Optimal Asynchronous Byzantine Consensus with Fair Separability

Vincent Gramoli\textsuperscript{1,2}, Zhenliang Lu\textsuperscript{1}, Qiang Tang\textsuperscript{1}, and Pouriya Zarbafian\textsuperscript{1}

\textsuperscript{1} University of Sydney, Sydney, Australia  
\{zhenliang.lu, qiang.tang, pouriya.zarbafian\}@sydney.edu.au  
\textsuperscript{2} Redbelly Network, Sydney, Australia  
v Vincent.gramoli@redbelly.network

Abstract. Despite ensuring both consistency and liveness, state machine replication protocols remain vulnerable to adversaries who manipulate the transaction order. To address this, researchers have proposed order-fairness techniques that rely either on building dependency graphs between transactions, or on assigning sequence numbers to transactions. Existing protocols that handle dependency graphs suffer from sub-optimal performance, resilience or security. On the other hand, Pompe (OSDI ’20) introduced the novel ordering notion of ordering linearizability that uses sequence numbers. However, Pompe’s ordering only applies to committed transactions, opening the door to order-fairness violation when there are network delays, and vulnerability to performance downgrade when there are Byzantine attackers. A stronger notion, fair separability, was introduced to require ordering on all observed transactions. However, no implementation of fair separability exists.

In this paper, we introduce a protocol for state machine replication with fair separability (SMRFS); moreover, our protocol has communication complexity $O(n\ell + \lambda n^2)$, where $n$ is the number of processes, $\ell$ is the input (transaction) size and $\lambda$ is the security parameter. This is optimal when $\ell \geq \lambda n$, while previous works have cubic communication. To the best of our knowledge, SMRFS is the first protocol to achieve fair separability, and the first implementation of fair ordering that has optimal communication complexity and optimal Byzantine resilience.

1 Introduction

State machine replication (SMR) is one of the fundamental concepts of distributed systems and has been studied for decades. SMR enables processes to replicate a set of transactions while ensuring that each correct process adopts the same order of transactions. In the past decade, the widespread adoption of blockchains [17] in decentralized applications such as decentralized finance has brought attention to the underlying technology. It has been observed [8] that malicious users were leveraging the fact that the SMR specification does not ensure any specific order, and reordering transactions to steal profits from honest users. For example, front-running attacks (and more broadly, miner extractable values), which are illegal in centralized exchanges, have been causing users hundreds of millions of financial loss in decentralized exchange systems [19].

To ensure fair transaction ordering, several recent solutions [11,10,6,12,23] propose extending the SMR specification with constraints on the ordering of transactions. The first paradigm for achieving order-fairness [11,10,6], relies on building dependency graphs between transactions based on the relative order of transactions observed locally by each process. However, this approach requires handling complex dependency graphs between transactions and can lead to cyclic dependencies, also known as Condorcet cycles [4], between transactions. It follows that the protocols along these lines are either sacrificing optimal corruption, (e.g., only handling 1/4 corruption [11,10]), providing only weaker form of liveness [11,6], or sub-optimal communications (e.g., cubic for [11,6]).

Ordering linearizability and its insufficiencies. A notable alternative solution, Pompe [23], extends the SMR specification with ordering linearizability (OL). OL requires that if a transaction
tx_1 is observed by all correct processes before any correct process observes a transaction tx_2, and that both tx_1 and tx_2 are committed, then tx_1 must be ordered before tx_2. Pompé gave up explicit “identification” of the Condorcet cycles (if exist), and only deals with meaningful and realizable order fairness among the rest, giving the hope of eliminating the drawbacks mentioned above.

In Pompe [23], time is divided into timeslots (cf. Figure 1), and each timeslot \( k = [time_1, time_2] \), comprises an ordering phase where processes collect sequence numbers for their transactions (i.e., \( [time_1, time_2 + \Delta] \)), followed by a consensus phase, starting at \( time_2 + \Delta \), when processes decide the transactions output in the timeslot \( k \).

![Fig. 1: The execution flow of Pompe.](image)

Despite its ingenuity and potential, Pompe still presents serious drawbacks, both on security (fair ordering) and performance.

First, OL is not universally applicable to all the transactions observed by correct processes; rather, it is limited to the transactions that have been committed. This condition actually opens the door for fair-ordering violation. Consider a scenario where a transaction \( tx_1 \) is observed by all correct processes before any correct process observes another transaction \( tx_2 \). According to the Pompe protocol, if the issuer of \( tx_1 \) is Byzantine, it can abort the protocol prematurely for \( tx_1 \) so that \( tx_1 \) is not committed in this epoch. Consequently, \( tx_2 \) ends up being committed first, while \( tx_1 \) remains uncommitted. Furthermore, as it is also highlighted in [10], even when the issuer of \( tx_1 \) is correct, the non-synchronous (or adversarial) nature of the network can introduce delays for the messages from the issuer. This delay may prevent processes from receiving the sequence numbers collected for \( tx_1 \) before \( time_2 + \Delta \), thus leading to the expiration of the sequence numbers collected for \( tx_1 \). As a result, \( tx_1 \) is not committed, whereas \( tx_2 \) is not influenced and can be committed in the current epoch. Even though the broadcaster of \( tx_1 \) can collect a new set of sequence numbers and resubmit \( tx_1 \), \( tx_1 \) is ordered after \( tx_2 \), and thus the fair-ordering of OL simply vanishes. (Or more precisely, there is an inherent trade-off between OL and standard liveness: if the protocol still wants to ensure OL, then \( tx_1 \) has to be dropped, which directly violates the liveness property of SMR).

The second drawback of Pompe resides in its communication complexity: Pompe has a high cubic communication complexity. What is worse, there can potentially be a significant blow-up of communication complexity in adversarial cases. For example, in the ordering phase of Pompe, a malicious client can send a substantial number of transactions to a malicious process \( p_b \), and \( p_b \) can consistently abort the protocol after collecting its set of timestamps. Although, the transactions broadcast by \( p_b \) have been assigned sequence numbers by all correct processes, these transactions are not included in the SMR output. It follows that only a few transactions submitted by correct processes are included in the final output. Such “downgrade attack” leads to a significant increase in communication complexity for each committed transaction. Consequently, in adversarial cases, Pompe may incur \( O(N) \) communication complexity, where \( N \) could be an arbitrarily large number \( (\text{poly}(n)) \) induced by the adversary, say \( n^{50} \).

In [12], Kursawe introduced the novel notion of fair separability (FS), which expands upon the concept of OL by applying the same ordering requirement to all the transactions observed by correct
processes (rather than just committed transactions). However, implementing FS has remained an open problem. In this article, we aim to address the following question:

Is it possible to devise an asynchronous SMR protocol that not only achieves FS, but does so with an optimal communication complexity?

Our contributions: In this paper, we provide an affirmative answer to this question.

- We introduce the first implementation of fair separability in state machine replication. Furthermore, our protocol is resilience optimal and has standard liveness.
- We achieve FS with optimal communication complexity, and our protocol is also resilient to “downgrade attacks”, and thus has stable performance in all cases.

As shown in Table 1, our protocol not only implements the FS correctness condition for all transactions, but also achieves an optimal communication complexity of \(O(n\ell)\) bits per transaction when the input size \(\ell \geq n\lambda\). Additionally, it maintains resilience optimality with \(f < n/3\). The key contributions of our work can be summarized as follows.

Table 1: Average communication complexity (in bits per transaction) of existing protocols for order-fairness. Here, \(\ell\) denote the size of a transaction, and \(\lambda\) the security parameter.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Async.</th>
<th>Definition</th>
<th>Tolerance</th>
<th>Liveness</th>
<th>Time 1</th>
<th>Commu.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Optimistic</td>
<td>Worst</td>
</tr>
<tr>
<td>Pompè [23]</td>
<td>×</td>
<td>OL</td>
<td>(n &gt; 3f)</td>
<td>Weak</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Themis [10]</td>
<td>×</td>
<td>Deferring OF</td>
<td>(n &gt; 4f)</td>
<td>Standard</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Aequitas [11]</td>
<td>✓</td>
<td>Block OF</td>
<td>(n &gt; 4f)</td>
<td>Weak</td>
<td>(O(\log n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Quick Order-Fair [6]</td>
<td>✓</td>
<td>Differential OF</td>
<td>(n &gt; 3f)</td>
<td>Weak</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Ours 1, Section 7</td>
<td>✓</td>
<td>FS</td>
<td>(n &gt; 3f)</td>
<td>Standard</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Ours 2, Section 9</td>
<td>✓</td>
<td>FS</td>
<td>(n &gt; 3f)</td>
<td>Standard</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

1 We assess this metric when \(O(n)\) transactions are input simultaneously, meaning each correct process inputs \(O(1)\) transactions in constant time.
2 It means that the network is synchronous and all processes are correct.
3 In fact, the communication complexity of Pompè can be decided by the adversary.

Technical overview. The gap between FS and OL is subtle, and closing this gap while using only minimal communications requires special care. As depicted in Figure 2, our protocol comprises the following concurrent procedures.

1. **Sequencing.** A continuous transaction sequencing procedure where process collect sets of sequence numbers for their transactions and broadcast the collected sets so that they be added to the mempools of processes. At the same time, each process also FIFO broadcasts the history and order of the transactions that it observes.

2. **Output.** A finite consensus/finalization procedure for each epoch where processes try to expand the set of transactions output by SMR while preserving FS.

For each epoch, the output phase consists of three consecutive phases.

1. **Consensus 1:** a first consensus instance determines a tentative output for the epoch by combining the mempools of processes.

2. **Consensus 2:** a second consensus instance “extracts” the history of all transactions observed by \(2f + 1\) processes. This auxiliary data is used in a subsequent finalization step to determine whether earlier transactions should be included in the output so that FS is not violated.
3. Finalization: correct processes output a set of transactions that satisfies FS.

Our key addition is Consensus$_2$ (with the help of sequencing) so that Consensus$_1$ prepares proper data dissemination, while Consensus$_2$ enables processes to find and extract all potentially not-yet-committed but earlier legitimate transactions, and output them together with the tentative output from Consensus$_1$. To do this efficiently, we introduce and make use of $n$ concurrent instances of Provable and Notarizable First-in First-out Broadcast (PNFIFO-BC, see Section 5). Our main intuition is that when FS requires that $tx_1$ be ordered before $tx_2$, then $tx_1$ can be detected by looking at the history of observed transactions of any set of $2f + 1$ processes (see Sec. 6).

To prevent malicious processes from sending redundant transactions that would blow up communication complexity, we made full use of PNFIFO-BC. This primitive restricts a highly parallel ordering phase, allowing each sender to initiate the next transaction ordering phase only after completing the previous one. Furthermore, our protocol ensures that any transaction that completes the ordering phase is guaranteed to be output. To achieve optimal communication complexity: Firstly, to ensure that a transaction $tx$ observed by all correct processes is ultimately output despite the presence of a Byzantine broadcaster, correct processes rebroadcast all the transactions that they observe. And, we carefully design our protocol by incorporating erasure codes and vector commitments to reduce communication. Additionally, we share the received transaction history through the sharing of vector commitments, rather than the actual transactions themselves. These approaches altogether ensure that our protocol incurs only $O(n\ell + \lambda n^2)$ communication complexity per transaction, where $\ell$ represents the transaction size.

2 Related work

Ordering linearizability (OL) was introduced in Pompê [23] as a new paradigm for the fair ordering of transactions in SMR. In Pompê, a sender collects a set of sequence numbers for its transaction, and then broadcasts the collected set to order its transaction. By assigning a unique sequence number to each transaction, OL circumvents the potential cyclic dependencies between transactions that may arise in other paradigms for fair ordering [11,10,6]. However, OL is only ensured conditionally, and therefore Pompê only implements the weaker liveness of SMR to ensure OL in partial synchrony. In this paper, we analyze how to achieve standard liveness and unconditional OL (FS), and provide a protocol that satisfies FS by combining the secure broadcast of each observed transaction with
additional delivery rules. By leveraging cryptographic and broadcast primitives, we also make our protocol optimal in terms of communication complexity per transaction. Various works [12,22,21] also offer implementations of order-fairness derived from FS, but require a synchronous setting.

In the asynchronous setting, several works have been introduced to address order-fairness, including Aequitas [11], and a more refined approach known as Quick order-fair [6]. It consists of agreeing on the local orderings observed by a set of processes, and then ordering transactions using the relative ordering at a majority of processes. In this paradigm, during the building of dependency graphs between transactions, cycles may appear between transactions. Furthermore, in the finalization step of [6], transactions that are decided may not always be output as processes may have to wait for additional ordering information to be output by the protocol. In contrast, the finalization step of our protocol checks for other transactions that should also be output, but enables direct output of all decided transactions. Finally, current implementations of this paradigm incur at least $O(n^3)$-bit communication complexity per transaction.

3 Model and Problem Statement

3.1 Processes and Network

We consider a system of $n$ processes $P = \{p_1, p_2, \ldots, p_n\}$. Processes that follow the prescribed protocol are denoted correct, whereas Byzantine processes, can deviate from the protocol arbitrarily. We assume that at any time, the number of Byzantine processes is bounded by $f = \lceil \frac{n}{3} \rceil - 1$. We also assume a static adversary [5,16] that fully controls corrupted processes. In the static adversary model, the adversary can select up to $f$ processes prior to the start of the protocol, gain access to their internal states, and control their behaviors during the execution of the protocol.

We consider an asynchronous network where there are no bounds on message delays, as an adversary can delay messages arbitrarily, but each message sent by a correct process is eventually delivered and untampered. Furthermore, communication channels are authenticated, and Byzantine processes cannot impersonate correct processes.

3.2 Goal: State Machine Replication with Fair Separability

In this paper, our goal is to design an efficient asynchronous state machine replication protocol [13,20,18] where the ordering of the output transactions satisfies fair separability. In state machine replication (SMR), two fundamental properties must be satisfied by all correct processes: consistency and liveness. The former requires that all correct processes must output transactions in the same order, while the latter ensures that once an honest client submits a transaction, it should be output within a reasonable amount of time. At a high level, it involves clients continuously sending transactions to the correct processes. These correct processes then submit transactions by SMR-broadcasting them to all processes, and correct processes must SMR-deliver a subset of the submitted transactions in the same order within finite steps. Fair separability requires that if all correct processes observe a transaction $tx_1$ before any of them observe a transaction $tx_2$, then $tx_1$ must be SMR-delivered before $tx_2$. Formally, we define State Machine Replication (SMR) as follows:

**Definition 1 (State Machine Replication Problem).** A state machine replication protocol must ensure the following properties.

- **SMR-Consistency.** If a correct process $p_i$ SMR-delivers transactions $\{tx_1, tx_2, \ldots, tx_s\}$ and another correct process $p_j$ SMR-delivers transactions $\{tx'_1, tx'_2, \ldots, tx'_s'\}$, then $tx_k = tx'_k$ for $\forall k = \min\{s, s'\}$. Additionally, if a correct process SMR-delivers a transaction $tx$, then $tx$ is eventually SMR-delivered by all correct processes.
• **SMR-Liveness.** If a correct process SMR-broadcasts a transaction \( tx \), then \( tx \) is eventually SMR-delivered by all correct processes.

The notion of fair separability strengthens the notion of ordering linearizability. Fair separability requires that if the lowest sequence number assigned by any correct process to a transaction \( tx_2 \) is greater than the highest sequence number assigned by any correct process to a transaction \( tx_1 \), then \( tx_1 \) is ordered before \( tx_2 \) by correct processes. In order to achieve this goal, processes need to assign sequence numbers to the transactions that they observe. Then, fair separability can be achieved by using for each transaction a sequence number \( \bar{s} \) that is the median value of a set of \( 2f + 1 \) sequence numbers, because \( \bar{s} \) is upper bounded and lower bounded by a sequence number that has been assigned by a correct process [23].

**Definition 2 (Partial Order).** If a transaction \( tx_1 \) (resp. \( tx_2 \)) is assigned a sequence number \( s_1 \) (resp. \( s_2 \)), then \( tx_1 \) must be SMR-delivered before \( tx_2 \), if \( s_1 < s_2 \). We say that \( tx_1 \) is SMR-delivered before \( tx_2 \), and denote it \( tx_1 < tx_2 \).

**Definition 3 (Fair Separability).** If the highest sequence number assigned by a correct process to a transaction \( tx_2 \) is lower than the lowest sequence number assigned by any correct process to a transaction \( tx_1 \), then \( tx_1 \) must be SMR-delivered before \( tx_2 \). More formally, let \( S_1 \) (resp. \( S_2 \)) denote the set of sequence numbers assigned to transaction \( tx_1 \) (resp. \( tx_2 \)) by correct processes, then if

\[
\max_{s \in S_1} (s) < \min_{s \in S_2} (s) \Rightarrow tx_1 < tx_2.
\]

**Remark:** We would like to emphasize that SMR-Liveness alone cannot ensure fair separability (FS). It only guarantees that a correct process’s input can be output. Therefore, there is an inherent risk of malicious process input. For instance, even if all correct processes receive some input from a malicious process, these inputs may never be output, potentially compromising FS. To design a protocol that achieves FS, we must ensure that a transaction \( tx \) that has been observed by all correct processes is SMR-delivered despite a Byzantine broadcaster that decides to abort the protocol prematurely. Our SMRFS protocol addresses this issue, as confirmed in Lemma 7. In this paper, we design a framework that can assemble any underlying Byzantine consensus protocol, and enable these underlying protocols to implement fair separability.

## 4 Preliminaries

In this section, we present the building blocks that are used in our protocols. Throughout the paper, we use \( \ell \) to represent the bit length of each transaction, and \( \lambda \) denotes the cryptographic security parameter, which includes the size of the (threshold) signatures.

**Erasure code scheme.** A \((k, n)\)-erasure code scheme [2] consists of a tuple of two deterministic algorithms \( \text{Enc} \) and \( \text{Dec} \). The \( \text{Enc} \) algorithm maps any vector \( \mathbf{v} = (v_1, \cdots, v_k) \) of \( k \) data fragments into a vector \( \mathbf{m} = (m_1, \cdots, m_n) \) of \( n \) coded fragments, such that any \( k \) elements in the code vector \( \mathbf{m} \) is enough to reconstruct \( \mathbf{v} \) with the \( \text{Dec} \) algorithm. Throughout the paper, we consider a \((f + 1, n)\)-erasure code scheme where \( 3f + 1 = n \).

**Threshold signatures.** A \((t, n)\) threshold signature scheme [3] is a protocol involving \( n \) processes, where at most \( t - 1 \) processes can be corrupted and \( 0 \leq t \leq n \). Formally, a \((t, n)\)-threshold signature scheme consists of the following algorithms: \( \text{TS.KeyGen}, \text{TS.SigShare}_t, \text{TS.VrfShare}_t, \text{TS.COMB}_t \), and \( \text{TS.Vfy}_t \). The \( \text{TS.SigShare}_t \) algorithm takes a message \( m \) as input and produces a signature share. The \( \text{TS.VrfShare}_t \) algorithm is then used to verify whether the signature share is valid or not. The \( \text{TS.COMB}_t \) algorithm can generate a complete signature from at least \( t + 1 \) valid signature shares. Finally, the \( \text{TS.Vfy}_t \) algorithm is employed to verify a full signature.
Digital Signatures. We assume that all processes possess a key pair \( pk/sk \) for use in digital signatures [9]. The algorithm consists of two algorithms: \( \text{Sign}, \text{Vrf} \). For any message \( m \), it holds that \( \text{Vrf}(m, \text{Sign}(m, sk), pk) = 1 \).

Position-binding vector commitment (vc). The \( n \)-vector commitment (vc) [7] comprises three algorithms: \( \text{VecCom}, \text{Open}, \text{VerifyOpen} \). On input a vector \( m \) of any \( n \) elements, the algorithm \( \text{VecCom} \) produces a commitment \( vc \) for the vector \( m \). On input, \( m \) and \( vc \), the \( \text{Open} \) algorithm can reveal the element \( m_i \) committed in \( vc \) at the \( i \)-th position while producing a short proof \( \pi_i \), which later can be verified by \( \text{VerifyOpen} \).

Remark. Throughout the paper, we employ the vector commitment scheme from [7] and we might omit \( aux \) for presentation simplicity. All algorithms are deterministic, and both commitment \( vc \) and openness \( \pi \) are \( O(\lambda) \) bits in size.

Multi-valued validated Byzantine Agreement (MVBA): The MVBA protocol [5,1,15] always guarantees that the output value \( v \) satisfies a predefined external predicate \( Q \), i.e., \( Q(v) = 1 \). All correct processes only input values \( v \) to MVBA such that \( Q(v) = 1 \). Formally, an MVBA protocol satisfies the following properties with all but negligible probability.

- **MVBA-Termination.** If every correct process \( p_i \) inputs an externally valid value \( v_i \), then every correct process outputs a value.
- **MVBA-External-Validity.** If a correct process outputs a value \( v \), then \( Q(v) = 1 \).
- **MVBA-Agreement.** Any two distinct correct processes always output the same value.
- **MVBA-Quality.** If a correct process outputs \( v \), then the probability that \( v \) was input by the adversary is at most \( 1/2 \).

Note: In our paper, we utilize the MVBA protocol proposed by Lu et al. in [15]. In this MVBA protocol, the time complexity is \( O(1) \), the message complexity is \( O(n^2) \), and the communication complexity is \( O(n\ell + \lambda n^2) \).

5 Provable and Notarizable FIFO Broadcast

In this section, we introduce an important broadcast component used in our main protocol.

5.1 First-in First-out Broadcast

In the First-in First-out Broadcast (FIFO-BC) protocol, there exists a process known as the sender, whose primary objective is to broadcast a sequence of messages to all processes. The crucial guarantee provided by this protocol is that if the sender \( p_s \) is correct, then all correct processes will receive and deliver \( p_s \)’s messages in the exact order in which \( p_s \) broadcasts them. Importantly, even if \( p_s \) is Byzantine, the protocol guarantees that all correct processes deliver the same message set from \( p_s \) and in the same order.

Let \( \text{FIFO-BC}_s \) denote the instance of FIFO-BC initiated by sender process \( p_s \). Each process \( p_i \) maintains a local log denoted \( \text{Log}_s \) that records the output of the \( \text{FIFO-BC}_s \) instance. To refer to the \( k^{th} \) invocation and to the \( k^{th} \) output of \( \text{FIFO-BC}_s \), we employ the notations \( \text{FIFO-BC}_s[k] \) and \( \text{Log}_s[k] \), respectively. Formally, \( \text{FIFO-BC}_s \) is defined as follows:

**Definition 4 (First-in First-out Broadcast (FIFO-BC) Problem).** A First-in First-out Broadcast protocol with sender \( p_s \) ensures the following properties.

- **FIFO-BC-Liveness.** If \( p_s \) is correct and broadcasts a message \( m \), then every correct process eventually delivers \( m \).
- **FIFO-BC-Integrity.** If some correct process delivers a message \( m \), then \( m \) was previously broadcast.
• **FIFO-BC-Total-Order.** If some correct process \( p_i \) delivers \( \{ \log_s[1], \ldots, \log_s[k] \} \), and another correct process \( p_j \) delivers \( \{ \log'_s[1], \ldots, \log'_s[k'] \} \), then for every \( i \), where \( 1 \leq i \leq \min\{k, k'\} \), \( \log_s[i] = \log'_s[i] \).

• **FIFO-BC-Delivery.** If the sender is correct and is input a message \( v \) before \( v' \), then no correct process delivers \( v' \) unless it has already delivered \( v \).

Note: FIFO-BC-Total-Order implies that if a correct process \( p_i \) has delivered \( \log_s[k] \), then another correct process \( p_j \) cannot deliver \( \log_s[k] \) unless it has already delivered the preceding \( \log_s[k-1] \).

### 5.2 Provable and Notarizable First-in First-out Broadcast

In this paper, we present a protocol named Provable and Notarizable First-in First-out Broadcast (PNFIFO-BC). The output \( \log_s[k] \) of PNFIFO-BC \( k \) is structured as \( \log_s[k] := (v_k, \sigma_k) \), where \( \sigma_k \) acts as a proof that guarantees the validity of the value \( v_k \) in the \( k \)th output. This property provides provability, ensuring that each value can be verified, i.e., the value is indeed the \( k \)th output of the instance. Furthermore, we have introduced additional constraints on the output value. Our goal is for the output to satisfy a predefined predicate \( Q \), which is similar to MVBA-External-Validity, as outlined in MVBA [5,1].

The paper “Bolt-Dumbo Transformer” by Lu et al. [14] introduced a new primitive known as notarizable weak atomic broadcast (nw-ABC). This primitive exhibits a notarizability property, wherein if any correct process outputs a valid \( \log_s[k] \), then it guarantees the existence of at least \( f + 1 \) correct processes that have either already output \( \log_s[k] \) or already output \( \log_s[k-1] \). Our protocol, PNFIFO-BC, was inspired by the notarizability property, and PNFIFO-BC also provides a similar notarizability property as introduced in nw-ABC.

Compared with FIFO-BC, PNFIFO-BC offers more comprehensive guarantees. PNFIFO-BC not only fulfills all the properties of FIFO-BC, but it also introduces three additional properties related to external validity, provability, and notarizability. In PNFIFO-BC, external validity makes sure the output value is meaningful. Provability ensures that each output can be verified, while notarizability states that if any process successfully outputs a valid \( \log_s[k] \), then there are at least \( f + 1 \) correct processes that have already output a valid \( \log_s[k-1] \). Formally,

**Definition 5 (PNFIFO-BC).** A Provable and Notarizable First-in First-out Broadcast (PNFIFO-BC) protocol, with sender \( p_s \) among \( n \) processes, ensures the following properties.

- **PNFIFO-BC-Liveness.** Same as in Definition 4.
- **PNFIFO-BC-Integrity.** Same as in Definition 4.
- **PNFIFO-BC-Total-Order.** Same as in Definition 4.
- **PNFIFO-BC-Delivery.** Same as in Definition 4.
- **PNFIFO-BC-Provability.** If any correct process delivers a valid \( \log_s[k] := (v_k, \sigma_k) \), then using \( \sigma_k \) it can verify that the value \( v_k \) indeed is the \( k \)th output of PNFIFO-BC.
- **PNFIFO-BC-Notarizability.** If any correct process delivers \( \log_s[k] \), then there exists at least \( f + 1 \) correct processes that have already deliver \( \log_s[k-1] \).
- **PNFIFO-BC-External-Validity.** If any correct process delivers a valid \( \log_s[k] := (v_k, \sigma_k) \), then \( Q(v_k) = 1 \).

### 5.3 Instantiation of PNFIFO-BC

A PNFIFO-BC \( k \) protocol can be constructed using sequential multicasts and threshold signatures, as outlined in Algorithm 1. The PNFIFO-BC \( k \) protocol can be divided into four logical phases which are detailed as follows:
Algorithm 1 Provable and Notarizable First-in First-out Broadcast with sender \( p_k \): PNFIFO-BC, code for process \( p_k \) who runs the protocol in consecutive slot number \( k \) with predicate \( Q \).

\[
\text{Initialize: } k \leftarrow 1 \quad \triangleright \text{slot number} \\
\text{flags}_s \leftarrow 0 \\
\text{Log}_s \leftarrow \{\} \\
1: \text{function PNFIFO-BC}_s[k](v_k) \\
2: \quad \text{multicast Proposal}(s, k, v_k) \\
3: \quad \text{upon receiving Proposal}(s, k, v_k) \text{ from } p_s \text{ for the first time} \text{ do} \\
4: \quad \quad \text{wait flags}_s = 0 \text{ and } Q(v_k) = 1 \text{ do} \\
5: \quad \quad \quad \sigma_{k,i} \leftarrow \text{TS.SigShare}_{2f+1}(s, k, h(v_k)) \\
6: \quad \quad \quad \text{send Vote}(s, k, \sigma_{k,i}) \text{ to the sender } p_s \\
7: \quad \quad \text{flags}_s \leftarrow 1 \\
8: \quad \quad \text{upon receiving Vote}(s, k, \sigma_{k,i}) \text{ from } p_j \text{ for the first time} \text{ do} \\
9: \quad \quad \quad \text{if } T_k \leftarrow T_k \cup \{j, \sigma_{k,i}\} \text{ do} \\
10: \quad \quad \quad \text{if } |T_k| = 2f + 1 \text{ do} \\
11: \quad \quad \quad \quad \sigma_k \leftarrow \text{TS.Comb}_{2f+1}(s, k, h(v_k), T_k) \\
12: \quad \quad \quad \quad \text{multicast Final}(s, k, \sigma_k) \\
13: \quad \quad \text{upon receiving Final}(s, k, \sigma_k) \text{ from } p_s \text{ for the first time} \text{ do} \\
14: \quad \quad \quad \text{wait flags}_s = 1 \text{ do} \\
15: \quad \quad \quad \text{if } T_k \leftarrow T_k \cup \{s, k\} \text{ do} \\
16: \quad \quad \quad \quad \text{Log}_s[k] \leftarrow (v_k, \sigma_k) \\
17: \quad \quad \quad \text{flags}_s \leftarrow 0 \text{ and } k \leftarrow k + 1 \\
\]

- **Value broadcast phase:** (lines 1-2). The sender process \( p_s \) invokes PNFIFO-BC\(_s[k](v_k) \) to multicast Proposal\((s, k, v_k) \) to all.
- **Vote phase:** (lines 3-7). Whenever a correct process \( p_i \) receives a message Proposal\((s, k, v_k) \) from the sender, and that Log\(_s[k-1] \) has been delivered by \( p_i \) and that \( v_k \) satisfies the predicate \( Q \), then \( p_i \) generates a threshold share signature \( \sigma_{k,i} \) for \((s, k, h(v_k)) \) and sends back Vote\((s, k, \sigma_{k,i}) \) to the sender.
- **Generate proof phase:** (lines 8-13). When the sender has received \( n - f \) valid Vote messages, i.e., \( n - f \) valid threshold signature shares for \((s, k, h(v_k)) \), it combines these signature shares into a full threshold signature \( \sigma_k \), and multicasts Final\((s, k, \sigma_k) \) to all.
- **Output phase:** (lines 14-18). When a correct process \( p_i \) receives a valid message Final\((s, k, \sigma_k) \) from the sender, and that \( p_i \) has already sent a message Vote\((s, k, *) \) back to the sender, i.e., flags\(_s = 1 \), then \( p_i \) delivers Log\(_s[k] := (v_k, \sigma_k) \) and proceeds into the next slot \( k + 1 \).

### 5.4 Security and Complexity Analysis

In this section, we conduct an analysis of the security and costs associated with the PNFIFO-BC protocol presented in Algorithm 1.

**Theorem 1.** Algorithm 1 implements a Provable and Notarizable First-in First-out Broadcast protocol (cf. definition 5).

**Proof.** We prove each property separately.

1. **PNFIFO-BC-Liveness.** If a correct sender \( p_s \) broadcasts a message \( v_k \) by sending a Proposal message, then all correct processes eventually receive this Proposal message. Since \( 2f + 1 \leq n - f \), \( p_s \) eventually receives a set \( T_k \) of \( 2f + 1 \) valid Vote messages. As a result, \( p_s \) multicasts a valid Final message, and all correct processes eventually receive this Final message, and deliver the message \( v_k \).
2. **PNFIFO-BC-Integrity.** If a correct process $p_i$ delivers a message $m$ from a correct sender $p_s$, then it implies that $p_i$ has received a valid Final message and the corresponding Proposal message from $p_s$. Hence, it is trivial that the message $m$ was broadcast by $p_s$.

3. **PNFIFO-BC-Total-Order.** If a correct process delivers $\text{Log}_s[k] := (v_k, \sigma_k)$, it means that it satisfies the condition $\text{TS.Vfy}_{2f+1}(s, k, h(v_k), \sigma_k) = 1$. This implies that at least $2f+1$ processes have sent one Vote message in slot $k$. Since correct processes only send one Vote message in each slot, if any two distinct processes deliver $\text{Log}_s[k]$ and $\text{Log}_s'[k]$ respectively, then $\text{Log}_s[k]$ must be equal to $\text{Log}_s'[k]$. Furthermore, if a correct process does not deliver in slot $k$, it will not deliver in slot $k+1$ based on the pseudo-code (line 18). This implies that $\text{Log}_s[k+1]$ will not be delivered unless $\text{Log}_s[k]$ has been delivered. Therefore, if one correct process $p_i$ delivers $\{\text{Log}_s[1], \ldots, \text{Log}_s[k]\}$, and another correct process $p_j$ delivers $\{\text{Log}_s'[1], \ldots, \text{Log}_s'[k']\}$, it follows that $\text{Log}_s[i] = \text{Log}_s'[i]$ for every $i$ such that $1 \leq i \leq \min\{k, k'\}$.

4. **PNFIFO-BC-Delivery.** According to the pseudo-code, a correct process can deliver $\text{Log}_s[k']$ only if it has already delivered $\text{Log}_s[k]$, where $k' > k$. If the sender is correct and inputs a message $v$ in slot $k$ and another message $v'$ in slot $k' (> k)$, it follows that all correct processes will deliver $v$ before $v'$.

5. **PNFIFO-BC-Provability.** If any correct process delivers a valid $\text{Log}_s[k] := (v_k, \sigma_k)$, it means that it satisfied the condition $\text{TS.Vfy}_{2f+1}(s, k, h(v_k), \sigma_k) = 1$. If there exists another message $v'_k$ that belongs to the $k^{th}$ output, then it will also satisfy $\text{TS.Vfy}_{2f+1}(s, k, h(v'_k), \sigma_k) = 1$. However, this would require at least one correct process to perform two threshold signature shares for two distinct messages $v_k$ and $v'_k$, which contradicts the code in line 7. Therefore, $\sigma_k$ guarantees that the value $v_k$ indeed belongs to the $k^{th}$ output.

6. **PNFIFO-BC-Notarizability.** If any correct process delivers $\text{Log}_s[k] := (v_k, \sigma_k)$, it means that it satisfies the condition $\text{TS.Vfy}_{2f+1}(s, k, h(v_k), \sigma_k) = 1$. This implies that at least $f+1$ correct processes participated in the instance for slot $k$, and that they have already output $\text{Log}_s[k-1]$ as indicated by the pseudo-code (lines 17-18).

7. **PNFIFO-BC-External-Validity.** When any correct process delivers $\text{Log}_s[k] := (v_k, \sigma_k)$, it indicates that the process has successfully received a valid Final message and the corresponding Proposal message from the sender $p_s$, then, based on line 4, it follows that $v_k$ satisfies the predicate $Q$.

**Theorem 2.** The communication complexity of the PNFIFO-BC$_s$ protocol is $O(n\ell + \lambda n)$ bits per value, where $\ell$ is the size of the input value and $\lambda$ the security parameter.

**Proof.** Based on the pseudo-code of Algorithm 1, the cost breakdown can be summarized into the following four phases:

1. **Value broadcast phase:** In this phase, the value $v_k$ is broadcasted to all processes by a Proposal message, resulting in a cost of $O(n\ell)$ bits.

2. **Vote phase:** This phase involves “$n$-to-one” $n$ Vote messages, where each process sends a Vote response to the sender. The cost for this phase is $O(n\lambda)$ bits.

3. **Generate proof phase:** In this phase, the sender multicasts a Final message to all processes. The size of the Final message is $O(\lambda)$, resulting in a communication cost of $O(n\lambda)$ bits for this phase.

4. **Output phase:** This phase does not incur any additional communication cost.

Therefore, the communication complexity of the PNFIFO-BC$_s$ protocol is $O(n\ell + \lambda n)$ bits per value.

## 6 Fair Separability Conditions

In this section, we present lemmas to give an intuition on how to achieve Fair Separability (FS) (cf. Definition 3). These lemmas will be used in the analysis of the SMRFS protocol.
6.1 Ordering Linearizability: Weak Fair Separability

As mentioned earlier, in Pompē [23], a Byzantine process can collect a set of sequence numbers $S_1$ for a transaction $tx_1$, and simply abort the protocol, or collect a new set of sequence numbers at a later time. It follows that even when the network is synchronous, a transaction $tx_2$ submitted by a correct process $p_i$, and for which $p_i$ has collected a set of sequence numbers $S_2$ such that $\max_{s_{1} \in S_1}(s_1) < \min_{s_{2} \in S_2}(s_2)$, is output before $tx_1$, thus violating FS. It’s also possible that the sender of $tx_1$ is correct, but network delays cause the set $S_1$ collected for $tx_1$ to expire, which could similarly result in a violation of FS. Hence, we refer to the property achieved in [23] by weak fair separability because it only applies under the condition that both $tx_1$ and $tx_2$ are committed. For FS, the challenge is to ensure that:

- Requirement 1: if a transaction is observed by correct processes, then it must be committed.
- Requirement 2: the final ordering of transactions satisfies FS.

To satisfy FS, the key insight is to make sure that whenever transaction $tx_2$ is delivered with a sequence number $\sigma_2$, check if processes have observed transaction $tx_1$ as it should be ordered before $tx_2$, and the histories of transactions observed by $2f+1$ processes enable this verification, as we prove in claims 1 and 2. Therefore, our solution to satisfy requirements 1 and 2 is to have, on the one hand, each process submit all received transactions to the PNFIFO-BC protocol, and on the other hand, require that the final output is determined based on the outcomes of the PNFIFO-BC instances involving $2f+1$ processes.

6.2 Fair Separability

We now give an intuition of how we achieve FS. Intuitively, a transaction can be safely SMR-delivered when there is no other transaction that requires prior SMR-delivery according to FS.

**Lemma 1.** For any two transactions $tx_1, tx_2$, let $\sigma_1, \sigma_2$ be the median value of any set of $2f+1$ sequence numbers that have been assigned to $tx_1, tx_2$, respectively. If $tx_1$ and $tx_2$ satisfy FS, then $\sigma_1 < \sigma_2$.

**Proof.** Cf. Lemma 4.1 of [23], the median value of a transaction is bounded by the sequence numbers assigned by correct processes. Hence, if $tx_1 < tx_2$ holds, then $tx_1, tx_2$ meet the FS condition, thus $\max_{s_{1} \in S_1}(s_1) < \min_{s_{2} \in S_2}(s_2) \Rightarrow \sigma_1 < \sigma_2$.

**Claim 1.** Let $\sigma_1$ denote the median value of any set of $2f+1$ sequence numbers that have been assigned by processes to a transaction $tx_1$. If at least $f+1$ processes have assigned sequence numbers to a transaction $tx_2$ that are less than $\sigma_1$, then ordering $tx_1$ after $tx_2$ does not violate FS.

**Proof.** This results from the fact that $\sigma_1$ is the median of a set of $2f+1$ values, $\sigma_1$ is upper bounded and lower bounded by the sequence numbers assigned to $tx_1$ by correct processes. Therefore, if $f+1$ processes have assigned to $tx_2$ sequence numbers that are less than $\sigma_1$, then there must be at least one correct process assigning a sequence number to $tx_2$ that is lower than the sequence number assigned to $tx_1$ by another correct process, and therefore FS does not require that $tx_1 < tx_2$.

Reciprocally, if at most $2f$ processes observe a transaction $tx_1$ before a transaction $tx_2$, then ordering $tx_1$ before $tx_2$ does not violate FS.

**Claim 2.** Let $\sigma_1$ denote the median value of any set of $2f+1$ sequence numbers assigned to a transaction $tx_1$. If at most $2f$ processes have assigned sequence numbers to a transaction $tx_2$ that are less than $\sigma_1$, then ordering $tx_1$ before $tx_2$ does not violate FS.
Proof. Because $\pi_1$ is necessarily upper bounded and lower bounded by the sequence numbers assigned to $tx_1$ by correct processes, if less than $2f + 1$ processes have assigned to $tx_2$ sequence numbers that are less than $\pi_1$, then there must be at least one correct process assigning a sequence number to $tx_2$ that is larger than the sequence number assigned to $tx_1$ by another correct process, and therefore ordering $tx_1$ before $tx_2$ does not violate FS.

By adhering to these observations, we ensure that the ordering of transactions satisfies the desired FS property. Intuitively, in our protocol, we rely on a first instance of consensus (cf. Consensus$_1$) to decide on some transactions that are output in the current epoch. Suppose that transaction $tx$ is output with an order sequence number $s$. Then, in a second instance of consensus (cf. Consensus$_2$), we determine the Logs of $2f + 1$ processes. Finally, we search for potential prior transactions, denoted as $tx'$, by examining the $2f + 1$ Logs decided by Consensus$_2$. If at least $f + 1$ sequence numbers assigned to a transaction $tx'$ belonging to these $2f + 1$ Logs are less than $s$, then $tx'$ must be output before $tx$. The core idea is that if $tx_1$ and $tx_2$ satisfy the FS, and we use $\pi_2$ to represent the median value of any set of $2f + 1$ sequence numbers assigned to $tx_2$, then at least $f + 1$ sequence numbers have been assigned to transaction $tx_1$ from any of the $2f + 1$ Logs, and these $f + 1$ sequence numbers must be less than $\pi_2$. This process helps identify other transactions that may require being output before $tx_2$, and thus maintain the desired ordering properties.

7 Asynchronous State Machine Replication with Fair Separability

In this section, we present our protocol for State Machine Replication (SMR) with fair separability (SMRFS) in the asynchronous setting.

7.1 High-level Overview of SMRFS

As illustrated in Figure 2, our protocol comprises three key parts: transaction sequencing, consensus, and finalization. We first present an overview of the protocol, and then present each part in detail. In each epoch, correct processes engage in two concurrent phases. The first phase is “transaction sequencing”, which is a procedure that is continuously running. Simultaneously, the second phase, composed of two sequential consensus protocols and a “finalization”, is run for each epoch.

First, during transaction sequencing, the goal is to prepare “indexed” transaction mempools and metadata for received transcripts. Each transaction will have an order sequence number, which is the median value of a set of $2f + 1$ sequence numbers assigned to the transaction. Additionally, each process $p_i$ provably disperse all its received transactions with other processes. Furthermore, when a process $p_i$ assigns a sequence number $s$ to transaction $tx$, it produces a store[$s$] entry to indicate that $tx$ is the $s^{th}$ observed transaction.

Then, the consensus phase comprises two consensus protocols. The first one, Consensus$_1$, ensures that all correct processes agree on which transactions (from the mempools) can be output. After Consensus$_1$ has generated some tentative output, each process $p_i$ shares the order of observed transactions with all participating processes via a corresponding PNFIFO-BC2, instance. Next, the second consensus protocol, Consensus$_2$, is employed to determine which PNFIFO-BC2 instances’ output can serve as auxiliary information for identifying potential transactions that should be included in the current epoch to preserve FS. Finally, during the finalization phase, the main objective is to ensure that all correct processes deliver the same set of transactions (that maybe the combination of the candidate outputs of Consensus$_1$ and earlier ones identified by Consensus$_2$).

Note that for any two distinct transactions $tx_1$, $tx_2$, their vector commitments, $vc_{tx_1}$, $vc_{tx_2}$, are different, i.e., $tx_1 \neq tx_2 \Rightarrow vc_{tx_1} \neq vc_{tx_2}$. This uniqueness allows us to use the vector commitment as a representation of the transaction. Importantly, the size of the vector commitment is smaller than the size of the transaction. To ensure optimal communication complexity, we utilize the vector commitments of transactions as inputs for both the PNFIFO-BC phase and the consensus phase, rather
than using the actual transactions. Additionally, output transactions are organized into epochs, and each epoch is processed sequentially. The protocol does not handle transactions individually but rather decides which transactions are output in batches for each epoch, where the batch size could be linear and denoted as $K = \mathcal{O}(n)$.

Furthermore, to mitigate the impact of a “downgrade attack”, we cannot solely rely on that a transaction $tx$ can be output if it has been received by all correct processes. It is possible for a malicious process to selectively send $tx$ to only a few correct processes, thereby inflating communication complexity again. Consequently, for any correct process that receives a transaction $tx$, we must relay it to guarantee that all processes receive it. In our protocol, we leverage vector commitments to help us maintain $\mathcal{O}(n\ell)$ term, where $\ell$ represents the size of the input. However, if we were to naively employ vector commitments, it would enable malicious processes to flood the PNFIFO-BC1 instances and the transaction relay procedure with redundant messages, where no correct process can recover the actual transactions corresponding to these vector commitments. To address this concern, we introduce a local vector commitment buffer called $buf$. In the PNFIFO-BC1 instances, messages are only processed if they belong to this buffer $buf$ (verified by a predicate $Q$). Additionally, for any relayed message from process $p_i$, it is permitted to proceed only if all of $p_i$’s previously sent messages have been verified, i.e., all actual transactions corresponding to the previously sent vector commitments have been received. Moreover, for any vector commitments determined through two consensus protocols, we can ensure that all correct processes will be able to receive these corresponding transactions at the end of the finalization phase without increasing the cost of communication complexity, regardless of whether the client’s transactions were initially received by just one correct process or by $\mathcal{O}(n)$ correct processes.

### 7.2 Transaction Sequencing

In this section, we introduce the transaction sequencing algorithm. In the algorithms presented in this section, statements with blue comments generate $M_i$, which serves as the input for $\text{Consensus}_1$, in Algorithm 3 (the collected $M_i$ is marked with brown color in line 44). Statements with orange comments generate $store$, which constitutes part of the input for $\text{Consensus}_2$ in Algorithm 3 (the generated $store$ set is marked with brown color in line 51). Specifically, the input of $\text{Consensus}_2$ contains $n - f$ $store$ messages. The statements with red comments serve as a defense against downgrade attacks. Further details can be found in Algorithm 2. The transaction sequencing protocol comprises the five following phases.

1. **Broadcast transaction:** (lines 1-2). When a process $p_i$ receives a new transaction $tx$ from a client, $p_i$ submits $tx$ to the SMRFS protocol using the SMRFS-broadcast method (line 1). This allows $p_i$ to request a set of sequence numbers for its transaction $tx$ (line 2).

2. **Assign sequence number:** (lines 3-14, 24-29). Upon receiving the request ($\text{SEQ-REQUEST}, tx$) for the first time (line 4), process $p_i$ assigns a sequence number $s$ to transaction $tx$ and encodes $tx$ as a vector $\{m_1, m_2, \ldots, m_n\}$. Subsequently, a vector commitment $vc_{tx}$ is generated using a vector commitment primitive. To safeguard against malicious processes sending useless messages in the PNFIFO-BC1 phase, $vc_{tx}$ is added to the local vector commitment buffer $buf$. After adding $vc_{tx}$ to $buf$, process $p_i$ sends a DIFFUSION message to all processes to ensure that all correct processes also receive transaction $tx$. This step is crucial to satisfy the condition in line 57 of Algorithm 3, which is necessary for the second consensus protocol (see Algorithm 3).

Following this, $p_i$ initiates PNFIFO-BC1[$s$] with $vc_{tx}$ as input. The rationale behind this step is to defend against malicious processes sending redundant messages to the entire network, which could result in an excessive increase in communication complexity. Finally, $p_i$ multicasts a SEQ-RESPONSE message containing $vc_{tx}$, its signature, sequence number $s$, and an ENDORSE message that carries $vc_{tx}$ and its threshold signature. If $p_i$ is the sender of transaction $tx$, it broadcasts $S[vc_{tx}]$, which is a set of $2f + 1$ sequence numbers collected for median value computation (lines 24-29).
Algorithm 2  Transaction Sequencing, (code for process $p_i$)

Initialize: $seq \leftarrow \{1\}$; $buf \leftarrow \{\}$  \hfill $\triangleright$ sequence number and local vector commitment buffer
Let: $\{P_{NIFO-BC1_j}\}_{j \in \mathbb{N}}$ refer to $n$ instances, the external function $Q$ as follows: $Q(x) \equiv 1$ if $x \in buf$

1: function SMRFS-broadcast($tx$) \hfill $\triangleright$ once receiving a $tx$ from client
2: multicast (SEQ-REQUEST, $tx$) \hfill $\triangleright$ request a sequence number for $tx$
3: upon receiving (SEQ-REQUEST, $tx$) do
4: \hspace{1em} if this transaction $tx$ ($\neq \bot$) is received for the first time do \hfill $\triangleright$ perceive $tx$
5: \hspace{2em} $s \leftarrow seq$ \hfill $\triangleright$ assign sequence number to $tx$
6: \hspace{2em} seq. $\leftarrow$ seq. + 1 \hfill $\triangleright$ increment local sequence
7: \hspace{2em} $\{m_1, \ldots, m_n\} \leftarrow Enc(tx, n, f + 1)$ \hfill $\triangleright$ encode
8: \hspace{2em} $vc_{tx} \leftarrow VecCom(m_1, \ldots, m_n)$; $buf \leftarrow buf \cup \{vc_{tx}\}$ \hfill $\triangleright$ vector commitment
9: \hspace{2em} for each $k \in [n]$ do \hfill $\triangleright$ generate proof for fragment
10: \hspace{3em} $\pi_k \leftarrow Open(vc_{m_k}, k)$ \hfill $\triangleright$ ensure all processes receive $tx$
11: \hspace{2em} send (DIFFUSION, $s$, $vc_{tx}$, $m_k$, $k$, $\pi_k$) to $p_k$ \hfill $\triangleright$ see algorithm 1, $vc_{tx}$ of $tx$ is the input
12: PNFIFO-BC1_i(s)($vc_{tx}$) \hfill $\triangleright$ send signed sequence
13: \hspace{2em} $\sigma_{tx, s} \leftarrow \text{Sign}(vc_{tx} \parallel s, sk_i)$; multicast (SEQ-RESPONSE, $vc_{tx}$, $s$, $\sigma_{tx, s}$) \hfill $\triangleright$ send ENDORSE of $vc_{tx}$
14: \hspace{2em} $\sigma_{tx,s} \leftarrow TS.VrfShare_{f+1}(vc_{tx} , sk_i)$; multicast (ENDORSE, $vc_{tx}$, $\sigma_{tx,s}$) \hfill $\triangleright$ send ENDORSE of $vc_{tx}$
15: upon receiving (DIFFUSION, $s$, $vc_{m_j,i}$, $\pi_i$) from $p_j$ for the first time do \hfill $\triangleright$ DIFFUSION message
16: \hspace{2em} if VerifyOpen($vc_{m_j,i}, \pi_i, \pi_j$) = 1 do \hfill $\triangleright$ verify message
17: \hspace{3em} wait $\log_2(s - 1) = (vc_{s'}, \sigma')$ \hfill $\triangleright$ avoid downgrading attack
18: \hspace{3em} wait the corresponding $tx'$ has received s.t. $vc' = VecCom(Enc(tx', n, f + 1)$ do \hfill $\triangleright$ SPREAD message
19: \hspace{3em} multicast (SPREAD, $s$, $vc_{m_j,i}$, $\pi_i$) if (SPREAD, $s$, $\ast$, $\ast$) has not been sent yet \hfill $\triangleright$ SPREAD
20: upon receiving (SPREAD, $vc_{m_j,i}$, $\pi_i$) from process $p_j$ with $vc$ for the first time do \hfill $\triangleright$ $P_i$ is the sender of $tx$
21: \hspace{2em} if $P_i$ is the sender of (SEQ-REQUEST, $tx$) do \hfill $\triangleright$ $P_i$ is the sender of $tx$
22: \hspace{3em} send (SEQ-RESPONSE, $vc_{tx}$, $s$, $\sigma_{tx,s}$) from $p_j$ do \hfill $\triangleright$ verify signature is valid
23: \hspace{3em} $\sigma_{tx,s} \leftarrow \text{Sign}(vc_{tx} \parallel s, sk_i)$; multicast (SEQ-RESPONSE, $vc_{tx}$, $s$, $\sigma_{tx,s}$) \hfill $\triangleright$ collect sequence numbers for $tx$
24: \hspace{3em} $\sigma_{tx,s} \leftarrow TS.VrfShare_{f+1}(vc_{tx} , sk_i)$; multicast (ENDORSE, $vc_{tx}$, $\sigma_{tx,s}$) \hfill $\triangleright$ collected $2f + 1$ sequence for $tx$
25: \hspace{2em} multicast (ORDER-REQUEST, $vc_{tx}, S[vc_{tx}]$) \hfill $\triangleright$ multicast $S[vc_{tx}]$ to $p_j$ \hfill $\triangleright$ $P_i$ is the sender of $tx$
26: \hspace{2em} if $\| S[vc_{tx}] \| = 2f + 1$ and $\| S[vc_{tx}] \| = 2f + 1 \land \forall (j, s, \sigma_{tx,s}) \in S[vc_{tx}]$, $\text{Vrf}(vc_{tx} \parallel s, \sigma_{tx,s}, pk_i) = 1$ do \hfill $\triangleright$ sign $\pi_{tx}$ with $p_i$'s threshold sk
27: \hspace{3em} $\pi_{tx} \leftarrow \text{Median}(S[vc_{tx}])$ \hfill $\triangleright$ pick up the median value of $S[vc_{tx}]$
28: \hspace{3em} $\sigma_{seqtx} \leftarrow TS.VrfShare_{f+1}(vc_{tx}, \pi_{tx}, sk_i)$ \hfill $\triangleright$ sign $\pi_{tx}$ with $p_i$'s threshold sk
29: \hspace{3em} send (SEQ-MEDIAN, $vc_{tx}$, $\pi_{seqtx}$) to $p_j$ \hfill $\triangleright$ send proof to all
30: upon receiving (ORDER-REQUEST, $vc_{tx}, S[vc_{tx}]$) from $p_j$ do \hfill $\triangleright$ $P_i$ is the sender of $tx$
31: \hspace{2em} if $\| S[vc_{tx}] \| = 2f + 1 \land \forall (j, s, \sigma_{tx,s}) \in S[vc_{tx}]$, $\text{Vrf}(vc_{tx} \parallel s, \sigma_{tx,s}, pk_i) = 1$ do \hfill $\triangleright$ verify threshold sign
32: \hspace{3em} $\pi_{tx} \leftarrow \text{median}(S[vc_{tx}])$ \hfill $\triangleright$ collect threshold sign for median of $tx$
33: \hspace{3em} $\sigma_{seqtx} \leftarrow TS.VrfShare_{f+1}(vc_{tx}, \pi_{tx}, sk_i)$ \hfill $\triangleright$ $\Sigma$ can verify the median $\pi_{tx}$
34: \hspace{3em} multicast (FINAL-MEDIAN, $vc_{tx}, \pi_{seqtx}, \Sigma$) \hfill $\triangleright$ send median proof to all
35: upon receiving (FINA, $vc_{tx}$, $\pi_{tx}$, $\Sigma$) and $TS.VrfShare_{f+1}(vc_{tx}, \pi_{tx}, \Sigma) = 1$ do \hfill $\triangleright$ $add$ into $M_i$, to be input in line 54 in Alg.2
36: \hspace{2em} if $vc_{tx} \notin VCledger$ do \hfill $\triangleright$ $add$ into $M_i$, to be input in line 54 in Alg.2
37: \hspace{3em} $M_i \leftarrow M_i \cup (vc_{tx}, \pi_{tx}, \Sigma)$ \hfill $\triangleright$ add into $M_i$, to be input in line 54 in Alg.2
38: \hspace{2em} upon receiving (ENDORSE, $vc_{tx}$, $\sigma_{tx,s}$) from process $p_j$ and $TS.VrfShare_{f+1}(vc_{tx}, \sigma_{tx,s}) = 1$ do \hfill $\triangleright$ collect threshold sign for $vc_{tx}$
39: \hspace{3em} $\sigma \leftarrow TS.VrfShare_{f+1}(vc_{tx}, E[vc_{tx}])$ \hfill $\triangleright$ $\sigma$ can verify the $vc_{tx}$
40: \hspace{3em} $\sigma \leftarrow TS.VrfShare_{f+1}(vc_{tx}, E[vc_{tx}])$ \hfill $\triangleright$ $\sigma$ can verify the $vc_{tx}$
41: \hspace{3em} if $store[s] = \bot$ do \hfill $\triangleright$ s is sequence number
42: \hspace{4em} $store[s] \leftarrow (vc_{tx}, \sigma)$ \hfill $\triangleright$ generate $store[s]$ (to be input in line 59 in Alg.2)
3. **Diffusion phase:** (lines 15-23). Upon receiving a (DIFFUSION, \( s, \text{vc}, m_i, i, \pi_i \)) message from process \( p_j \) (line 15), process \( p_i \) checks the validity of the message. If the message is valid, and \( p_i \) has received the \( \text{Log}1_j[s-1] = (\text{vc}', \sigma') \) from PNFIFO-BC\( 1_j \), along with one transaction \( tx' \) that corresponds to \( \text{vc}' \), then \( p_i \) forwards the message to all processes using a SPREAD message, but only if \( p_j \) has not done so before. This step is important to ensure that communication complexity does not increase. The reason is as follows:

First, we ensure that when any correct process receives a transaction, all other correct processes also receive it. This assurance is based on the fact that correct processes broadcast their received transactions to all other processes.

Second, it is possible for malicious processes to attempt to send an unlimited number of transactions. However, even if up to \( f \) correct processes forward invalid transactions, these transactions do not affect the final output of the protocol. The lines 17-18 prevent the communication complexity from blowing up.

4. **Decide median:** (line 30-41). Upon receiving a valid set \( S[\text{vc}_{tx}] \) (line 30), processes send back a threshold signature share of the median value of \( S[\text{vc}_{tx}] \) to the sender \( p_x \). Once \( p_i \) (if \( p_i \) is the sender) has collected at least \( f + 1 \) threshold signature shares for the median value \( \bar{s} \) of \( S[\text{vc}_{tx}] \) (line 39), it combines these shares into a full proof \( \Sigma \) and broadcasts \( (\text{FINAL}, \text{vc}_{tx}, \bar{s}, \Sigma) \) to all processes.

5. **Add ordered transaction to submission buffer and generate store[s] for sequence number s:** (line 42-51). When a correct process \( p_i \) receives a valid message \( (\text{FINAL}, \text{vc}_{tx}, \bar{s}, \Sigma) \), if the corresponding transaction has not yet been delivered, it appends \( (\text{vc}_{tx}, \bar{s}, \Sigma) \) to its submission buffer \( M_i \) only if \( (\text{vc}_{tx}, *, *) \) has not been added to \( M_i \) previously (line 44). Additionally, whenever a correct process \( p_i \) receives \( f + 1 \) valid messages \( (\text{ENDORSE}, \text{vc}_{tx}, \sigma) \) from distinct processes, and if it has assigned a sequence number \( s \) for the corresponding \( tx \) of \( \text{vc}_{tx} \), it generates \( \text{store}[s] \) for the corresponding \( tx \).

For any given \((\text{vc}_{tx}, \bar{s}_{tx}, \Sigma)\), the size of \((\text{vc}_{tx}, \bar{s}_{tx}, \Sigma)\) is \( O(\lambda) \). A valid \( \Sigma \) signifies that \((\text{vc}_{tx}, \bar{s}_{tx})\) has been signed by at least one correct process, which, in turn, implies that at least \( f + 1 \) correct processes have sent \((\text{SEQ-RESPONSE}, \text{vc}_{tx}, s, \sigma_{tx,s})\). Consequently, at least one correct process, denoted as \( p_i \), must possess a sequence number \( s \geq \bar{s}_{tx} \) (cf. Lemma 4.1 of [23]). Furthermore, the DIFFUSION operation is always invoked when a new transaction is received (lines 9-11), ensuring that all correct processes can receive at least \( \bar{s}_{tx} \) transactions (lines 15-23). Consequently, all correct processes will multicast an ENDORSE message for \( \text{vc}_{tx} \). Therefore, for any correct process \( p_i \), once \( p_i \) receives \( tx \) and assigns a sequence number \( s \) to it, it is guaranteed to obtain a non-null \( \text{store}[s] \).

Remark: In our protocol, we can also add one step to ensure that the input transaction always satisfies “external validity” as defined in MVBA [1,15]. If a process sends an invalid transaction (as detected in lines 4 and 18), it is possible to reject any future message from that process.

### 7.3 Transaction Consensus

In this section, we introduce the construction of the consensus phase, which is further elaborated in Algorithm 3. For simplicity, we represent the set \( \{\text{Log}_j[1:s_t-1]\} \) \( \in \mathcal{T}_e \) as \( \text{Logs}[s_t-1, T_e] \), where \( \text{Log}_j[a:b] := \{\text{Log}_j[a], \text{Log}_j[a+1], \cdots, \text{Log}_j[b]\} \). Additionally, we use \( \text{store}[a:b] \) to represent the set \( \{\text{store}[a], \text{store}[a+1], \cdots, \text{store}[b]\} \). Furthermore, we use \( \text{FinalLedger} \) to represent the delivered transactions, and \( \text{VCLedger} \) to represent the vector commitments of the corresponding delivered transactions. The consensus protocol is composed of the five following phases.

1. **Consensus**: (lines 52-54). When the submission buffer \( M_i \) of a process \( p_i \) reaches size \( K \), which means \( p_i \) has received \( K \) undelivered transactions with order sequence numbers, and if \( p_i \) has not yet submitted in the current epoch, it inputs \( M_{i,e} \) to \( \text{Consensus}_1[e] \).
Algorithm 3: Consensus for epoch $e$, (code for process $p_i$)

Initialize: epoch $e \leftarrow 1$  \hspace{1cm} $\triangleright$ consensus epochs
\cite{PNFIFO-BC}_2 \{\} \in [n] refers to $n$ instances of PNFIFO-BC  \hspace{1cm} $\triangleright$ PNFIFO-BC instances
$\forall x$, the predicate $Q_1(x)$ for Consensus$_1[e]$ holds if all of the following conditions hold:  \hspace{1cm} $\triangleright$
predicate $Q$

\begin{enumerate}
\begin{itemize}
\item[(1):] $x = K$ and $x := \{(vtx, \bar{tx}, \Sigma)\}_K$ and vtx $\notin$ VCLedger
\item[(2):] $\forall(vtx, \bar{tx}, \Sigma) \in x$, TS.Vfy$_{2f+1}(vtx, \bar{tx}, \Sigma) = 1$
\item[(3):] no two distinct $(vtx, \bar{tx}, \Sigma) \in x$ share the same vtx
\end{itemize}
$\forall x$, the predicate $Q_2$ for Consensus$_2[e]$ holds if all of the following conditions hold:

\begin{enumerate}
\begin{itemize}
\item[(1):] $x = 2f + 1$ and $x := \{(j, e, H_{j,e}, Proof_{j,e})\}_{2f+1}$
\item[(2):] $(j, e, H_{j,e}, Proof_{j,e})$ is valid for $\forall(j, e, H_{j,e}, Proof_{j,e}) \in x$
\item[(3):] no two distinct store$s[\tau]$ := $(vtx, \Sigma) = 1$
\item[(4):] no two distinct store$s[\tau]$ := $(vtx, \Sigma) \in H_{j,e}$ share the same vtx
\end{itemize}
\end{enumerate}

52: \textbf{upon} $|M_i| \geq K$ and $p_i$ has not submitted in epoch $e$ do  \hspace{1cm} $\triangleright$ with $K = O(n)$
53: \quad $M_i,e \leftarrow \text{argmin}_{M_i,M_i,|M_i| = K} \sum_{(vtx,\bar{tx},\Sigma) \in M_i}(s)$  \hspace{1cm} $\triangleright$ $K$ elements in $M_i$ with lowest sequence numbers
54: \quad \text{invoke} Consensus$_1[e]$ with $M_i,e$ as input  \hspace{1cm} $\triangleright$ invoke Consensus$_1[e]$ in Alg.4
55: \quad \text{wait} Consensus$_1[e]$ outputs $M_e$ do  \hspace{1cm} $\triangleright$ $M_e$ is the output of Consensus$_1$: $M_e$ is utilized in line 77 in Alg.4
56: \quad \quad let $h_e := \text{MAX}\{\bar{tx}\}$ in all $(vtx, \bar{tx}, \Sigma) \in M_e$  \hspace{1cm} $\triangleright$ max order seq number in $M_e$
57: \quad \quad \text{upon} store$k[\tau] \neq 0$ for $\forall h_{e-1} + 1 \leq k \leq h_e$ do
58: \quad \quad \quad let $H_{j,e} := \text{store}[h_{e-1} + 1 : h_e]$  \hspace{1cm} $\triangleright$ prepare the input for consensus$_2$
59: \quad \quad \quad \quad \text{PNFIFO-BC}$_2[e]$($H_{j,e}$)  \hspace{1cm} $\triangleright$ invoke PNFIFO-BC2 to feed Consensus$_2$ below
60: \quad \text{wait} PNFIFO-BC2$_j[e]$ outputs ($H_{j,e}, Proof_{j,e}$) for any $j \in P$ do  \hspace{1cm} $\triangleright$ 2f + 1 outputs from PNFIFO-BC2
61: \quad \quad if $(j, e, H_{j,e}, Proof_{j,e})$ is valid do
62: \quad \quad \quad $S_{i,e} \leftarrow S_{i,e} \cup (j, e, H_{j,e}, Proof_{j,e})$
63: \quad \quad \quad \text{if} $|S_{i,e}| = 2f + 1$ do  \hspace{1cm} $\triangleright$ 2f + 1 Log$_h$
64: \quad \quad \quad \quad \text{invoke} Consensus$_2[e]$ with $S_{i,e}$ as input  \hspace{1cm} $\triangleright$ invoke Consensus$_2[e]$ in Alg.5
65: \quad \text{wait} Consensus$_2[e]$ outputs $S_{e}$ do
66: \quad \quad \text{for} $\forall(j, e, H_{j,e}, Proof_{j,e}) \in S_{e}$ do
67: \quad \quad \quad $Te \leftarrow Te \cup j$  \hspace{1cm} $\triangleright$ indexes set
68: \quad \quad \quad \text{if} the newest output PNFIFO-BC2$_j$ is ($H_{j,s_r}, Proof_{j,s_r}$) and $s_r < e + 1$ do
69: \quad \quad \quad \quad  \$help \leftarrow \$help \cup (j, s_r + 1)  \hspace{1cm} $\triangleright$ see algorithm 5
70: \quad \text{CallHelp}(e, S_{help}, e - 1)$ do
71: \quad \text{for} $\forall j \in T_e$ do
72: \quad \quad \text{wait} until Log$_2[1 : e]$ := ($H_{j,1}, H_{j,2}, \cdots, H_{j,e}$) have been received
73: \quad \quad \text{for} $\forall k \in [e]$, $\forall storek[s] := (vtx, \Sigma) \in H_{j,k}$ do
74: \quad \quad \quad Log$_{h}[s] \leftarrow \text{vtx}  \hspace{1cm} $\triangleright$ Log$_j$ is a part of Logs: Logs is utilized in line 82 in Alg.4
75: \quad \quad \quad if vtx $\notin$ VCLedger do
76: \quad \quad \quad \quad Pending$_e \leftarrow$ Pending$_e$ $\cup$ vtx  \hspace{1cm} $\triangleright$ add into Pending$_e$
\end{itemize}

2. Wait for enough outputs from distinct PNFIFO-BC2$_j[e]$ instances: (lines 55-59). Upon receiving the output $M_e$ from Consensus$_1$, the process checks the maximum order sequence number, denoted as $h_e$, among $(\bar{tx})$, where $(vtx, \bar{tx}, \Sigma) \in M_e$.

Afterward, $p_i$ waits for all store$k[\tau]$ to be received, where $1 \leq k \leq h_e$. Once these are all received, $p_i$ invokes PNFIFO-BC2 to disseminate this received transaction history to all processes. The underlying motivation for this step is to efficiently distribute its received transaction history.
throughout the entire network. This particular step plays a critical role in the finalized output phase when identifying potential transactions that should be included in the output.

3. **Consensus** (lines 60-64). Once at least \( n - f \) distinct PNFIFO-BC2[e] instances have output, **Consensus** is invoked to decide which Logs will be used to determine the potential transactions.

4. **Recover missed Log blocks** (lines 65-70). When **Consensus**[e] outputs \( S_e \), the indexes are denoted as \( T_e \) (lines 65-67). Process \( p_i \) iterates through each index \( k \) of \( T_e \). If the newest output of PNFIFO-BC2k is \( (H_{k,e}, \text{Proof}_{k,e}) \) and \( s_k < e - 1 \), the process adds \( (k, s_k + 1) \) to \( S_{help} \) (lines 68-69). The process then invokes the CallHelp function to recover all missed Log2 values (line 70).

5. **Generate Log** and **Pending** (lines 71-76). Once all values have been received, \( p_i \) initiates the generation of Log based on the corresponding \( store \) value (line 74). Subsequently, any values that do not belong to VCLedger are included in Pending (lines 75-76).

For any Log2k[e]: \( (H_{k,e}, \text{Proof}_{k,e}) \), based on the PNFIFO-BC-Notarizability property, at least \( f + 1 \) correct processes have received Log2k[e - 1] := \( (H_{k,e-1}, \text{Proof}_{k,e-1}) \). As a result, all correct processes receive Log2k[1 : e - 1] via CallHelp function in Algorithm 5.

---

**Algorithm 4 Output Finalization for epoch \( e \), (code for process \( p_i \))**

77: \( \text{parse } M_e := \{ (\text{vc}_{tx,e}, \pi_{tx,e}, \Sigma) \} \); for all \( (\text{vc}_{tx,e}, \pi_{tx,e}, \Sigma) \in M_e \): \( S' \leftarrow S' \cup \pi_{tx,e} \) \( \triangleright M_e \) is the output of **Consensus**

78: let \( S' = [\pi_{tx,1}, \pi_{tx,2}, \ldots, \pi_{tx,|S'|}] \), where \( \pi_{tx,i} \) is the \( i \)-th smallest value in the set \( S' \) and \( \pi_{tx,e} = h_e \)

79: let \( M'_e = \{ \text{vc}_{tx,1}, \text{vc}_{tx,2}, \ldots, \text{vc}_{tx,|S'|} \} \)

80: for \( \text{vc}_{tx,1} \in M'_e \) and \( \text{vc}_{tx,2} \) picks in order do  \( \triangleright \) first \( \text{vc}_{tx,1} \), then \( \text{vc}_{tx,2} \), and so on

81: for \( \text{vc}_{tx,1} \in \text{Pending} \) and \( \text{vc}_{tx,2} \notin M'_e \) do  \( \triangleright \) selecting \( \text{vc}_{tx,1} \)

82: if \( \text{vc}_{tx,1} \) appears at least \( f + 1 \) times in \( \text{Logs}(\pi_{tx,1} - 1, T_e) \) do  \( \triangleright \) Logs were decided by **Consensus**

83: for \( j \in T_e \) do

84: if \( \text{Log}_j[k] := \text{vc}_{tx,1} \) and \( k < \pi_{tx,1} \) do

85: \( \text{seq}[\text{vc}_{tx,1}] \leftarrow \text{seq}[\text{vc}_{tx,1}] \cup k \)

86: \( \text{seq}[\text{vc}_{tx,1}] \leftarrow \text{Sort}(\text{seq}[\text{vc}_{tx,1}]) \) \( \triangleright \) sort in ascending order

87: let the \( (f + 1) \)-th element of \( \text{seq}[\text{vc}_{tx,1}] \) as the median value, and denote \( s_{tx,1} \)

88: \( S[\text{vc}_{tx,1}] \leftarrow S[\text{vc}_{tx,1}] \cup (\text{vc}_{tx,1}, s_{tx,1}) \)

89: \( S[\text{vc}_{tx,1}] \leftarrow \text{Sort}(S[\text{vc}_{tx,1}]) \) \( \triangleright \) sort in ascending order of \( s_{tx,1} \)

90: Value[\text{vc}_{tx,1}][i] \leftarrow \text{vc}_{tx,1} \) if \( S[\text{vc}_{tx,1}][i] = (\text{vc}_{tx,1}, s_{tx,1}) \)

91: \( M'_e \leftarrow \{ \text{vc}_{tx,1}, \text{vc}_{tx,2}, \ldots, \text{vc}_{tx,|S'|} \} \)

92: for all \( \text{vc}_{tx,1} \in M'_e \) do \( \triangleright \) extract the corresponding \( tx \)

93: \( \text{wait} \) until \( tx \) has been received such that \( \text{vc}_{tx,1} = \text{VecCom}(\text{Enc}(tx, n, f + 1)) \) do  \( \triangleright \) extract \( tx \)

94: FinalLedger[c][i][t] \leftarrow tx \) if \( M'_e[i] = \text{vc}_{tx,1} \)

95: \( M_i \leftarrow M_i \setminus (\text{vc}_{tx,1}, \ast, \ast) \) for any \( \text{vc}_{tx,1} \in M'_e \)

96: VCLedger \leftarrow VCLedger \cup M'_e \)

97: SMRFS-delivery(FinalLedger[c]) \( \triangleright \) delivery

98: \( e \leftarrow e + 1 \)

---

**7.4 Transaction Finalised Output**

In this section, we present the finalization protocol, depicted in Algorithm 4. The finalization phase consists of the four following phases.
1. **Sorting** $M_e$: (line 77-79). Process $p_i$ sorts the output $M_e$ of $\text{Consensus}_1$ based on the corresponding $\pi_{tx}$ in ascending order, resulting in the sorted set $M_e' = \{v_{tx_{x,1}'}, v_{tx_{x,2}'}, \cdots, v_{tx_{x,y}'}\}$.

2. **Find potential transactions**: (lines 80-91). Process $p_i$ iterates through the elements of $M_e'$. For each $v_{tx_{x,y}'} \in M_e'$, it checks if $v_{tx_{x,y}'} \in \text{Pending}_{x}$ and $v_{tx_{x,y}'} \notin M_e'$. If these conditions are met, it looks for $v_{tx_{x,y}'}$ in $\text{Logs} \{s_{tx_{x,y}} - 1, T_e\}$ and counts how many times it appears. If $v_{tx_{x,y}'}$ appears $f + 1$ times, it records the sequence numbers in $\text{seq}[v_{tx_{x,y}'}]$ and orders them from smallest to largest. The $(f + 1)^{th}$ element of $\text{seq}[v_{tx_{x,y}'}]$ is taken as the median value of $tx_{x,y}'$, and it is denoted as $s_{tx_{x,y}'}$, which becomes the order sequence number of $v_{tx_{x,y}'}$. Afterwards, process $p_i$ adds $(v_{tx_{x,y}'}, s_{tx_{x,y}'})$ into $S[v_{tx_{x,y}'}]$ (lines 80-88). If all potential values before $v_{tx_{x,y}'}$ have been thoroughly searched, the operation $\text{Sort}(S[v_{tx_{x,y}'}])$ is executed to arrange $S[v_{tx_{x,y}'}]$ in ascending order based on $s_{tx_{x,y}'}$. Last, the new values $v_{tx_{x,y}'}$ are inserted into $M_e'$ (lines 89-91).

3. **Extract the corresponding transaction**: (lines 92-94). For any element (vector commitment) $v_{tx_{x,y}} \in M_e'$, if the transaction $tx_{x,y}$ corresponding to $v_{tx_{x,y}}$ has been received, then it is recorded in $\text{FinalLedger}[\text{e}[i]]$.

4. **Deliver transaction**: (lines 95-98). After extracting all corresponding transactions, the process updates its submission buffer $M_i$ and the VCledger (lines 95-96). Then, it delivers the finalized output $\text{FinalLedger}[e]$ for epoch $e$ (line 97) and proceeds to the next epoch (line 98).

**Algorithm 5 CallHelp daemon and Help daemon, code for process $p_i$**

```
/* CallHelp daemon */

external function CallHelp(e, $S_{\text{help}}$, $s_d$) ➤ recovery Log
99:   multicast CALLHELP(e, $S_{\text{help}}$, $s_d$)
100: upon receiving message HELP(e, $S_j$) from process $p_j$ for the first time do
101:   for any $(k, s_r + 1) \in S_{\text{help}}$ do
102:     if $S_j[k] := (v_{ck}, m_j, j, \pi_j) \neq \bot$ do
103:       if VerifyOpen($v_{ck}, m_j, j, \pi_j$) = 1 do
104:         $F[v_{ck}] \leftarrow F[v_{ck}] \cup \{j, m_j\}$
105:       if $|F[v_{ck}]| = f + 1$ do
106:         $\text{Blocks}_k \leftarrow \text{Dec}(F[v_{ck}])$; parse $\text{Blocks}_k := \text{Log}_k[s_r + 1 : s_d]
107:         return $\text{Log}_k[s_r + 1 : s_d]

/* Help daemon */

Help: It is a daemon process that can read the output Log of PNFIFO-BC, and it listens to the down below event:

108: upon receiving CALLHELP(e, $S_{\text{help}}$, $s_d$) from process $p_j$ for the first time do
109:   for any $(k, s_r + 1) \in S_{\text{help}}$ do
110:     if PNFIFO-BC2_k[$s_d$] outputs $(v_{ck}, \sigma_{ck})$ do ➤ $\text{Log}_2_k[s_d] \neq \emptyset$
111:       let $\text{Blocks}_k \leftarrow \text{Log}_k[s_r + 1 : s_d]
112:       $\{m_1, \cdots, m_n\} := \text{Enc}(\text{Blocks}_k, n, f + 1)$ ➤ encode
113:       $v_{ck} \leftarrow \text{VecCom}(m_1, \cdots, m_n)$ ➤ vector commitment
114:       $\pi_i \leftarrow \text{Open}(v_{ck}, m_i, i)$ ➤ generate proof for fragment
115:       let $S_i := \{\bot, \cdots, \bot\}$ and $|S_i| = n
116:       $S_i[k] \leftarrow (v_{ck}, m_i, i, \pi_i)$ ➤ record the corresponding value
117:       send HELP(e, $S_i$) to $p_j$

For any $v_{tx_{x,y}} \in M_e'$, at least one correct process has received the corresponding transaction, ensuring that all processes receive the corresponding $tx$ (with the help of DIFFUSION procedure).
```
8 Protocol Analysis

In this section, we first examine the security of SMRFS and follow by a discussion on its communication complexity.

8.1 Security Analysis

In this section, we first prove that the SMRFS protocol satisfies all the properties of state machine replication protocol (cf. Definition 1), and then that its output satisfies OL (cf. Definition 3).

Lemma 2. If a correct process \( p_i \) receives a message \((\text{SEQ-REQUEST}, tx)\), then all correct processes eventually receive transaction \( tx \).

Proof. Whenever a correct process \( p_i \) receives a message \((\text{SEQ-REQUEST}, tx)\) (line 3), it assigns a sequence number \( s \) to \( tx \). Subsequently, it divides \( tx \) into fragments using an erasure code scheme and creates a vector commitment \( vc_{tx} \) for these fragments. Process \( p_i \) subsequently creates \( n \) openings \( \pi_k \) for each fragment and sends its respective fragment \( m_k \) along with an opening \( \pi_k \) to each process \( p_k \) using a DIFFUSION message (line 11). Finally, \( p_i \) submits \( vc_{tx} \) to PNFIFO-BC1\([s]\). Moreover, for any correct process, it invokes PNFIFO-BC1 with input \( vc_{tx} \) only if it has already received the corresponding \( tx \).

Consequently, when a correct process \( p_j \) receives a DIFFUSION message (line 15) from another correct process, it proceeds to multicast a SPREAD message containing the fragment \( m_j \) that it has received. This occurs because all previously sent vector commitments corresponding to transactions can be received. As a result, each correct process is ensured to receive at least \( n - f \) fragments of \( tx \). This enables each process to reconstruct \( tx \) and locally generate a \((\text{SEQ-REQUEST}, tx)\) message.

Lemma 3. If a correct process \( p_i \) receives a message \((\text{SEQ-REQUEST}, tx)\), then \( p_i \) will assign a sequence number \( s \) to \( tx \) and \( store[s] \) will eventually become non-null.

Proof. According to lemma 2, if a correct process \( p_i \) receives a message \((\text{SEQ-REQUEST}, tx)\), then all correct processes will be able to receive \( tx \). Based on the code, once a correct process \( p_i \) receives \( tx \), it will assign a sequence number \( s \) to \( tx \) and multicast an ENDORSE message for \( vc_{tx} \). Therefore, for any correct process \( p_i \), upon receiving \( tx \), it can receive at least \( f + 1 \) valid ENDORSE messages from distinct processes. Consequently, it is guaranteed that \( p_i \) will obtain a non-null \( store[s] \).

Lemma 4. The order sequence number \( \pi_{tx} \) decided for a transaction \( tx \) by the SMRFS protocol is upper bounded and lower bounded by the sequence numbers assigned to \( tx \) by correct processes.

Proof. The order sequence number \( \pi_{tx} \) decided for a transaction \( tx \) in the SMRFS protocol is determined using the median value of a set \( S_{tx} \) of signed sequence numbers assigned to \( tx \) by \( 2f + 1 \) distinct processes. Since there can be at most \( f \) Byzantine processes, the value of \( \pi_{tx} \) is guaranteed to be within the range of the sequence numbers assigned to \( tx \) by correct processes. Additionally, the threshold of \( f + 1 \) used when building a threshold signature for \( \pi_{tx} \) ensures that at least one correct process has verified the correctness of the signatures in \( S_{tx} \) and confirmed that \( \pi_{tx} \) is indeed the median of \( S_{tx} \).

Lemma 5. For any valid tuple \((vc_{tx}, \pi_{tx}, \Sigma)\), all correct processes can receive \( \pi_{tx} \) number of transactions.

Proof. Based on the code of Algorithm 2, a valid \( \Sigma \) implies that \((vc_{tx}, \pi_{tx})\) has been signed by at least one correct process. This, in turn, signifies that \( \pi_{tx} \) is the median value among the set of \( 2f + 1 \) sequence numbers that distinct processes assigned to \( tx \). Consequently, it follows that at least one correct process \( p_i \) must possess a sequence number \( s \) such that \( s \geq \pi_{tx} \) (cf. Lemma 4.1 of [23] and lemma 4).

As a result, it also means that \( p_i \) has received at least \( \pi_{tx} \) distinct transactions. Following Lemma 2, all correct processes can also receive \( \pi_{tx} \) number of transactions.
Lemma 6. For any valid tuple \((vc_{tx}, s_{tx}, \Sigma)\), all correct processes \(store[k] \neq \perp\) for \(\forall k \in [s_{tx}]\).

Proof. If the tuple \((vc_{tx}, s_{tx}, \Sigma)\) is valid, then according to Lemma 5, all correct processes can receive \(s_{tx}\) number of transactions. Additionally, based on Lemma 3, for each received \(tx\), all correct processes will assign a sequence number \(s\) to \(tx\), and \(store[s]\) will eventually become non-null. Therefore, with this assumption, all correct processes have \(store[k] \neq \perp\) for all \(k \in [s_{tx}]\).

Lemma 7. If a correct process \(p_i\) receives a message \((seq\text{-}request, tx)\), then \(tx\) is SMRFS-delivered by all correct processes.

Proof. If a correct process \(p_i\) receives a message \((seq\text{-}request, tx)\), then, based on Lemma 2 and Lemma 3, all correct processes will assign a sequence number \(s\) to \(tx\), and \(store[s] \neq \perp\) will eventually hold for all processes. We can consider the following two cases:

1. If \(vc_{tx}\) along with a median value is added to the submission buffers and output in Consensus1, then following the consensus agreement, all correct processes will output \(vc_{tx}\).
2. Otherwise, since each correct process has received \(tx\) and assigned a sequence number \(s\) to it, all correct processes will have a \(store[s] = (vc_{tx}, \sigma)\). Consequently, once the maximum order sequence number determined by Consensus1 surpasses the maximum sequence numbers assigned by all correct processes, then the output of Consensus2 must include at least \(f + 1\) correct processes’ Log2s, and all of these Log2s will contain \(vc_{tx}\). According to the code of Algorithm 4, \(vc_{tx}\) will then be output.

In both cases, once \(vc_{tx}\) is determined to be output, it is guaranteed that the corresponding \(tx\) will be received by all correct processes. This is because both cases imply that at least one correct process possesses \(vc_{tx}\). When at least one correct process has \(vc_{tx}\), it implies that at least one correct process has received \(tx\). Then, according to Lemma 2, it is ensured that this transaction is received by all correct processes. As a result, all correct processes can SMRFS-deliver \(tx\).

Theorem 3. The SMRFS protocol (Algorithm 2, Algorithm 3 and Algorithm 4) implements a State Machine Replication protocol (cf. Definition 1).

Proof. We prove each property separately.

1. SMR-Consistency. During each epoch, the Consensus-Agreement property ensures that correct processes output the same submission buffer from Consensus1. Additionally, in accordance with Lemma 6, all correct processes will have input into Consensus2, and Consensus-Termination ensures that all correct processes have an output. Moreover, the PNFIFO-BC-Total-order and Consensus-Agreement properties guarantee that each correct process outputs the same set of transactions during the finalization phase following Consensus2. Consequently, in each epoch, each correct process SMRFS-delivers the same set of transactions associated with the same order sequence numbers, ordering these transactions in ascending order based on these order sequence numbers. These transactions are deterministically sorted using a lexicographical order in the case of a tie. As a result, all correct processes can SMR-deliver the same set in the same epoch. Furthermore, according to the code, they will SMR-deliver in epoch \(e\) only if SMR-delivery occurred in epoch \(e - 1\). The ordering rule also determines that if a transaction \(tx_1\) is SMRFS-delivered by a correct process before a transaction \(tx_2\), then all other correct processes will also SMRFS-deliver \(tx_1\) before \(tx_2\).

When a correct process SMRFS-delivers a transaction \(tx\), based on the previous analysis, all correct processes also SMRFS-deliver the same transaction \(tx\).

2. SMR-Liveness. If a correct process \(p_i\) SMRFS-broadcasts a transaction \(tx\), then according to the code in Algorithm 2, \(p_i\) multicasts a message \((seq\text{-}request, tx)\) to all processes. Given the network model assumption, all correct processes can receive \((seq\text{-}request, tx)\) within a single...
step and then separately multicast a signed sequence number \( s \) for \( tx \) and a threshold signature share of \( vc_{tx} \) to all processes. As \( 2f + 1 \leq n - f \), \( p_i \) will collect a set \( S_{tx} \) of \( 2f + 1 \) signed sequence numbers for \( tx \). After broadcasting a commitment \( vc_{tx} \) and a set \( S_{tx} \), process \( p_i \) can collect \( f + 1 \) shares for the median value \( \pi_{tx} \) of \( S_{tx} \). Process \( p_i \) then broadcasts a final message.

If \( tx \) has not been SMRFS-delivered, then the communication channel ensures that \( (vc_{tx}, \pi_{tx}, \Sigma) \) is eventually added to the submission buffers of all correct processes; in other words, all correct processes will add \( (vc_{tx}, \pi_{tx}, \Sigma) \) to their submission buffers. Following the consensus validity property, \( vc_{tx} \) is eventually output by Consensus \( i \) if \( vc_{tx} \) has not been output in the finalization phase. According to the code, all correct processes will receive a unique \( tx \) corresponding to \( vc_{tx} \) at the end of the finalization phase.

**Theorem 4.** Our SMRFS protocol satisfy fair separability (Definition 3).

**Proof.** Let \( tx_1 \) and \( tx_2 \) be two transactions that are observed by correct processes with the sets of sequence numbers \( S_1 \) and \( S_2 \), respectively, with \( \max_{s \in S_1}(s) < \min_{s \in S_2}(s) \). First, note that due to Lemma 4, and because \( \max_{s \in S_1}(s) < \min_{s \in S_2}(s) \), for any order sequence numbers \( \pi_1 \) and \( \pi_2 \) output by the protocol for \( tx_1 \) and \( tx_2 \), respectively, we have \( \pi_1 < \pi_2 \).

Considering the following scenarios: (1) if \( tx_1 \) is output during the same epoch as \( tx_2 \), or during an earlier epoch, then we trivially have \( tx_1 \prec tx_2 \); (2) \( tx_1 \) has not yet been output when \( tx_2 \) is output by Consensus \( i \). We thus only need to consider the case (2).

Recall that for a transaction \( tx \), the sequence number \( s \) assigned to \( tx \) by a correct process \( p_i \) corresponds to \( p_i \) has \( store[s] \neq \bot \). Let \( e \) denote the epoch where \( tx_2 \) is output with order sequence number \( \pi_2 \in S_2 \) by Consensus \( i \), and let \( h_e \) denote the highest sequence number among the transactions output by Consensus \( i \). Before starting Consensus \( i \), a correct process \( p_i \) waits until at least \( n - f \) processes have completed their \( e^{th} \) instances of PNFIFO-BC2 (line 64). Then, Consensus \( i \) outputs a set \( T_e \) of \( 2f + 1 \) processes whose respective instances have \( h_e \) store output. Because there are at least \( f + 1 \) correct processes in \( T_e \), \( T_e \) contains the Logs of at least \( f + 1 \) correct processes that have assigned to \( tx_1 \) a sequence number that is less than \( \pi_2 \). As a result, when checking for transactions that should be delivered before \( tx_2 \) (line 82), each correct process sees that transaction \( tx_1 \) should be added to the delivered set in epoch \( e \), and order \( tx_1 \) before \( tx_2 \). Furthermore, considering claims 1 and 2, the SMRFS ensures the preservation of FS when both \( tx_1 \) and \( tx_2 \) adhere to the FS conditions.

### 8.2 Complexity Analysis

In this section, we provide a comprehensive breakdown of the costs associated with our construction.

**Theorem 5.** The communication complexity is \( O(n\ell + \lambda n^2) \) bits per transaction, which is optimal when \( \ell \geq \lambda n \), where \( \ell \) is the size of the transaction and \( \lambda \) is the security parameter.

**Proof.** Based on the pseudo-code for SMRFS, the cost breakdown can be summarized into the following three phases:

1. **Sequencing phase:** This phase involves three main parts. Firstly, the broadcasting of a transaction \( tx \) to all processes to assign a sequence number incurs a communication cost of \( O(n\ell) \) bits. Secondly, according to Theorem 2, the \( n \) concurrent executions of the PNFIFO-BC instances with the vector commitment \( vc \) of \( tx \) \( (|vc| = \lambda) \) as input incurs a communication cost of \( O(n^2\lambda) \) bits. Thirdly, when a transaction \( tx \) is received for the first time, a correct process sends a diffusion message to all other processes. Meanwhile, all correct processes also multicast an endorse message to generate a store for \( tx \). If a correct process receives a valid diffusion message and the previous transactions of the sender have also been received, the process forwards it by multicasting a diffusion message once for \( tx \). The size of the diffusion and spread messages
both are $O(\ell/n + \lambda)$, while the size of the ENDORSE is $O(\lambda)$, so the cost of this part is $O(n\ell/n^2\lambda)$. Besides, we ensure that any output in PNFIFO-BC corresponds to a valid transaction that will eventually be included in the final output. This prevents any wasted communication in the PNFIFO-BC instances. As a result, the communication cost in the sequencing phase is $O(n\ell/n^2\lambda)$ bits per transaction $tx$.

2. **Consensus phase:** In this phase, two Consensus instances and one CallHelp module are involved. The first Consensus instance has an input size of $O(K\lambda) = O(n\lambda)$, and the second Consensus instance has an input size of $O(nk\lambda)$, where $k = h_e - h_{e-1} - 1$. Since the Consensus protocol has $O(n|m| + \lambda n^2)$ communication complexity, where $|m|$ is the size of the input value of Consensus, the total cost of the Consensus instances is $O(\lambda n^2 k)$ for $n$ distinct vector commitments. Regarding the CallHelp module, for a single correct process invoking it, the size of $S_{help}$ is at most $O(n)$. For each element $j$ in $S_{help}$, suppose that $k$ distinct vector commitments must be recovered, resulting in a communication cost of $O(kn\lambda + n\lambda)$. Thus, the total cost per process is $O(kn\lambda + n^2\lambda)$. If $O(n)$ processes need to invoke this CallHelp module, the total cost for at least $k$ vector commitments would be $O(kn^2\lambda + n^3\lambda)$ for the current epoch.

The $k$ distinct vector commitments correspond to $k$ distinct transactions and these transactions will eventually be delivered. If $k < n$, then the communication cost of the $n$ distinct vector commitments is $O(\lambda n^2)$, resulting in $O(\lambda n^2)$ for each vector commitment. If $k \geq n$, then the communication cost of the $k$ distinct vector commitments is $O(\lambda n^2 \lambda)$, which still results in $O(\lambda n^2)$ for each vector commitment. In summary, for any single transaction (vector commitment), the communication cost in this phase is $O(n^2\lambda)$.

3. **Finalization:** During this phase, no further communication between processes is required.

Summing up the communication complexity of all three phases, the communication complexity per transaction for the entire protocol is $O(n\ell + \lambda n^2)$ bits.

9 **Constant Time State Machine Replication with Ordering Linearizability**

From section 7, we observe that the communication complexity is $O(n\ell + \lambda n^2)$ per transaction. However, even if $O(n)$ transactions are input in constant time (each correct process inputs $O(1)$ transactions in constant time), the time complexity reaches up to $O(n)$ in the worst case, unless in the optimistic scenario, where the network is synchronous and all processes are correct, the time complexity is reduced to $O(1)$. The primary reason is that line 57 in algorithm 3 can be influenced by malicious processes. As an illustration, consider a scenario where one malicious process sends a transaction $tx$ to a correct process $p_i$, and $p_i$ assigns a sequence number $n$ to $tx$. Following this, all malicious processes go into an offline state. The execution of lines 15-23 would necessitate $O(n)$ steps to ensure that all correct processes receive the transaction $tx$. This is due to correct processes accepting the forwarded transaction from $p_i$ only if a correct process has already received the output of PNFIFO-BC$_1[n-1]$. Subsequently, each correct process assigns a sequence number and broadcasts an ENDORSE message for $tx$. As a result, $p_i$ requires $O(n)$ steps to generate the corresponding store[$n$]. Consequently, $p_i$ needs $O(n)$ steps to initialize PNFIFO-BC$_2$, resulting in the entire protocol requiring $O(n)$ steps to produce the output.

In this section, we ensure that the time complexity consistently remains in $O(1)$ in any case where $O(n)$ transactions are input in constant time. However, it is important to note that this improvement in time complexity comes at the cost of an increased communication complexity, which rises to $O(n^2\ell + \lambda n^2)$ per transaction.

The core idea is to utilize the actual transactions instead of involving any vector commitments. For any received transactions, every correct process consistently forwards them to all other processes. Nevertheless, we also provide a modified algorithm for SMRFS that has a constant time complexity.
The constant-time SMRFS protocol can be found in Appendix A and is composed of the three algorithms detailed in Algorithm 6, Algorithm 7, and Algorithm 8. The protocol analysis follows a similar approach to the SMRFS protocol discussed in section 7. The communication complexity of the protocol is $O(n^2) + \lambda n^2)$ per transaction, and the time complexity is $O(1)$ when $O(n)$ transactions are input in constant time.

A Instantiation of Constant Time SMRFS

Algorithm 6 Transaction Sequencing, code for process $p_i$

```
Initialize: seq, $\leftarrow 1$; buf $\leftarrow \{\}$  \hspace{1cm} \triangleright$ sequence number and local vector commitment buffer
1: function SMRFS-broadcast($tx$)  \hspace{1cm} \triangleright request a sequence number for $tx$
2:  multicast (SEQ-REQUEST, $tx$)  \hspace{1cm} \triangleright$ upon$ receiving (SEQ-REQUEST, $tx$) do
3:  if this transaction $tx$ ($\neq \bot$) is received for the first time do  \hspace{1cm} \triangleright$ perceive$ $tx$
4:    $s \leftarrow$ seq  \hspace{1cm} \triangleright$ assign sequence number to $tx$
5:    seq, $\leftarrow$ seq + 1  \hspace{1cm} \triangleright$ increment local sequence
6:  multicast (SEQ-REQUEST, $tx$)  \hspace{1cm} \triangleright$ diffusion$ tx$
7:  $\sigma_{tx,s} \leftarrow$ Sign($tx || s, sk_i$); multicast (SEQ-RESPONSE, $tx, s, \sigma_{tx,s}$)
8:  $\sigma_{tx} \leftarrow$ TS.SigShare$_{f+1}(tx, sk_i)$; multicast (ENDORSE, $tx, \sigma_{tx}$)
10: if $P_i$ initiates SMRFS-broadcast($tx$) do
11:  upon receiving (SEQ-RESPONSE, $tx, s, \sigma_{tx,s}$) from $p_j$ do  \hspace{1cm} \triangleright$ verify signature is valid$
12:    \text{if}$ Vrf($tx || s, \sigma_{tx,s}, pk_j$) $\neq 1$ do  \hspace{1cm} \triangleright$ collect sequence numbers for $tx$
13:      $S[tx] \leftarrow$ $S[tx] \cup (j, s, \sigma_{tx,s})$
14:    if $|S[tx]| = 2f + 1$ do  \hspace{1cm} \triangleright$ collected $2f + 1$ sequences for $tx$
15:      multicast (ORDER-REQUEST, $tx, S[tx]$)  \hspace{1cm} \triangleright$ multicast $S[tx]$
16:  upon receiving (ORDER-REQUEST, $tx, S[tx]$) from $p_j$ do  \hspace{1cm} \triangleright$ pick up the median value of $S[tx]$
17:    if $|S[tx]| = 2f + 1 \land \forall (j, s, \sigma_{tx,s}) \in S[tx], Vrf(tx || s, \sigma_{tx,s}, pk_j) = 1$ do  \hspace{1cm} \triangleright$ sign $\bar{s}_{tx}$ with $p_i$’s threshold $sk_i$
18:      $\bar{s}_{tx} \leftarrow$ Median($S[tx]$)
19:      $\sigma_{seqtx} \leftarrow$ TS.SigShare$_{f+1}(tx, \bar{s}_{tx}, sk_i)$  \hspace{1cm} \triangleright$ collect threshold sign for median of $tx$
20:      send (SEQ-MEDIAN, $tx, \bar{s}_{tx}, \sigma_{seqtx}$) to $p_j$  \hspace{1cm} \triangleright$ send median proof to all
21: if $P_i$ initiates SMRFS-broadcast($tx$) do
22:  upon receiving (SEQ-MEDIAN, $tx, \bar{s}_{tx}, \sigma_{seqtx}$) from $p_j$ do  \hspace{1cm} \triangleright$ verify threshold sign$
23:    \text{if}$ TS.VrfShare$_{f+1}(tx, \bar{s}_{tx}, (j, \sigma_{seqtx})) = 1$ do  \hspace{1cm} \triangleright$ collect threshold sign for median of $tx$
24:      $S[\bar{s}_{tx}] \leftarrow S[\bar{s}_{tx}] \cup (j, \sigma_{seqtx})$
25:    if $|S[\bar{s}_{tx}]| = f + 1$ do  \hspace{1cm} \triangleright$ collected $f + 1$ threshold sign for median of $tx$
26:      $\Sigma \leftarrow$ TS.Comb$_{f+1}(tx, \bar{s}_{tx}, S[\bar{s}_{tx}])$  \hspace{1cm} \triangleright$ $\Sigma$ can verify the median $\bar{s}_{tx}$
27:      multicast (FINAL, $tx, \bar{s}_{tx}, \Sigma$)  \hspace{1cm} \triangleright$ send median proof to all
28: upon receiving (FINAL, $tx, \bar{s}_{tx}, \Sigma$) and TS.Vfy$_{f+1}(tx, \bar{s}_{tx}, \Sigma) = 1$ do
29:    if $tx \not\in$ Ledger do  \hspace{1cm} \triangleright$ add into $M_t$
30:      $M_t \leftarrow M_t \cup (tx, \bar{s}_{tx}, \Sigma)$
31: upon receiving (ENDORSE, $tx, \sigma_{tx}$) from process $p_j$ and TS.VrfShare$_{f+1}(tx, (j, \sigma_{tx})) = 1$ do
32:    if assigned a sequence number $s$ for $tx$ do  \hspace{1cm} \triangleright$ collect threshold sign for $tx$
33:      $E[tx] \leftarrow E[tx] \cup (j, \sigma_{tx})$
34:    if $|E[tx]| = f + 1$ do  \hspace{1cm} \triangleright$ collected $f + 1$ threshold sign for $tx$
35:      $\sigma \leftarrow$ TS.Comb$_{f+1}(tx, E[tx])$
36:    if store[$s$] $\neq \bot$ do  \hspace{1cm} \triangleright$ $s$ is sequence number
37:      store[$s$] $\leftarrow$ ($tx, \sigma$)  \hspace{1cm} \triangleright$ generate store[$s$]
```

23
Algorithm 7 Consensus with epoch number $e$, code for process $p_i$

Initialize: epoch $e \leftarrow 1$  \hspace{1cm} $\triangleright$ consensus epochs

\{$\text{PNFIFO-BC}_j\}$ refer to $n$ instances \hspace{1cm} $\triangleright$ initial $n$ PNFIFO-BC instances

the validation of input $x$ in $\text{Consensus}[e]$ is as follows if $x$ satisfies the following conditions:

1. $|x| = K$ and $x := \{(tx, \pi_{tx}, \Sigma)\}_K$ and $tx \notin \text{Ledger}$
2. $\text{TS.Vfy}_2\{tx, \pi_{tx}, \Sigma\} = 1$ for all $(tx, \pi_{tx}, \Sigma) \in x$
3. no two distinct $(tx, \pi_{tx}, \Sigma) \in x$ share the same $tx$

the validation of input $x$ in $\text{Consensus}[e]$ is as follows if $x$ satisfies the following conditions:

1. $|x| = 2f + 1$ and $x := \{(j, e, H_{j,e}, \text{Proof}_{j,e})\}_2f+1$
2. $(j, e, H_{j,e}, \text{Proof}_{j,e})$ is valid for all $(j, e, H_{j,e}, \text{Proof}_{j,e}) \in x$
3. for all $\text{store}[s] := (tx, \sigma) \in H_{j,e}$: $\text{TS.Vfy}_1\{tx, \sigma\} = 1$
4. no two distinct $\text{store}[s] := (tx, \sigma) \in H_{j,e}$ share the same $tx$

38. upon $|M_i| \geq K$ and $p_i$ has not submitted in epoch $e$ do $\triangleright$ with $K = O(n)$
39. invoke $\text{Consensus}[e]$ with $M_{i,e}$ as input $\triangleright$ invoke $\text{Consensus}[e]$
40. wait $\text{Consensus}[e]$ outputs $M_i$ do $\triangleright$ max order seq number in $M_e$
41. let $h_e := \text{MAX}\{\pi_{tx}\}$ in all $(tx, \pi_{tx}, \Sigma) \in M_e$
43. upon $\text{store}[k] \notin \top$ for all $h_{e-1} + 1 \leq k \leq h_e$ do
44. let $H_{j,e} := \text{store}[h_{e-1} + 1 \leq k \leq h_e]$
45. invoke $\text{PNFIFO-BC}[-1]_i(H_{j,e})$

46. wait PNFIFO-BC$[-1]_j$ outputs $(H_{j,e}, \text{Proof}_{j,e})$ for any $j \in P$ do $\triangleright 2f + 1 \log_2 k$
47. if $(j, e, H_{j,e}, \text{Proof}_{j,e})$ is valid do
48. $S_{i,e} \leftarrow S_{i,e} \cup (j, e, H_{j,e}, \text{Proof}_{j,e})$
49. if $|S_{i,e}| = 2f + 1$ do $\triangleright$ indexes set
50. invoke $\text{Consensus}[e]$ with $S_{i,e}$ as input $\triangleright$ invoke $\text{Consensus}_2[e]$
51. wait $\text{Consensus}_2[e]$ outputs $S_e$
52. for all $(j, e, H_{j,e}, \text{Proof}_{j,e}) \in S_e$ do
53. $T_e \leftarrow T_e \cup j$
54. if the newest output PNFIFO-BC$[-1]_j$ is $(H_{j,s_r}, \text{Proof}_{j,s_r})$ and $s_r < e - 1$ do
55. $\text{Help} \leftarrow \text{Help} \cup (j, s_r + 1)$
56. CallHelp($e, \text{Help}, e - 1$)  $\triangleright$ see algorithm 5
57. for all $j \in T_e$ do $\triangleright$ $p_j$ received transactions history
58. wait until $\log_2[1 : e] := \{H_{j,1}, H_{j,2}, \ldots, H_{j,e}\}$ have been received do
59. for all $k \in [e]$ : $\forall \text{store}[s] := (tx, \sigma) \in H_{j,k}$ do
60. $\log_2[\text{store}[s] := tx$ $\triangleright$ $p_j$ received transactions history
61. if $tx \notin \text{Ledger}$ do
62. $\text{Pending}_e \leftarrow \text{Pending}_e \cup tx$ $\triangleright$ add into $\text{Pending}_e$
Algorithm 8: Finalized Output with epoch number $e$, code for process $p_i$

63: parse $M_e := \{\langle v_{tx_i}, \pi_{tx_i}, \Sigma \rangle \}$; \hspace{1em} $\forall (v_{tx_i}, \pi_{tx_i}, \Sigma) \in M_e$: $S' \leftarrow S' \cup \pi_{tx_i}$ \hspace{1em} $\triangleright M_e$ is the output of consensus

64: let $S' = \{\pi_{tx_1}, \pi_{tx_2}, \cdots, \pi_{tx_e}\}$, where $\pi_{tx_i}$ is the $i$-th smallest value in the set $S'$ and $\pi_{tx_i} = h_e$

65: let $M'_e := \{\langle v_{tx_1}, v_{tx_2}, \cdots, v_{tx_e} \rangle \}$

66: for $tx'_i \in M'_e$ and $tx' \in $ \hspace{1em} $\triangleright$ first $tx'_1$, then $tx'_2$, and so on

67: for $tx' \in $ \hspace{1em} $\triangleright$ selecting $t$

68: if $tx'$ appears at least $f + 1$ times in $\log_2(s_{tx'_i} - 1, T_e)$ \hspace{1em} $\triangleright$ sort in ascending order

69: if $Log_2(k) \leq tx'$ and $k < s_{tx'_i}$ \hspace{1em} $\triangleright$ sort in ascending order of $s_{tx'}$

70: \hspace{1.5em} $\triangleright$ let the $f + 1$-th element of $seq(tx')$ as the median value, and denote $s_{tx'}$

71: $seq(tx') \leftarrow seq(tx') \cup k$

72: $S[tx'_i] \leftarrow Sort(S[tx'])$

73: do the following for each $tx_i \in M'_e$

74: $\{tx'_1, tx'_2, \cdots, tx'_{i-1}, Value(tx'_i), tx'_i, \cdots, tx'_e\}$ \hspace{1em} $\triangleright$ insert $Value(tx'_i)$

75: $\{tx'_1, tx'_2, \cdots, tx'_{i-1}, Value(tx'_i), tx'_i, \cdots, tx'_e\}$

76: $\{tx'_1, tx'_2, \cdots, tx'_{i-1}, Value(tx'_i), tx'_i, \cdots, tx'_e\}$

77: $M'_e \leftarrow M'_e \setminus \{tx_i \} \rightarrow tx_i$

78: $\{tx_i \} \rightarrow tx_i$, if $M'_e[tx_i] = tx_i$

79: $\{tx_i \} \rightarrow tx_i$, if $M'_e[tx_i] = tx_i$

80: $\{tx_i \} \rightarrow tx_i$, if $M'_e[tx_i] = tx_i$

81: $\{tx_i \} \rightarrow tx_i$, if $M'_e[tx_i] = tx_i$

82: $\{tx_i \} \rightarrow tx_i$, if $M'_e[tx_i] = tx_i$

References


