), $\gamma n - f > \frac{n}{2}$ must hold. Equivalently, $n > \frac{2f}{2\gamma - 1}$. For $\gamma = 1$ block order-fairness, we require an honest majority.

Communication Complexity. Let s be the size (in bits) of a transaction. Let $\mathcal{B}(\lambda)$ be the communication complexity of an optimal λ-bit Byzantine agreement protocol. Then, reliable broadcast for one sender can be instantiated with $\mathcal{O}(\mathcal{B}(s))$ communication for each s-bit transaction tx. Consequently, for each transaction, $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$ has communication complexity $\mathcal{O}(n\mathcal{B}(s) + n\mathcal{B}(1))$.

6.2 Protocol Pseudocode

Initialization. At the start of the protocol, we assume that i receives the identities of other protocol nodes, n, f, the maximum network delays δ_{int}, δ_{ext}, and the binary relation Ord. A $\mathsf{FIFO\text{-}BC}$ protocol Π_{fifocast} and a $\mathsf{Set\text{-}BA}$ protocol Π_{sba} have also been agreed upon a priori. Let $T_{\text{confirm}}^{\text{fifocast}}$ and $T_{\text{confirm}}^{\text{sba}}$ represent the liveness bounds for Π_{fifocast} and Π_{sba} respectively. Now, for each $j \in \mathcal{N}$, i initializes $\mathsf{Log}_i^j \leftarrow []$. It also initializes an empty graph G_i and a final output log LOG_i.

- At the start of a round r, when i receives a set of transactions txs from $\mathcal{Z}$, it does the following:

1. **(Gossip)**

   (a) Fork an instance of $\Pi_{\text{fifocast}}[(\mathsf{sid}, i)]$ with i as the sender, if it does not already exist.

   (b) Send txs as input to $\Pi_{\text{fifocast}}[(\mathsf{sid}, i)]$.

   (c) Record $(\mathsf{sid}, \mathsf{txs}, \mathsf{gossip\text{-}end}, r + \delta_{\mathsf{ext}} + T^{\text{fifocast}}_{\text{confirm}})$

2. **(Agreement)**

   (a) Check if there is any previously recorded tuple $(\mathsf{sid}, \mathsf{gossip\text{-}end}, \mathsf{txs}', r')$ such that $r = r'$.

   (b) For such a tuple for txs', for each $\text{tx} \in \mathsf{txs}'$, fork an instance of $\Pi_{\text{sba}}[(\mathsf{sid}, \text{tx})]$ and provide it the input U_i^{tx}.

   (c) Record $(\mathsf{sid}, \mathsf{agreement\text{-}end}, \text{tx}, r + T^{\text{sba}}_{\text{confirm}})$ for each $\text{tx} \in \mathsf{txs}'$.

3. **(Build Graph)**

   (a) Check if there is any previously recorded tuple $(\mathsf{sid}, \mathsf{agreement\text{-}end}, \text{tx}, r')$ such that $r = r'$.

   (b) For such a tuple for tx, first add a vertex denoted by tx to G_i if it does not already exist. Now, for any other transaction tx' seen so far that has not yet been delivered,

      i. Let $u = \left|\left\{k \in L^{\text{tx}} \mid \text{tx}' \in \mathsf{Log}_i^k\right\}\right|$.

      ii. If $u \geq |L^{\text{tx}}| - (n - f) + 1$, compute $l_{(\text{tx},\text{tx}')}$ as per Section 5.1.

      iii. If $l_{(\text{tx},\text{tx}')} \leq \left|L^{\text{tx}} \cup L^{\text{tx}'}\right| - \gamma n + f$, then record $\text{tx} \lhd \text{tx}'$. Add an edge (tx', tx) to G_i if it does not already exist.

   (c) Record $(\mathsf{sid}, \mathsf{graph\text{-}end}, \text{tx}, r + 2\delta_{\mathsf{ext}} + 1)$ for tx

4. **(Finalization)**

   (a) Compute the *condensation* graph G_i^* of G_i by collapsing each strongly connected component into a single vertex.

   (b) Let V_{source} be the set of vertices in G_i^* where $v \in V_{\mathsf{source}}$ if it satisfies:
      - All transactions in $\mathsf{TXS}(v)$ have been received.
      - v is a source vertex in G_i^*. That is, v has no incoming edges.

   (c) Let $V_{\mathsf{finalize}} \subseteq V_{\mathsf{source}}$ be the set of vertices v that also satisfy:
      - For all $\text{tx}^* \in \mathsf{TXS}(v)$, there is any previously recorded tuple $(\mathsf{sid}, \mathsf{graph\text{-}end}, \text{tx}^*, r')$ with $r \geq r'$

   (d) For $v \in V_{\mathsf{source}}$, let $\mathsf{Desc}(v)$ denote the descendants of v in G_i^*. Let $\mathsf{nDesc}(v) = |\mathsf{Desc}(v)|$ i.e. the number of descendants.

   (e) For $v \in V_{\mathsf{finalize}}$ and $v' \in V_{\mathsf{source}}$, let $\mathsf{common\text{-}desc}_{(v,v')}$ be a boolean that denotes whether v and v' have a common descendant. That is, we define $\mathsf{common\text{-}desc}_{(v,v')} := (\mathsf{Desc}(v) \cap \mathsf{Desc}(v') \neq \emptyset)$

   (f) If there is a $v \in V_{\mathsf{finalize}}$ such that for all $v' \in V_{\mathsf{source}}$,
      - $\mathsf{common\text{-}desc}_{(v,v')} = \mathsf{true}$
      - Either $\mathsf{nDesc}(v) > \mathsf{nDesc}(v')$ holds or $(\mathsf{nDesc}(v) = \mathsf{nDesc}(v')) \wedge (\mathsf{TXS}(v), \mathsf{TXS}(v')) \in \mathsf{Ord}$.

then, deliver transactions in v by appending $\mathsf{TXS}(v)$ to LOG_i. Remove v from G_i^* and the corresponding vertices form G_i.

(g) Repeat steps 4b to 4f until there is no such v in step 4f.

(h) Output the current LOG_i to $\mathcal{Z}$.

- When i receives txs from $\Pi_{\text{fifocast}}[(\mathsf{sid}, j)]$, it appends txs to Log_i^j and adds j to the set U_i^{tx}.
- When i receives the output from $\Pi_{\text{sba}}[(\mathsf{sid}, \text{tx})]$, it stores it as L^{tx}.

6.3 Consistency Proof

First, we present a simple result showing that the graphs G_i^* and G_j^* for honest nodes i and j get built in the same way.

Lemma 6.2. *Suppose that when an honest node i delivers* tx, $v = \mathsf{SCC}_i(\text{tx})$ *is the vertex that contains* tx *in* G_i^*. *That is,* tx *is delivered in the set of transactions* $\mathsf{TXS}_j(v)$. *Now, if another honest node j delivers* tx *and* $v' = \mathsf{SCC}_j(\text{tx})$ *at that point, then* $\mathsf{TXS}_i(v) = \mathsf{TXS}_j(v')$. *This means that we can drop the node subscripts.*

Theorem 6.3 (Consistency of $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$)**.** *Consider any* $n, f, \gamma, \Delta_{\mathsf{ext}} = (\mathsf{full}, \delta_{\mathsf{ext}}), \Delta_{\mathsf{int}} = (\mathsf{full}, \delta_{\mathsf{int}})$ *with* $n > \frac{2f}{2\gamma-1}$. *Let* Π_{fifocast} *be a secure* FIFO-BC *protocol and* Π_{sba} *be a secure* Set-BA *protocol. Then,* $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$ *satisfies consistency w.r.t. any* $(\mathcal{A}, \mathcal{Z})$ *that respects* $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$*-classical execution.*

Proof. Suppose that an honest node delivers a transaction tx_1 before another one tx_2. We need to show no honest node will deliver tx_2 without delivering tx_1 first. Let $v_{(1,i)} = \mathsf{SCC}_i(\text{tx}_1)$ and $v_{(2,i)} = \mathsf{SCC}_i(\text{tx}_1)$ be vertices in G_i^* when tx_1 and tx_2 were delivered. Further, let $v_{(1,j)} = \mathsf{SCC}_j(\text{tx}_1)$ and $v_{(2,j)} = \mathsf{SCC}_j(\text{tx}_1)$ be vertices in G_j^* when tx_1 and tx_2 were delivered. From Lemma 6.2, we know that $v_1 = v_{(1,i)} = v_{(1,j)}$ and $v_2 = v_{(2,i)} = v_{(2,j)}$. Further, $\mathsf{TXS}_i(v_1) = \mathsf{TXS}_j(v_1)$ and $\mathsf{TXS}_i(v_2) = \mathsf{TXS}_j(v_2)$. Now, either tx_1 was delivered even before tx_2 was added to G_i, or there is an edge from v_1 to v_2 in G_i^* (which caused tx_1 to be output before) or v_1 and v_2 are incomparable.

- tx_1 was delivered before tx_2 was added to G_i. This means at least $\gamma n - f (> \frac{n}{2})$ nodes have received tx_1 before tx_2. Now, for another honest node j, even if tx_2 gets added to G_j before, there will be an edge from tx_1 to tx_2 added to G_j. Further, since $\mathsf{SCC}_i(\text{tx}_1) \neq \mathsf{SCC}_i(\text{tx}_2)$, this holds for j as well. Consequently j cannot deliver tx_2 before delivering tx_1.

- If (v_1, v_2) is an edge in G_i^*, then if j delivers $\mathsf{TXS}(v_2)$, then the edge (v_1, v_2) will exist in G_j^* as well. This means that j cannot output $\mathsf{TXS}(v_2)$ before it delivers $\mathsf{TXS}(v_1)$.

- There is no edge between v_1 and v_2 in G_i^*. Node i delivers $\mathsf{TXS}(v_1)$ before because v_1 had more descendants. Since j waits for $2\delta_{\mathsf{ext}}$ time, both v_1 and v_2 are present in its graph G_j^* if j outputs $\mathsf{TXS}(v_2)$ first. Since there is no edge between them, j will need to wait for a common descendant. By this time, any other vertex that is not a common descendant will also be in G_j^*. Consequently, the difference between the number of descendants of v_1 and those of v_2 will be the same as when i delivered $\mathsf{TXS}(v_1)$. But this means that j will take the same decision as i to deliver $\mathsf{TXS}(v_1)$ before $\mathsf{TXS}(v_2)$. In other words, node j will also deliver tx_1 before tx_2.

The consistency result follows. □

6.4 Liveness Proof

As mentioned before, we show that $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$ satisfies weak-liveness.

Theorem 6.4 (Liveness of $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$)**.** *Consider any* n, f, γ*,* $\Delta_{\mathsf{ext}} = (\mathsf{full}, \delta_{\mathsf{ext}})$*,* $\Delta_{\mathsf{int}} = (\mathsf{full}, \delta_{\mathsf{int}})$ *with* $n > \frac{2f}{2\gamma-1}$*. Let* Π_{fifocast} *be a secure* FIFO-BC *protocol and* Π_{sba} *be a secure* Set-BA *protocol. Then,* $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$ *satisfies* $(4\delta_{\mathsf{ext}}, T^{\mathrm{fifocast}}, T^*_{\mathrm{confirm}})$*-weak-liveness where* $T^*_{\mathrm{confirm}} = 2\delta_{\mathsf{ext}} + T^{\mathrm{fifocast}}_{\mathrm{confirm}} + T^{\mathrm{sba}}_{\mathrm{confirm}}$ *w.r.t. any* $(\mathcal{A}, \mathcal{Z})$ *that respects* $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$*-classical execution.*

Proof. Suppose that a transaction tx was first input by $\mathcal{Z}$ in round $r > T^{\mathrm{fifocast}}_{\mathrm{warmup}}$. By round $r' = r + \delta_{\mathsf{ext}} + T^{\mathrm{fifocast}}_{\mathrm{confirm}} + T^{\mathrm{sba}}_{\mathrm{confirm}}$, tx gets added to the graph G_i and by round $r' + \delta_{\mathsf{ext}}$, it gets added to the graph G_j for any honest node j.

Any transaction that tx may depend on has to be input no later than round $r + 2\delta_{\mathsf{ext}}$. This means that in G_i^*, any vertex that $\mathsf{SCC}(\mathrm{tx})$ is incomparable needs to be added to all honest G_j^* by round $r' + 2\delta_{\mathsf{ext}}$. Now, if a transaction tx′ first arrives at round $r_{\mathsf{flush}} > r + 4\delta_{\mathsf{ext}}$, then tx′ will be a common descendant of $\mathsf{SCC}(\mathrm{tx})$ and any other vertex that it is incomparable to. $\mathsf{TXS}(\mathrm{tx})$ can now be delivered based on the descending order of number of descendants. Note that tx′ will be added by honest nodes to their graphs by time $r_{\mathsf{flush}} + 2\delta_{\mathsf{ext}} + T^{\mathrm{fifocast}}_{\mathrm{confirm}} + T^{\mathrm{sba}}_{\mathrm{confirm}}$.

In summary, if a transaction tx is first input at round r and another transaction tx′ first arrives after round $r_{\mathsf{flush}} > r + 4\delta_{\mathsf{ext}}$, then tx will be delivered by all nodes by round $r_{\mathsf{flush}} + 2\delta_{\mathsf{ext}} + T^{\mathrm{fifocast}}_{\mathrm{confirm}} + T^{\mathrm{sba}}_{\mathrm{confirm}}$. Therefore, $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$ satisfies $(4\delta_{\mathsf{ext}}, T^{\mathrm{fifocast}}_{\mathrm{warmup}}, 2\delta_{\mathsf{ext}} + T^{\mathrm{fifocast}}_{\mathrm{confirm}} + T^{\mathrm{sba}}_{\mathrm{confirm}})$-weak-liveness. □

6.5 Block Order-Fairness Proof

Theorem 6.5 (Block Order-Fairness of $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$)**.** *Consider any* $n, f, \gamma, \Delta_{\mathsf{ext}} = (\mathsf{full}, \delta_{\mathsf{ext}}), \Delta_{\mathsf{int}} = (\mathsf{full}, \delta_{\mathsf{int}})$ *with* $n > \frac{2f}{2\gamma-1}$*. Let* Π_{fifocast} *be a secure* FIFO-BC *protocol (Definition 3.4) and* Π_{sba} *be a secure* Set-BA *protocol (Definition 3.1). Then,* $\Pi_{\mathsf{Aequitas}}^{\mathsf{sync,nolead}}$ *satisfies* $(\gamma, T_{\mathrm{warmup}})$ *block order-fairness w.r.t. any* $(\mathcal{A}, \mathcal{Z})$ *that respects* $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$*-classical execution.*

Proof. The proof is straightforward. First, we note that if γn nodes receive tx_1 before tx_2, then at least $\gamma n - f$ honest ones do. This means that then there will be an edge from tx_1 to tx_2 in all honest G_i. Consequently, either tx_1 will be delivered before tx_2 by all nodes, or it will end up in the same strongly connected component as tx_2 and be delivered at the same time. □

7 The Asynchronous Aequitas protocol

We describe the leaderless asynchronous protocol $\Pi_{\mathsf{Aequitas}}^{\mathsf{async,nolead}}$ in this section. The general technique is the same as the one for its synchronous equivalent which we discussed in Section 6.1. We note the major modifications here:

- First, we note that we can no longer wait for a specific number of rounds, since we are not making any synchrony assumptions. Rather, to start the agreement phase, a node i waits to receive a transaction $n - f$ times from other nodes. This means that after the agreement phase for tx returns the set L^{tx}, only $n - 2f$ indices are guaranteed to be honest (instead of $n - f$ honest in the synchronous protocol).

- Now, to realize $(\gamma, T_{\text{warmup}})$ block order-fairness, we need $\gamma n - 2f > \frac{n}{2}$ to hold. Equivalently, we need $n > \frac{4f}{2\gamma-1}$.
- When i receives the output L^{tx} from the agreement phase, it makes sure that any transaction that appears in at least $f+1$ logs is added to the graph G_i. This ensures that before delivering tx, any other vertex that it is "incomparable" with exists in the graph.
- Suppose that v and v' are incomparable. The synchronous protocol delivered the vertex with the higher number of descendants when a common descendant v_{common} was added to the graph. This is not enough to ensure consistent ordering across nodes in the asynchronous setting. Before delivering v or v', a node also needs to wait for vertices incomparable with v_{common} to be received.

The rest of the protocol is more or less identical to the synchronous protocol $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$. For completeness, we write down the entire protocol in Section 7.1. In the subsequent sections, we show that $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$ satisfies consistency, liveness, and order-fairness.

7.1 Protocol Pseudocode

We describe $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$ for $\gamma = 1$ for an honest node i below:

- **(Gossip)** When i receives a set of transactions txs from $\mathcal{Z}$, it does the following:
  1. Fork an instance of $\Pi_{\text{fifocast}}[(\mathsf{sid}, i)]$ with i as the sender, if it does not already exist.
  2. Send txs as input to $\Pi_{\text{fifocast}}[(\mathsf{sid}, i)]$.
- When i receives txs from $\Pi_{\text{fifocast}}[(\mathsf{sid}, j)]$, it does the following:
  1. Append txs to Log_i^j and add j to the set U_i^{tx}.
  2. **if** $|U_i^{\text{tx}}| \geq n - f,$ **then** fork an instance of $\Pi_{\mathsf{Set\text{-}BA}}[(\mathsf{sid}, \text{tx})]$ and provide it the input U_i^{tx}
- When i receives L^{tx} from $\Pi_{\text{sba}}[(\mathsf{sid}, \text{tx})]$, it does the following:
  1. Record the output L^{tx}
  2. Add a vertex denoted by tx to G_i if it does not already exist
  3. For any tx$'$ seen in at least $f+1$ Log_i^j, add tx$'$ to G_i if it does not already exist
  4. For any tx$'$ in G_i, if $V^{\text{tx}'}$ exists, calculate $l_{(\text{tx},\text{tx}')}$ as per Section 5.1. If $l_{(\text{tx},\text{tx}')} \leq f$, add the edge (tx', tx) to G_i
  5. Run the $\mathsf{Finalization}$ step
- **(Finalization)**
  1. Compute the *condensation* graph G_i^* of G_i by collapsing each strongly connected component into a single vertex.
  2. Let V_{source} be the set of vertices in G_i^* where $v \in V_{\mathsf{source}}$ if it satisfies:
     - All transactions in $\mathsf{TXS}(v)$ have been received.
     - v is a source vertex in G_i^*. That is, v has no incoming edges.

3. For $v \in V_{\mathsf{source}}$, let $\mathsf{Desc}(v)$ denote the descendants of v in G_i^*. Let $\mathsf{nDesc}(v) = |\mathsf{Desc}(v)|$ i.e. the number of descendants.
4. For $v, v' \in V_{\mathsf{source}}$, let $\mathsf{common\text{-}desc}_{(v,v')}$ be a boolean that denotes whether v and v' have a common descendant. That is, we define $\mathsf{common\text{-}desc}_{(v,v')} := (\mathsf{Desc}(v) \cap \mathsf{Desc}(v') \neq \emptyset)$
5. If there is a $v \in V_{\mathsf{source}}$ such that for all $v' \in V_{\mathsf{source}}$,
   - $\mathsf{common\text{-}desc}_{(v,v')} = \mathsf{true}$
   - Suppose that v_{common} is the common descendant. Then, check that the transactions in any vertex incomparable with v_{common} has been received.
   - Either $\mathsf{nDesc}(v) > \mathsf{nDesc}(v')$ holds or $(\mathsf{nDesc}(v) = \mathsf{nDesc}(v')) \wedge (\mathsf{TXS}(v), \mathsf{TXS}(v')) \in \mathsf{Ord}$.

   then, deliver transactions in v by appending $\mathsf{TXS}(v)$ to LOG_i. Remove v from G_i^* and the corresponding vertices form G_i.
6. Repeat steps 2 to 5 until there is no such v in step 5.
7. Output the current LOG to $\mathcal{Z}$.

7.2 Consistency Proof

Theorem 7.1 (Consistency of $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$)**.** *Consider any $n, f < \frac{n}{4}, \Delta_{\mathsf{ext}}, \Delta_{\mathsf{int}}$. Let Π_{fifocast} be a secure* FIFO-BC *protocol and Π_{sba} be a secure* Set-BA *protocol. Then, $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$ satisfies consistency w.r.t. any $(\mathcal{A}, \mathcal{Z})$ that respects $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$-classical execution.*

Proof. Suppose that an honest node i delivers transactions in $v_1 = \mathsf{SCC}_i(\mathrm{tx}_1)$ before $v_2 = \mathsf{SCC}_i(\mathrm{tx}_2)$. We first note that the proof for Lemma 6.2 carries over even for the asynchronous setting. This implies that for any honest node j and a transaction tx, $\mathsf{SCC}_j(\mathrm{tx}) = \mathsf{SCC}_i(\mathrm{tx}) = \mathsf{SCC}(\mathrm{tx})$. Now, one of the following three cases can arise:

1. tx_1 was delivered by i even before tx_2 was added to G_i. This means that at least $n - 2f$ logs for indices in L^{tx_2} contained tx_1 before tx_2. Consequently, for any other honest node j, even if tx_2 was added to G_j before, an edge from tx_1 to tx_2 would also be added. Since tx_1 and tx_2 are not in the same strongly connected component, this implies that j cannot deliver tx_2 before first delivering tx_1.

2. $(v_1, v_2) \in G_i^*.E$. This means, that for any honest node j, G_j^* would also have this edge. Consequently, all honest nodes will deliver transactions in v_1 before transactions in v_2.

3. v_1 and v_2 are incomparable in G_i^*. Consequently, tx_2 was present before tx_1 in at least $f + 1$ logs which implies that the node tx_2 was present in G_i when tx_1 was delivered. Now, from the description of $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$, i needs to wait for a common descendant of v_1 and v_2 as well as any vertices it is incomparable with to be received and added to the graph. Let $\mathsf{Desc}_i(v_1)$ and $\mathsf{Desc}_i(v_2)$ be the descendants in G_i^* of v_1 and v_2 respectively.

   Now, let $v' \in \mathsf{Desc}_i(v_1); v' \notin \mathsf{Desc}_i(v_2)$. That is, v' is a descendant of v_1 but not of v_2. We need to show that v' is present in G_j^* for an honest j, before j delivers tx_1 or tx_2. First, we note that since v_1 and v_2 are incomparable, both are present in G_j^* before j delivers either one. This means that j also needs to wait for a common descendant v_{common} of v_1 and v_2 to be received and added to j's graph. Now, v' cannot be a descendant of v_{common} (otherwise it would also be a common descendant). Therefore, either there is an edge from v' to v_{common} in G_j^* or v' and v_{common}. In either case, j needs to wait for v' to be received and added to its graph.

This implies that any vertex that is a descendant of exactly one of v_1 and v_2 is also present in G_j^* when j is deciding whether to output transactions in v_1 or v_2 first. Consequently, the difference in the number of descendants between v_1 and v_2 is the same as when i made its decision. In other words, j will also deliver tx_1 before tx_2.

We conclude that any honest node j will also deliver tx_1 before tx_2. □

7.3 Liveness Proof

Theorem 7.2 (Liveness of $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$)**.** *Consider any* $n, f < \frac{n}{4}, \Delta_{\mathsf{ext}}, \Delta_{\mathsf{int}}$. *Let* Π_{fifocast} *be a secure* FIFO-BC *protocol and* Π_{sba} *be a secure* Set-BA *protocol. Then,* $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$ *satisfies eventual weak-liveness w.r.t. any* $(\mathcal{A}, \mathcal{Z})$ *that respects* $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$*-classical execution.*

Proof. Suppose that a transaction tx is input to a node. Eventual delivery in the external network guarantees that all nodes will eventually receiver tx. Subsequently, eventual delivery in the internal network guarantees that the agreement phase for tx will eventually end resulting in all nodes adding tx to their "waiting graph."

Now, let T be the set of transactions that were received by at least one node no later than tx. Suppose that another transaction tx′ was first received after all nodes received all transactions in T. Once again, by the eventual delivery property, tx′ is eventually added to the G_i graph for all honest nodes i.

Let $\mathsf{Incomp}_{\text{tx}}$ be the set of vertices that tx (or rather, $\mathsf{SCC}(\text{tx})$) is incomparable with. Note that tx′ will be common descendant of tx as well as any $v \in \mathsf{Incomp}_{\text{tx}}$. Any vertices incomparable with tx′ will also be eventually added to G_i for all honest nodes i. Once this happens, tx can be delivered based on the number of descendants $\mathsf{SCC}(\text{tx})$ has. In other words, tx will be eventually delivered by all nodes.

□

7.4 Block Order-Fairness Proof

Theorem 7.3 (Block Order-Fairness of $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$)**.** *Consider any* $n, f < \frac{n}{4}, \Delta_{\mathsf{ext}} = (\mathsf{full}, \delta_{\mathsf{ext}}), \Delta_{\mathsf{int}} = (\mathsf{full}, \delta_{\mathsf{int}}), \gamma = 1$. *Let* Π_{fifocast} *be a secure* FIFO-BC *protocol and* Π_{sba} *be a secure* Set-BA *protocol. Then,* $\Pi^{\mathsf{async,nolead}}_{\mathsf{Aequitas}}$ *satisfies* $(\gamma, T_{\text{warmup}})$ *block order-fairness w.r.t. any* $(\mathcal{A}, \mathcal{Z})$ *that respects* $(n, f, \Delta_{\mathsf{int}}, \Delta_{\mathsf{ext}})$*-classical execution.*

Proof. This proof proceeds in the same way as the block order-fairness proof for $\Pi^{\mathsf{sync,nolead}}_{\mathsf{Aequitas}}$. □

8 Other results

8.1 Leader-Based Aequitas Protocols

We use this section to describe a sketch of the leader-based Aequitas constructions, $\Pi^{\mathsf{sync,lead}}_{\mathsf{Aequitas}}$ and $\Pi^{\mathsf{async,lead}}_{\mathsf{Aequitas}}$. For this, we will pair an existing leader-based consensus protocol Π_{leader} with the three Aequitas stages described in Section 5.

The Aequitas stages. Each node follows the three stages of the Aequitas protocol. An honest node i broadcasts or "gossips" transactions as it receives them from the environment. Next, all nodes agree on which of these broadcasts to use to determine the ordering for a particular transaction. Finally, i builds the "waiting" graph G_i.

Leader proposal. The actual method of selecting the leader is orthogonal to our construction. Leaders may be cycled periodically or only when there is a detected failure. We only assume that the current leader node is known to all nodes so that proposals from non-leader nodes can be immediately rejected. The current leader node proposes a set of blocks to add to the log. Suppose that we represent the proposal by $S_1, \ldots, S_p$ where each of the S_x are sets of transactions. Before accepting the proposal, an honest node does the following:

1. Use the protocol Π_{leader} to reach agreement on the block proposal (to ensure that the leader does not equivocate). During the voting for Π_{leader}, ensure that at least $f+1$ nodes received the transactions in the proposal from $\mathcal{Z}$.

2. Ensure that the proposed transactions are valid.

3. For each set S_x from S_1 to S_p,

   - Wait for all $\mathrm{tx} \in S_x$ to be received and added to G_i. If all $\mathrm{tx} \in S_x$ do not belong to the same strongly connected component in G_i, then reject the proposal.
   - If $\mathsf{SCC}(\mathrm{tx})$ has an incoming edge in G_i^* that has not been delivered from S_1 to S_{x-1}, then reject the proposal.

4. Accept the proposal and append $S_1, \ldots, S_p$ to LOG_i.

5. Remove the delivered transactions from G_i (and G_i^*).

6. Output LOG_i to $\mathcal{Z}$.

8.2 Adding Order-Fairness to Any Consensus Protocol

As mentioned before, one of the upshots of our Aequitas constructions is that they provide a generic compiler that allows any standard consensus protocol to be converted into one that provides order-fairness. Aequitas protocols only rely on reliable broadcast and Byzantine broadcast, both of which can be realized by any existing consensus protocol.

8.3 Send Order-Fairness

Throughout the paper, we focused on notions of order-fairness for which transaction ordering is determined by the order in which transactions were *received* by the protocol nodes. An mentioned before, an alternative notion is that of *send* order-fairness, where transactions are ordered according to the time they were sent by users. It is easy to see that for this to work, it would require a trusted or verifiable client-side timestamp. In other words, there needs to be a trusted way for a client to prove that her transaction was generated at time t.

Intuitively, this would require the presence of some trusted party to attest to the accuracy of the generated timestamp. Trusted execution environments (TEEs), e.g., Intel SGX [28], are a potential way to provide such a trusted timestamp as they provide protection for client-side software from a untrusted host (i.e., the client). Unfortunately, current TEE implementations cannot provide trusted global time. Retrieving time from a trusted source is not enough since an untrusted host could arbitrarily delay incoming timestamps.

Still, a user could always generate a trusted timestamp and then simply hold on to the attested transaction until a favorable time. If the external network (between the users and protocol nodes) is asynchronous, then there is no way to distinguish whether a transaction was delayed by the network or simply withheld by the user. Moreover, an asynchronous network would also require protocol nodes to wait an unbounded amount of time to ensure that no transaction should be ordered earlier. Consequently, for send order-fairness to be feasible, it is imperative that the external network be synchronous.

We note that client-side timestamps would enable a new design paradigm for time-sensitive systems (e.g., financial exchanges) and we we plan to explore this in future works.

Acknowledgements

This work was funded by NSF grants CNS-1514163, CNS-1564102, and CNS-1704615, as well as support from IC3 industry partners. We would also like to thank Mic Bowman at Intel for drawing attention to potential applications.

References

[1] Ittai Abraham et al. "Solida: A Blockchain Protocol Based onReconfigurable Byzantine Consensus". In: *OPODIS*. 2017, 25:1–25:19.

[2] Ittai Abraham et al. *Sync HotStuff: Simple and Practical Synchronous State Machine Replication*. Cryptology ePrint Archive, Report 2019/270. `https://eprint.iacr.org/2019/270`. 2019.

[3] Yair Amir et al. "Prime: Byzantine Replication Under Attack". In: *IEEE Transactions on Dependable and Secure Computing* 8 (4 2011), pp. 564–577.

[4] Avi Asayag et al. "A Fair Consensus Protocol for Transaction Ordering". In: *ICNP*. 2018, pp. 55–65.

[5] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. "RBFT: Redundant Byzantine Fault Tolerance". In: *ICDCS*. 2013, pp. 297–306.

[6] Leemon Baird. *The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance*. `https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf`. 2016.

[7] Shehar Bano et al. *Consensus in the Age of Blockchains*. arXiv:/1711.03936. 2017.

[8] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *DSN*. 2014, pp. 355–362.

[9] Gabriel Bracha and Sam Toueg. "Asynchronous Consensus and Broadcast Protocols". In: *J. ACM* 32.4 (1985), pp. 824–840.

[10] Christian Cachin and Marko Vukolić. *Blockchain Consensus Protocols in the Wild*. arXiv:/1707.01873. 2017.

[11] Christian Cachin et al. "Secure and Efficient Asynchronous Broadcast Protocols". In: *CRYPTO*. 2001, pp. 524–541.

[12] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *FOCS*. 2001, pp. 136–147.

[13] Ran Canetti and Tal Rabin. "Universal composition with joint state". In: *CRYPTO*. 2003, pp. 265–281.

[14] Ran Canetti et al. "A Universally Composable Treatment of Network Time". In: *CSF*. 2017, pp. 360–375.

[15] Ran Canetti et al. "Universally composable security with global setup." In: *TCC*. 2007, pp. 61–85.

[16] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI*. 1999, pp. 173–186.

[17] T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. "Consensus through Herding". In: *EUROCRYPT*. 2019, pp. 720–749.

[18] Allen Clement et al. "Making byzantine fault tolerant systems tolerate byzantine faults". In: *NDSI*. 2009, pp. 153–168.

[19] *Condorcet Paradox*. `https://en.wikipedia.org/wiki/Condorcet_paradox`.

[20] Flaviu Cristian et al. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement". In: *Information and Computation* 118.1 (1995), pp. 158–179.

[21] Philip Daian et al. "Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges". In: *IEEE SP*. 2020.

[22] Danny Dolev and H. Raymond Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM J. Comput* 12 (1983), pp. 656–666.

[23] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323.

[24] *Ethereum*. `https://ethereum.org/`.

[25] Ittay Eyal and Emin Gün Sirer. "Majority is not Enough: Bitcoin Mining is Vulnerable". In: *FC*. 2014, pp. 436–454.

[26] Yossi Gilad et al. "Algorand: Scaling Byzantine Agreements for Cryptocurrencies". In: *SOSP*. 2017, pp. 51–68.

[27] Chi Ho, Danny Dolev, and Robbert van Renesse. "Making distributed systems robust". In: *OPODIS*. 2007, pp. 232–246.

[28] *Intel Software Guard Extensions*. `https://software.intel.com/en-us/sgx`.

[29] Aggelos Kiayias et al. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *CRYPTO*. 2017, pp. 357–388.

[30] Eleftherios Kokoris-Kogias et al. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *IEEE SP*. 2018, pp. 583–598.

[31] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *TOPLAS* 4.3 (1982), pp. 382–401.

[32] Kfir Lev-Ari et al. "FairLedger: A Fair Blockchain Protocol for Financial Institutions". In: *OPODIS*. 2019, 1:1–1:16.

[33] Michael Lewis. *Flash boys: a Wall Street revolt*. WW Norton & Company, 2014.

[34] Loi Luu et al. "SmartPool: Practical Decentralized Pooled Mining". In: *USENIX Security*. 2017, pp. 1409–1426.

[35] Jean-Philippe Martin and Lorenzo Alvisi. "Fast Byzantine Consensus". In: *IEEE Transactions on Dependable and Secure Computing* 3 (3 2006), pp. 202–215.

[36] Andrew Miller et al. "Non-outsourceable scratch-off puzzles to discourage bitcoin mining coalitions". In: *CCS*. 2015, pp. 680–691.

[37] Andrew Miller et al. "The Honey Badger of BFT Protocols". In: *CCS*. 2016, pp. 31–42.

[38] Rafael Pass and Elaine Shi. "FruitChains: A Fair Blockchain". In: *PODC*. 2017, pp. 315–324.

[39] Rafael Pass and Elaine Shi. "Hybrid Consensus: Efficient Consensus in the Permissionless Model". In: *DISC*. 2017, pp. 1–16.

[40] Rafael Pass and Elaine Shi. "Rethinking Large-Scale Consensus". In: *CSF*. 2017, pp. 115–129.

[41] Rafael Pass and Elaine Shi. "Thunderella: Blockchains with Optimistic Instant Confirmation". In: *EUROCRYPT*. 2018, pp. 3–33.

[42] Team Rocket et al. *Scalable and Probabilistic Leaderless BFT Consensus through Metastability*. arXiv:/1906.08936. 2019.

[43] Giuliana Santos Veronese et al. "Efficient Byzantine Fault-Tolerance". In: *IEEE Transactions on Computers* 62.1 (2013), pp. 16–30.

[44] Giuliana Santos Veronese et al. "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary". In: *SRDS*. 2009, pp. 135–144.

[45] Maofan Yin et al. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *PODC*. 2019, pp. 347–356.