Slipstream: Dynamically Available Consensus on a DAG with Fast Confirmation for UTXO Transactions

Author: Please provide author information

Abstract

This paper introduces Slipstream, a Byzantine Fault Tolerance (BFT) consensus protocol tailored for distributed networks with synchronized clocks. The protocol is designed to manage f Byzantine nodes by dividing time into slots, each comprising $f+2$ instants. At each instant, all awake nodes are tasked with proposing blocks of transactions to be added to a Directed Acyclic Graph (DAG). At the end of each slot, using the local DAG, each node generates a digest to the ordering of blocks created in previous slots. Unlike existing DAG-based BFT protocols that rely on threshold clocks, our protocol allows nodes to continue constructing their DAGs and optimistically commit to past slots even during network partitions. Upon recovery, synchronization in the view of slot digests is facilitated through a common random coin, leading to the merging of DAGs. This process enables the finalization of the common slot digest, resulting in obtaining the final sequence of all blocks committed to that digest.

Our protocol is shown to tolerate up to 33% of Byzantine nodes in an eventual lock-step synchronous model and supports fast two-instant UTXO-transaction confirmation during periods of synchrony. Furthermore, we demonstrate that by applying the same protocol and optimistically sequencing committed blocks, it is possible to tolerate up to 50% of Byzantine nodes in a slot-sleepy model, wherein nodes may be either awake or asleep for each slot.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Distributed ledgers, DAG-based consensus, Dynamic availability, Sleepy model, Fast confirmation

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The problem of ordering blocks or, more generally, events in a Byzantine distributed system is a fundamental challenge that has been extensively studied for at least four decades [17]. In recent years, a new promising approach has been proposed that distributes the responsibility for consensus among several nodes in the network and allows for balanced network utilization and high throughput. The idea is to collectively build a Directed Acyclic Graph (DAG), where each vertex in the DAG represents a block of transactions and references to some previously received vertices. Through the logical interpretation of the locally maintained DAG, each node finds the deterministic order on the set of final blocks [2, 4, 9, 13, 15, 16, 22, 27, 28].

Though the above DAG-based BFT protocols guarantee safety and liveness under partially synchronous and asynchronous networks, they are not dynamically available [23, 24]. Dynamic availability is the ability of a protocol to support an unknown number of nodes each of which can go to sleep and awake dynamically. In classical BFT protocols, if many nodes are asleep, achieving consensus becomes challenging due to an insufficient quorum of votes. A class of DAG-based protocols supports dynamic availability [3, 12, 19, 30], however, these solutions typically achieve probabilistic finality under synchronous models with dynamic participation and do not ensure deterministic safety under partially synchronous models.

To have both properties, the idea of flexible BFT was formalized in [21]. It was proposed to design protocols supporting multiple ledgers under different network conditions and adversarial models such that clients can make their assumptions and rely on the respected ledgers. Following this idea, a concept of Ebb-and-Flow consensus protocols was proposed in [23]. In such a protocol, two types of ledgers are utilized. One ledger favors safety and is a prefix of a dynamically available ledger. The former ledger is designed to capture partially synchronous networks, whereas the latter one is always live and secure except with a vanishing probability under a synchronous network with dynamic participation. Following a similar approach, FaBFT was proposed in [26]. This protocol is a flexible asynchronous BFT protocol that utilizes a DAG. In FaBFT, nodes maintain two ledgers: a safer ledger capturing the asynchronous setting and a faster ledger for the partially synchronous setting.

We found a publicly available open-source blockchain project[1], which developed protocol X inspired by the above ideas. By formalizing and enhancing protocol X, we aim to design an Ebb-and-Flow-like DAG-based consensus protocol that achieves fast confirmation for UTXO transactions under synchrony.

1.1 Protocol X

In protocol X, every node in the network can create its block with transactions to be added to a DAG. To achieve consensus, the protocol relies on special validator nodes that create their validation blocks on a time basis (say every 1/2 sec). Each node in the network maintains its local copy of the DAG and runs the consensus protocol through logical interpretation of the DAG without any extra communication overhead. Below we briefly describe some key ideas behind this protocol.

Blocks: Each block in the protocol contains a bounded number of hash references to previous blocks, a timestamp, and a special digest for a certain slot in the past. While normal blocks contain transactions that can update the ledger, validation blocks don't carry any payload but are used instead to achieve consensus.

Slot digest: The protocol X uses slots to divide time into fixed-duration periods. Every node includes in its blocks a cryptographic digest that identifies all blocks in the local DAG created before a certain slot. Generating such digests in a timely manner allows nodes to keep in memory only small data. Each node constantly checks which validators are online and finds which online validators have a similar perception of the DAG by comparing slot digests. After receiving validation blocks, the node identifies which blocks should be added to its local DAG. Such blocks must be referenced enough by the online validators. After the protocol's hard limit following the expiration of a slot, each node generates a digest for the respected slot. Each digest is additionally linked with a previous digest, thus, forming a *backbone chain.*

Reference selection: All nodes collectively build the DAG by attaching their blocks to a bounded number of randomly selected blocks. To guarantee consistency for generated slot digests, nodes reference only those blocks that are not yet referenced and are likely to be eventually added to the local DAG.

Transaction confirmation: The protocol adapts a UTXO ledger model. To confirm a UTXO transaction, each node needs to find in its local DAG a specific validation block pattern. First, when adding a block to the local DAG, the node can identify which transactions in the causal history of the block are approved by the block creator. Second, if there is a quorum ($> 2/3$ of the validator nodes) of validation blocks, where each block creator in the quorum observes another quorum of validation blocks, each approving a transaction, then the transaction gets confirmed.

Chain switching and finality: Once a quorum of validators observes from their blocks that another quorum of validators has adopted the same slot digest, the digest and all blocks

[1] In order to uphold the anonymity requirement of the submission, we do not disclose the name of the blockchain project.

committed to it become final. During asynchronous periods, nodes may perceive their DAGs differently. Upon detecting a divergence, a node switches its DAG to the one corresponding to the *heaviest* backbone chain unless it conflicts with the last final slot digest.

One can distinguish three design goals in this protocol:

1. Ability to provide final ordering of blocks; in particular, providing BFT security guarantees for asynchronous periods and liveness guarantees during synchrony;

2. Support dynamic availability, which includes optimistic ordering of blocks. This process should not halt even during the network partitions; in addition, it should provide security guarantees under a synchronous model with dynamic participation;

3. Support transaction confirmation, which is independent of the block ordering task and achieved promptly during synchronous periods.

1.2 Main challenges

When formalizing protocol X to a permissioned setting and designing protocol Slipstream, which provably achieves the above three goals, we met the following challenges.

Safety issue for confirmation: While confirmation in protocol X might be safe under a synchronous setting, it is easy to find asynchronous cases when a confirmed transaction can be reverted. This happens because when switching the chain, a node *forgets* about the part of its own DAG, which happens after the last final slot. We resolve this issue by merging DAGs when nodes switch their backbone chains.

Liveness issue for switching chain rule: After an asynchronous period ends, protocol X relies on a heuristic heaviest backbone chain rule, which can be manipulated by the adversary. We propose to use a common random coin to select a leader node that proposes its chain for ordering blocks. While this might sound like a cheap solution, it was non-trivial to not rely on the extra randomness when all correct nodes are already sitting on one backbone chain. In addition, to make the same protocol work with a model with dynamic participation, we designed special extra conditions in the chain switching rule and the waking up rule that will not let correct nodes switch their common chain to the one proposed by the adversary.

Locked UTXOs: This is one of the challenges faced by practical blockchain systems that allow for fast-path transaction confirmation (independent from the consensus path), e.g., Sui [2, 7]. When an account creates a double-spend (say, by a software bug), the inputs of the double-spend might forever get locked due to the lack of quorum votes. While protocol X provides the fast path confirmation, our protocol is enhanced with the consensus path that enables nodes to resolve possible double-spends and transactions that didn't get enough approvals. All transactions confirmed through the fast path are ensured to be attempted to be added to the ledger through the consensus path.

1.3 Slipstream framework

The main goal of this paper is to abstract many technical details of protocol X away and design a simple DAG-based protocol that achieves the above three design goals.

Communication and network models: To prove safety and liveness guarantees for the proposed protocol, we introduce two network and communication models. Both models have a lock-step synchronous nature; specifically, each slot is divided into $f+2$ instants with f being the number of Byzantine nodes and the execution of the protocol is proceeded in instants. One model, which we call an eventual lock-step synchronous (ELSS) model, is designated to capture safety under an asynchronous setting, i.e., before Global Stabilization Time or shortly instant GST, the network behaves asynchronously; after GST, it becomes lock-step synchronous. In the second model, which is called a slot-sleepy (SS) model, communication is always lock-step synchronous, but the set of online nodes evolves over slots. See formal details in Appendix A.2.

Description overview: We present Slipstream, a permissioned DAG-based consensus protocol, which can work in both ELSS and SS models without any change. In this protocol, each (awake) node proposes a block to the DAG at each instant, referencing all unreferenced blocks in the DAG. Using Best-Effort Broadcast, the block is sent to all other nodes. To check whether a received block, that is created at slot s and includes the same slot digest, should be added to the local DAG, a node checks a condition inspired by the Dolev-Strong Byzantine Agreement [11]: the validity of a block is also a function of the time when it is received; specifically, all new blocks in the causal history of the received block must have enough observes at slot s. At the end of slot s, the node computes a digest representing the ordering of blocks in the local DAG created before slot $s-1$. Both the finality of slot digests and fast-path transaction confirmation rely on quorums of certificates that are embedded in the local DAG. A certificate is simply a block, the causal history of which contains a quorum of blocks each approving a digest or a transaction. Changing the view on the ordering of the blocks to the one proposed by a random leader could happen at the first instant of the next slot when several conditions are met, including some related to certificates. If the backbone chain is switched, the local DAG gets merged with the leader DAG. Nodes waking up at the beginning of a slot adapt the DAG consistent with the chain proposed by most nodes at the end of the previous slot. See Sec. 3 and 4 for the protocol preliminaries and description.

Results: We show that under the ELSS model and for $n = 3f + 1$ nodes, the final ordering of blocks in Slipstream is always safe and live after GST, specifically, each block of a correct node gets final in $O(f)$ instants in both worst and average cases; each cautious honest UTXO transaction[2] gets always confirmed; after GST, confirmation is achieved in two instants. Under the SS model with each slot having an honest majority, e.g., $n = 2f + 1$ nodes, the optimistic ordering of blocks in Slipstream is always safe and live; specifically, each block of a correct node gets ordered in $O(f)$ instants in both average and worst cases. See formal statements in Appendix C and their proofs in Appendix D.

Our contribution: Up to our best knowledge, Slipstream is the first DAG-based protocol providing liveness and deterministic safety for optimistic and final ordering of blocks in, respectively, a sleepy-like model and a partially synchrony-like model. Once all correct nodes are synced with their backbone chains, the protocol behaves as leaderless. In addition, the protocol has a unique UTXO confirmation rule which can be achieved through the fast path and the consensus path, where the latter allows for resolving locked UTXOs.

2 System model

The network comprises n nodes, with f *Byzantine* nodes. The remaining $n - f$ *correct* (or honest) nodes strictly adhere to the protocol. We will consider $n = 3f + 1$ or $n = 2f + 1$ depending on the network model. There is a public-key infrastructure in the network. Each node is associated with a unique public key, which serves as a node identifier; a node signs blocks with its private key, and the signature is unforgeable; each node knows other nodes' public keys to verify signatures. Time is divided into *slots* of $f + 2$ *instants*. We encode each instant by a *timestamp* $\langle s, i\rangle$, where $s \in \mathbb{N}$ is a slot index and i, $1 \leq i \leq f + 2$, is an instant index within the slot.

Models: We consider two network and communication models each having a lock-step

[2] A transaction with already confirmed inputs and the account holding these inputs doesn't try to double-spend it. More formally, see Appendix A.1

nature. Each awake node proceeds in instants, where each instants has three phases RECEIVE, STATE UPDATE, SEND.

In the eventual lock-step synchronous (ELSS) model, all nodes are always awake and the number of nodes is $n = 3f + 1$. Communication is asynchronous before instant GST. After GST, communication gets synchronous; in particular, all blocks sent by correct nodes before GST are received at $GST + 1$.

In the slot-sleepy (SS) model, for each slot, each node is either awake or asleep for the whole slot and the number of correct slot-awake nodes is strictly larger than the number of Byzantine slot-awake nodes. Communication is synchronous, i.e., $GST = 0$, and a block sent by a correct node at an instant is received by other awake correct nodes at the next instant.

In addition, we assume that for each slot after GST, a random global coin is received by all correct nodes. Before GST, the coin might not be received by any subset of correct nodes. More details about models can be found in Appendix A.2.

Problems for an ELSS model: The first problem for this model is a classical consensus problem. We consider a notion of *finality* for blocks, which allows nodes to linearly sequence final blocks. Each node maintains its final order $\text{ORDER}_{\text{final}}$ of blocks.

▶ **Problem 1.** *Design a DAG-based consensus protocol that satisfies two requirements.* ***Safety:*** *For the sequences of final blocks* $\text{ORDER}_{\text{final}}$ *by any two correct nodes, one must be the prefix of another.* ***Liveness:*** *If a correct node creates a block, then the block becomes eventually final for any correct node, i.e., it gets included to* $\text{ORDER}_{\text{final}}$.

The second problem is related to payment systems. We assume that there exist accounts in the network. These accounts can issue UTXO transactions and broadcast them to nodes. When creating blocks, nodes include a subset of transactions in their blocks. How and when nodes include transactions in their blocks is out of the scope of this paper.

A UTXO is a pair consisting of a positive value and an account that can consume this UTXO. A single-owned UTXO transaction is a tuple containing (i) UTXO inputs, which can be consumed by one account, (ii) the account signature, and (iii) UTXO outputs. The sum of values in the inputs and the outputs are the same. Two different UTXO transactions are called a *double-spend* if they attempt to consume the same UTXO. Note that two identical transactions are not treated as a double-spend. Each account begins with an initial UTXO, which specifies the initial balance of the account.

The order over final blocks naturally allows to find the order over UTXO transactions in these blocks and check which of them can be executed, i.e., being confirmed. However, UTXO transactions don't require a total order, and *confirmation* can be achieved through a consensusless path. We distinguish two paths how confirmation can be achieved: *fast-path* confirmation and *consensus-path* confirmation. In either case, if any correct node regards a UTXO transaction as confirmed, then all correct nodes eventually regard it as confirmed. Once a UTXO transaction is confirmed, all its output UTXOs are said to get confirmed.

We say that a transaction is *cautious* and *honest* if the account creating this transaction is honest, i.e., never creates a double-spend, and the inputs of the transaction are already confirmed in the perception of a correct node to which the transaction is sent. See more details about cautious honest transactions in Appendix A.1.

Since we concentrate only on UTXO transactions in this paper, we consider a UTXO ledger, written as Ledger. This ledger is not a linearly ordered log of transactions, but instead, it is a set of confirmed UTXO transactions that must satisfy the two properties:

1. No double-spend appears in Ledger;
2. For any given UTXO transaction in Ledger, the transactions creating the inputs of the

given transaction must be in Ledger.[3]

▶ **Problem 2.** *Integrate a payment system in a DAG-based consensus protocol that satisfies the three requirements.* ***Safety:*** *For any two UTXO transactions that constitute a double-spend, at most one of them could be confirmed, i.e., included in* Ledger*.* ***Liveness:*** *Every honest cautious UTXO transaction is eventually consensus-path confirmed, i.e., gets included in* Ledger*. There is an instant i after GST such that every honest cautious UTXO transaction created after i is fast-path confirmed two instants after including in a block by a correct node.* ***Consistency:*** *If a UTXO transaction is fast-path confirmed by one correct node, then it gets consensus-path confirmed by every correct node.*

Problem for an SS model: We consider slot digests and a notion of *commitment*, which allow nodes to linear sequence blocks committed to a slot digest. Each node maintains its optimistic order $\textsc{Order}_{\text{own}}$ of blocks.

▶ **Problem 3.** *Design a DAG-based consensus protocol that satisfies two requirements.* ***Safety:*** *For the sequences of committed blocks $\textsc{Order}_{\text{own}}$ by any two correct nodes, one must be the prefix of another.* ***Liveness:*** *If a slot-s awake correct node creates a block, then the block becomes committed by any slot-t awake correct node with $t \geq s+2$.*

3 Protocol Preliminaries

The basic unit of data in the protocol is called a *block*. Each block B contains multiple fields:

- B.refs – a list of hash references to previous blocks;
- B.digest – a slot digest (see Def. 9);
- B.txs – a set of transactions;
- B.time – a timestamp (for simplicity, we refer to the slot and instant indices of the timestamp as B.slot and B.instant);
- B.node – a node's unique identifier;
- B.sign – a node's signature that verifies the content of the block.

A Directed Acyclic Graph (DAG) $\mathcal{D} = (V, E)$ is a pair of a vertex set V, where each vertex is a block, and an edge set E, where each directed edge represents a reference to the hash of a previous block, thereby forming the connections between blocks. Every DAG is supposed to start with the same block called *Genesis* with timestamp $\langle 0, f+2\rangle$. In the following, we often identify $\mathcal{D}$ with its vertex set V for ease of presentation. Informally, a DAG $\mathcal{D}$ is called *valid* if it can be the one maintained by a correct node during the execution of the protocol; this property is defined more formally in Def. 25 in Appendix B.

▶ **Definition 1** (Reachable block)**.** *Block B is called reachable from block C in a DAG if one can reach B starting at C and traversing the DAG along the directed edges.*

In particular, B is reachable from B. A special case of a DAG that we consider in the paper is defined based on the causal history of a block; see Fig. 1.

▶ **Definition 2** (Past cone)**.** *The past cone of block B, written as* Cone(B)*, is the DAG whose vertex set is the set of all blocks reachable from B.*

Each node maintains locally its own DAG $\mathcal{D}$ and the set of all received blocks, denoted as Buffer. In the protocol described later, a correct node issues a block every instant and always references its previous block. Thus, all blocks of a correct node are linearly ordered in $\mathcal{D}$.

[3] We assume that the initial UTXOs of accounts are generated by the genesis transaction, which is always a part of the ledger.

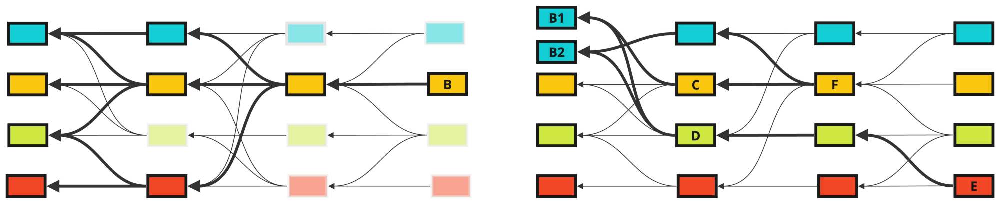

Figure 1 On the left, the past cone of a block B consist of all blocks that are reachable from B. Blocks depicted with less opacity are not in the past cone, while blocks that appear 'fatter' do form the past cone. On the right, $B1$ and $B2$ are equivocations of the blue node. The orange node issues a block C referring only to $B1$ and not to $B2$. The green sees the equivocation immediately and refers with its block D to both blocks $B1$ and $B2$ and adds the blue node to its set of equivocators. The orange node adds the blue node to the set of equivocators in the next instant with block F, and the red one finally witnesses the equivocation with its block E.

▶ **Definition 3** (Equivocation). *A node μ is an equivocator for node η if two blocks created by μ are not linearly ordered in* Buffer *maintained by η.*

We assume that if a correct node detects an equivocation, then this node includes a proof in the next block. After receiving this block, every correct node checks the proof, identifies the equivocator, and updates its known sets of equivocators, denoted as EqSet; see Fig. 1.

▶ **Definition 4** (Quorum). *A set of blocks S is called a quorum if $|\{B.\text{node} : B \in S\}| \geq 2f+1$.*

3.1 Fast-path confirmation for UTXO transactions

The next definition is motivated by cautious transactions, formally discussed in Def. 16.

▶ **Definition 5** (Ready transaction). *For a given block B, a UTXO transaction tx is called B-ready if the two conditions hold:* **1)** *the transaction tx is included in $B.\text{txs}$, and* **2)** *all transactions, whose outputs are the inputs of transaction tx, are confirmed in $\text{Cone}(B)$.*

We define the notion of approval for UTXO transactions.

▶ **Definition 6** (Transaction approval by block). *A block C approves a UTXO transaction tx in block B if the three conditions hold:* **1)** *transaction tx is B-ready,* **2)** *block B is reachable from C, i.e., $B \in \text{Cone}(C)$,* **3)** *no block, that contains a double-spend for tx, is reachable from C.*

Recall that two instances of the same transaction tx, that is included in different blocks B and C, i.e., $tx \in B.\text{txs}$ and $tx \in C.\text{txs}$, are not considered as a double-spend, and the number of transaction approvals is computed independently for tx in B and tx in C.

▶ **Definition 7** (Transaction certificate). *A block C is called a transaction certificate (TC) for a UTXO transaction tx in block B if $C.\text{slot} \in \{B.\text{slot}, B.\text{slot}+1\}$ and $\text{Cone}(C)$ contains a quorum S of blocks, each of which approves tx in B.*

We refer to Fig. 2 to demonstrate the concept of transaction certificates. By Def. 16, 19, and 6, if the causal history of a block C contains a quorum of blocks all observing an honest cautious transaction in a block B, then C is a transaction certificate.

▶ **Definition 8** (Fast-path confirmation). *A transaction tx is said to be fast-path confirmed in a DAG if the DAG contains a block B and a quorum of blocks, each of which is a certificate for tx in B.*

In Fig. 2, transaction tx is fast-path confirmed due to a quorum of TCs for tx in B.

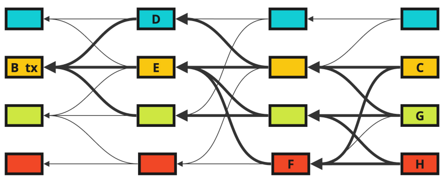

Figure 2 The block B contains an honest cautious transaction tx, i.e., $tx \in B$.txs, and thus tx is B-ready. The block C issued by the orange node is a transaction certificate for tx in B, as its past cone contains a quorum represented by blocks D, E, and F, approving block tx in B. In the same way, blocks G and H form transaction certificates; hence, transaction tx is fast-path confirmed.

3.2 Slot digest

We proceed with a definition of a slot digest that allows us to order blocks created before a certain slot optimistically. For a given slot digest σ, we write σ.slot to refer to the slot for which the digest is constructed.

▶ **Definition 9** (Slot digest). *A digest σ_s for a slot s is defined based on a valid DAG at time instant $\langle s+1, f+2\rangle$ in an iterative way:*

- *for $s \geq 1$,*

$$\sigma_s = \textsc{Hash}(\sigma_{s-1}, \textsc{Concat}(Blocks_{\leq s})),$$

  where $Blocks_{\leq s}$ denotes all blocks in the DAG that are created at or before slot s and not committed by σ_{s-1}. Here, σ_{s-1} is the digest that is included in all blocks in the DAG, which are created at slot $s+1$ and instant $\leq f+1$. A function $\textsc{Concat}(\cdot)$ orders blocks in a certain deterministic way and concatenates them. We say the digest σ_s commits all the blocks committed by σ_{s-1} and the blocks in set $Blocks_{\leq s}$;
- *$\sigma_{-1} = 0$ and $\sigma_0 = \textsc{Hash}(0, Genesis)$.*

Furthermore, we write $\textsc{BeforeDigest}(\sigma_s)$ to denote σ_{s-1} and write $Blocks_{=s}$ to denote all blocks created at slot s.

We assume that the adversary is computationally bounded and digests are unique. We refer to Fig. 3 to demonstrate an example of how digests are included in blocks; in particular, for the same slot, nodes could generate different digests. See Appendix B.3 for details on when slot digests are included in blocks in a valid DAG.

▶ **Definition 10** (DAG associated with digest). *For a given slot digest σ, we write $\mathcal{D}(\sigma)$ to denote the DAG whose vertex set is the set of all blocks committed to σ. We write* EqSet(σ) *to denote the set of equivocators that can be found based on $\mathcal{D}(\sigma)$.*

By definition, a digest for slot s corresponds to a unique sequence of committed blocks created at or before slot s. One can invoke $\textsc{Order}(\sigma_s)$ to get the order of blocks in $\mathcal{D}(\sigma_s)$. Specifically, the order is defined iteratively:

$$\textsc{Order}(\sigma_s) = \textsc{Order}(\sigma_{s-1}) \,||\, \textsc{Concat}(Blocks_{\leq s}), \tag{1}$$

where $Blocks_{\leq s}$ are all the blocks in $\mathcal{D}(\sigma_s) \setminus \mathcal{D}(\sigma_{s-1})$ with $\sigma_{s-1} = \textsc{BeforeDigest}(\sigma_s)$, and the base order is $\textsc{Order}(\sigma_0) = (Genesis)$. Each node maintains its optimistic order of blocks written as $\textsc{Order}_{\text{own}}$. This order is defined as $\textsc{Order}(\sigma)$, where σ is the digest adopted by the node.

Slot digests form a slot digest tree, where the path between the root and any vertex in the tree determines a backbone chain. If two correct nodes agree on digest σ, they share the same subDAG $\mathcal{D}(\sigma)$ in their respective DAGs.

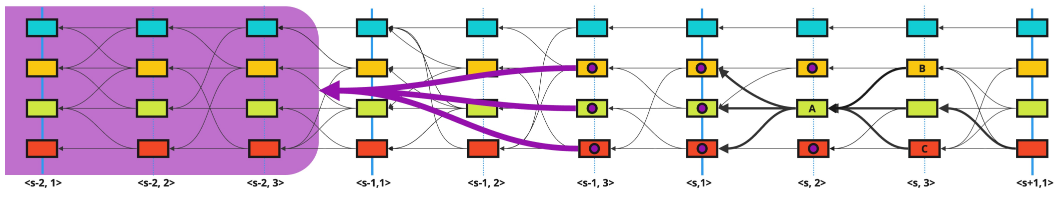

Figure 3 The blocks of the orange, green, and red nodes created at instant $\langle s-1, 3\rangle$ contain a digest of the DAG up to slot $s-2$. Here, the fat purple edges are not normal references but represent the B.digest entries, and this digest is depicted using a purple circle. The blue node has a different perception of what happened in slot $s-2$ and will create a different digest not shown in the image. Block A does contain blocks of a quorum with the same digest in its past cone, and hence serves as a digest certificate (DC) of the "purple" digest. This digest becomes final as blocks B and C also form DCs.

▶ **Definition 11** (Backbone chain). *For a given digest σ_s, define the backbone chain associated with σ_s as*

$$\mathrm{Chain}(\sigma_s) = (\sigma_{-1}, \sigma_0, \ldots, \sigma_{s-1}, \sigma_s),$$

where $\sigma_{j-1} = \textsc{BeforeDigest}(\sigma_j)$ for $j \in [s]$. When the backbone chain Chain *is clear from the context, we also write* Chain$[j]$ *to denote σ_j.*

One can define naturally which digests are consistent and conflicting.

▶ **Definition 12** (Consistent and conflicting digest). *A digest σ is called consistent with a digest θ if the backbone chain* Chain(σ) *contains θ. Two digests σ and θ are called conflicting if neither σ is consistent with θ, nor θ is consistent with σ. To check whether two digests are conflicting, nodes use a function* $\textsc{IsConflict}(\theta, \sigma)$ *that outputs* 1 *in case they are conflicting and* 0 *otherwise.*

3.3 Finality for slot digests

Similar to transaction certificates, we introduce the notion of digest certificates that allow nodes to finalize backbone chains and secure finality for a chain switching rule (see Sec. 4.7).

▶ **Definition 13** (Digest certificate). *A digest certificate* (DC) *for a digest σ is a block A with A.slot $= \sigma$.slot $+ 2$ such that* Cone(A) *contains a quorum of blocks, all issued at the same slot A.slot and include the same digest σ. Note that either A.digest $= \sigma$, or* $\textsc{BeforeDigest}(A.\mathrm{digest}) = \sigma$. *If A is a* DC, *nodes can invoke a function* $\textsc{Digest}(A)$ *to get the digest σ, certified by A.*

In Fig. 3, blocks A, B and C are certificates for the same digest.

▶ **Definition 14** (Final digest). *A digest σ is called final in a valid DAG if the DAG contains a quorum of* DCs *for σ. In addition, if σ is final, then all digests in the backbone chain ending at σ, i.e.,* Chain(σ), *are also final.*

In Fig. 3, the digest gets final due to a quorum of certificates.

Each node maintains its view on the final order of blocks denoted as $\textsc{Order}_{\mathrm{final}}$. This order is defined as $\textsc{Order}(\sigma)$ with σ being the latest final digest.

Even though we show that a digest that is final for one correct node will eventually become final for all other correct nodes, the finalization of the same digest might happen at different slots for different nodes. To have a more robust perception of the finality time of digests, we introduce the following definition which may sound like a double finality.

▶ **Definition 15** (Finality time). *Let $\mathcal{D}$ be a valid DAG. Let s_{final} be the slot index of the last final digest $\sigma_{s_{\text{final}}}$ in $\mathcal{D}$. Let s_{pre} be the slot index of the last final digest in $\mathcal{D}(\sigma_{s_{\text{final}}})$. Then for any digest σ_s with $1 \leq s \leq s_{\text{pre}}$, one can define the finality time* $\textsc{FinalTime}(\sigma_s)$ *as the first slot index τ such that σ_s is final in $\mathcal{D}(\sigma_\tau)$*[4]*. For $s > s_{\text{pre}}$, we set* $\textsc{FinalTime}(\sigma_s)$ *to be $\perp$.*

From the safety property of digest finality (see Appendix D.2), it follows that if $\textsc{FinalTime}(\sigma_s)$ is defined not as $\perp$ for two correct nodes, then their values are the same. Even though the notion of a finality time is defined for digests, one can define $\textsc{FinalTime}(s)$ for a slot s as $\textsc{FinalTime}(\sigma)$ with $\sigma.\text{slot} = s$ when the finality time for digest σ for slot s is defined.

3.4 Consensus-path confirmation for UTXO transactions

Given a valid DAG $\mathcal{D}$ with a backbone chain Chain, the notion of finality time allows to partition the DAG. Specifically, let $(\tau_1, \ldots, \tau_{i+1})$ be the vector of all distinct finality times for digests in Chain (here, τ_1 is assumed to be 0); then one can partition the DAG $\mathcal{D}$ using the corresponding digests: $\mathcal{D}(\sigma_{\tau_{j+1}}) \setminus D(\sigma_{\tau_j})$ for $2 \leq j \leq i$. Informally, the consensus-path confirmation is defined using this partition. First, all transactions with transaction certificates in the partition are attempted to be added to Ledger. Second, by traversing the blocks in the partition in the final order $\textsc{Order}_{\text{final}}$, all transactions that don't have certificates are attempted to be added to Ledger. The attempt is successful if no conflicting transaction is already in Ledger and the transactions creating transaction inputs are already in Ledger, i.e., when the UTXO ledger properties are not violated (see Sec. 2). More formally, we refer for consensus-path confirmation to procedure $\textsc{ConfirmTxsConsensusPath}()$ in Alg. 5.

4 Protocol description

We use $\langle s, i\rangle$ to denote the current timestamp consisting of the slot and the instant index. During the execution of the protocol, each node maintains its own DAG $\mathcal{D}$, which can only be extended by adding new vertices in the state update phase. During slot s, a node adapts one backbone chain $\text{Chain}(\sigma_{s-2}) = (\sigma_{-1}, \sigma_0, \ldots, \sigma_{s-2})$. At the last instant of slot s, the node updates the chain with the digest σ_{s-1} for slot $s-1$ such that $\textsc{BeforeDigest}(\sigma_{s-1}) = \sigma_{s-2}$. Switching the backbone chain can happen only at the state update phase of the first instant of a slot. Switching the chains is important for synchronization after instant GST when the asynchronous period ends. The broadcasting of blocks during the first $f+1$ instants of slot s is used to "synchronize" the nodes' perception of the previous slot $s-1$. In particular, after GST, all correct nodes do agree on the blocks of slot $s-1$ at instant $\langle s, f+2\rangle$ and do create the same slot digest for slot $s-1$. The flow of procedures is described in Alg. 1. The following sections delve into the different components of the protocol.

1. Creating a block: Creating a new block B_{own} at time instant $\langle s, i\rangle$ happens at the end of the state update phase, see line 30 of Alg. 1. For this purpose, nodes use function $\textsc{CreateBlock}()$, see Alg. 3. The node includes the timestamp $\langle s, i\rangle$, a subset of received transactions created by accounts, and the currently adopted slot digest σ_{s-2}.[5] The node also includes the hash references to all unreferenced blocks in the maintained DAG.

[4] Note that the finality time of some digests may coincide.
[5] This happens for instant $i < f+2$. For instant $\langle s, f+2\rangle$, the node includes σ_{s-1}, the digest corresponding to slot $s-1$.

Algorithm 1 Slipstream consensus protocol

```
1  Local variables:
2      $\mathcal{D} \leftarrow \{Genesis\}$                          // DAG maintained by the node
3      Buffer $\leftarrow \{Genesis\}$                               // Buffer maintained by the node
4      Chain $\leftarrow (0)$                                        // backbone chain adapted by the node
5      Ledger $\leftarrow \{\}$                                      // Ledger maintained by the node
6      $\text{ORDER}_{\text{own}} \leftarrow (Genesis)$              // Optimistic order of blocks
7      $\text{ORDER}_{\text{final}} \leftarrow (Genesis)$            // Final order of blocks
8      $s_{\text{final}} \leftarrow 0$                               // Slot index of the last final slot digest
9      $s_{\text{pre}} \leftarrow 0$        // Slot index of the last final slot digest in $\mathcal{D}(s_{\text{final}})$, see Def. 15
10     pk, sk                                                        // Public and secret keys of the node
11     $B_{\text{own}} \leftarrow \{\}$                              // Last block created by the node
12     EqSet $\leftarrow \{\}$                                       // Set of equivocators known to the node
13 for slot index $s = 1, 2, 3, \ldots$ do
14     for instant index $i = 1, \ldots, f+2$ do
15         Receive phase
16         RECEIVEMESSAGES()                                         // see Sec. 4.3
17         ------------------------------------------------------------
18         State update phase
19         UPDATEHISTORY()                                           // see Sec. 4.4
20         UPDATEDAG()                                               // see Alg. 2
21         if $i = 1$ then
22             if the node was slot-$(s-1)$ asleep then
23                 WAKEUPCHAIN()                                     // see line 37 in Alg. 4
24             else
25                 SWITCHCHAIN()                                     // see line 8 in Alg. 4
26         if $i = f+2$ then
27             UPDATECHAIN()                                         // see line 25 in Alg. 3
28         CONFIRMTXSFASTPATH()                                      // see Alg. 5
29         FINALIZESLOTS()                                           // see line 35 in Alg. 3
30         $B_{\text{own}} \leftarrow$ CREATEBLOCK()                 // see line 9 of Alg. 3
31         ------------------------------------------------------------
32         Send phase
33         BROADCAST($B_{\text{own}}$)                               // see Sec. 4.2
```

2. Best-effort broadcast: The key principle in broadcasting a newly created block B_{own} using BROADCAST() is to ensure that the recipient knows the past cone of the new block and, thus, might be able to update its DAG. Each node needs to track the history of the known blocks by every other node, e.g., in the perception of a given node, the set of known blocks by node η is denoted as History$[\eta]$. So, when sending the newly created block B_{own}, the node sends to node η the set Cone$(B_{\text{own}}) \setminus$ History$[\eta]$ and makes the update as follows History$[\eta] \leftarrow$ History$[\eta] \cup$ Cone(B_{own}).

3. Receiving messages: In the receive phase of an instant, the protocol executes a procedure called RECEIVEMESSAGES(), which allows to receive messages, i.e., blocks from nodes or a common random coin from the oracle. When receiving a new block B, the node puts this block into Buffer. Before instant GST, the node might not receive a block B with B.time $< GST$ created by a correct node. After GST, each block B with timestamp $\langle s, i\rangle = B.\text{time} \geq GST$, created by a correct node, is received by any other correct node at the next instant $\langle s, i+1\rangle$ (or $\langle s+1, 1\rangle$ in case $i = f+2$). If instant $\langle s, 1\rangle \geq GST$, then at the receive phase, the node receives a common random coin $r(s)$ that determines who is the leader node for slot s. Under certain circumstances, the block created at instant $\langle s-1, f+2\rangle$ by the leader node will be used for the chain-switching rule (see Sec. 4.7). Specifically, to determine which node is a leader, the node calls the function LEADER().

4. Updating the history: The state update phase includes the procedure, denoted as UPDATEHISTORY(), that updates the set of known blocks by other nodes. When the causal history of a block B is known to the node, i.e., Cone$(B) \subset$ Buffer, the node updates the history History$[\eta] \leftarrow$ History$[\eta] \cup$ Cone(B), where $\eta = B$.node.

5. Updating the backbone chain: The state update phase of the last instant $\langle s, f+2\rangle$ of slot s includes the procedure, UPDATECHAIN(), that extends the backbone chain Chain(σ_{s-2})

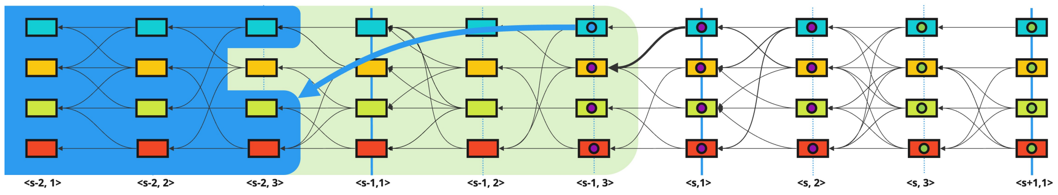

Figure 4 We are in the situation, which is similar to Fig. 3 up to instant $\langle s, 1\rangle$. Now, the blue node realizes that its digest on the "blue" past is not the one of the majority and merges its DAG with the past cone of the leader, which is the orange node. So, it switches to the "purple" chain at instant $\langle s, 1\rangle$. As before, the purple digest becomes final, and at instant $\langle s, 3\rangle$, all nodes have the same perception of what happened until slot $s-1$ and create the same "green" digest.

by digest σ_{s-1}, see Def. 9 and Alg. 3. It also naturally orders blocks in the maintained DAG that are created at or before slot $s-1$ and updates $\textsc{Order}_{\text{own}}$ as $\textsc{Order}(\sigma_{s-1})$.

6. Updating the DAG: The procedure $\textsc{UpdateDAG}()$ in Alg. 2 updates the DAG $\mathcal{D}$ maintained by the node. Suppose the current timestamp is $\langle s, i\rangle$. Suppose that the buffer contains a block B with timestamp $B.\text{time} = \langle s, i-1\rangle$ (or $\langle s-1, f+2\rangle$ for $i = 1$) and the same digest as the node adopts. Suppose that the past cone $\text{Cone}(B)$ is in Buffer; the node adds to the DAG $\mathcal{D}$ the whole past cone $\text{Cone}(B)$ if all the updating (U) conditions are met:

U1 No block at slot s in $\text{Cone}(B)$ is created by an equivocator from $\text{EqSet}(\sigma_{s-2})$, see Def. 10 and line 10 of Alg. 2.

U2 The past cone $\text{Cone}(B)$ is valid, e.g., all blocks in $\text{Cone}(B)$ that are created at slot s contain the same digest σ_{s-2}, see Def. 25 and line 18 of Alg. 2.

U3 For every block C in $\text{Cone}(B) \setminus \mathcal{D}$ which is not committed to the chain $(\sigma_{-1}, \sigma_0, \ldots, \sigma_{s-2})$ and created at slot $s-1$ or before, the node computes its *reachable number* $\textsc{ReachNumber}$ (C, B) - the number of distinct nodes who created blocks in $\text{Cone}(B)$ at slot s from which one can reach C. It must hold that $\textsc{ReachNumber}(C, B) \geq i-1$, see line 14 of Alg. 2.

7. Chain switching rule: If the node was slot-s awake, the node calls the procedure $\textsc{SwitchChain}()$ at instant $\langle s+1, 1\rangle$, see Alg. 4. First, the node checks how many other nodes generated the same digest σ_{s-1} as the node after running $\textsc{UpdateChain}()$ at the previous instant. Let N_{total} be the number of nodes having issued a block at instant $\langle s, f+2\rangle$ that is received by a node at instant $\langle s+1, 1\rangle$. Let N_{same} be the number of those nodes having the same digest as the given node. To compute these values, the node uses the function $\textsc{LastBlockFrom}(\textit{node})$ that outputs the latest block created by *node* and stored in Buffer.

Before checking the condition for switching the chain, the node runs the procedure $\textsc{CheckELSS}()$ in Alg. 4 that updates the indicator I_{ELSS} to 1 if there have been two different digests for slot $s-1$ with more than $f+1$ supporters. Recall that we have two different communication models, the ELLS model, Def. 22, and the SS model, Def. 23. Having two different digests with more than $f+1$ supporters indicates that we are not in the SS model. Similarly, in line 27, the digest certificate of any node can not conflict with the digest of a correct node in the SS model, and if so, I_{ELSS} turns to 1.

If the node has not finalized the digest at the previous slot s and the leader for switching chain is determined using the function $\textsc{Leader}()$, then the node has a chance to switch its chain, see line 20. We distinguish between two cases:

a. $2N_{\text{same}} \leq N_{\text{total}}$: if the digest of the leader is consistent with the own digest OR the node's digest certificate is created not later than the one of the leader, the node updates the DAG by adding the past cone of the leader block B_{leader} and adopts the leader's backbone chain together with its induced optimistic ordering for blocks.

b. $2N_{\text{same}} > N_{\text{total}}$: if the ELSS indicator is true, i.e., indicating that the underlying model is the ELSS model, AND the node's digest certificate is created not later than the one of the leader, the node merges the past cone of the leader block with its maintained DAG and adopts the leader's backbone chain and induced final ordering for blocks.

8. Waking up rule: Suppose a node is slot-$(s+1)$ awake and was slot-s asleep. The node can detect this by inspecting the local DAG at time instant $\langle s+1, 1\rangle$ and checking the existence of its own block B_{own} at instant $\langle s, f+2\rangle$. If no such block exists, then the node proceeds with WakeUpChain(), see line 37 in Alg. 4.

Using function Mode(M), the node checks which digest, say σ_{s-1}, is the most frequent one among all created by non-equivocator nodes at instant $\langle s, f+2\rangle$. Then the node adopts the backbone chain to the chain corresponding to σ_{s-1} and adapts the induced ordering for blocks. In addition, it updates the local DAG with past cones of all blocks that include σ_{s-1}.

9. Finality rule: Using Def. 14, the node finalizes slot digests using FinalizeSlots() at line 35 in Alg. 3. For this purpose, the node tries to find a quorum of digest certificates. Once a digest σ is final, a final ordering $\textsc{Order}_{\text{final}}$ of the blocks committed to σ is obtained using the function Order(), see Eq. (1). The exact nature of this order relies on the function Concat(), which concatenates the blocks committed by σ, but not by BeforeDigest(σ). While the order is not important for the results of this paper, the obtained order is used for traversing blocks in procedure ConfirmTxsConsensusPath() and, thus, the ordering must respect the causal order relationships of blocks in the DAG.

10. Confirmation rule: UTXO transactions can be confirmed either through the fast path or the consensus path. For the fast path, the protocol runs ConfirmTxsFastPath() at state update of each instant, see line 28 in Alg. 1. The DAG $\mathcal{D}$ is inspected to find a quorum of transaction certificates for all possible transactions, see Def. 8 and line 23 in Alg. 5. For the consensus path, the protocol runs procedure ConfirmTxsConsensusPath(). As said in Sec. 3.4, the consensus path relies on the notion of the finality time. Since the latest finality time could be updated only when the last final slot is updated, ConfirmTxsConsensusPath() is executed inside FinalizeSlots() at line 48 in Alg. 3. In procedure ConfirmTxsConsensusPath(), one first gets the latest finality time τ. Then, for each transaction included in blocks $B \in D(\sigma_\tau)$ with $B.\text{slot} \leq \tau - 2$, one checks the existence of a transaction certificate, and if such one exists, the transaction is attempted to be added to Ledger. The attempt is successful if the two UTXO ledger properties are preserved (see Sec. 2). Then, using the total order, all transactions in blocks $B \in D(\sigma_{\tau-2})$ without certificates are similarly attempted to be added to Ledger. We note that the set of blocks processed in the above two loops might be different in a general case.

5 Related work

DAG-based BFT protocols: There is a class of round-based consensus protocols on DAGs designed for partially synchronous and asynchronous networks. In a round-based protocol, a node can increase the round number only when the DAG contains a quorum of blocks with the current round number. These protocols optimize their performance by assigning leaders for certain rounds and using special commit rules for leader blocks. Committed leader blocks are linearly ordered, forming a backbone sequence on the DAG that allows for the partitioning of the DAG into slices and the deterministic sequencing of blocks in the slices.

Many protocols in this class such as Aleph [13], DAG-Rider [15], Tusk [9], and Bullshark [28] use special broadcast primitives such as Byzantine Reliable Broadcast (BRB) and Byzantine Consistent Broadcast (CRB) to disseminate *all* blocks. Therefore, during the execution of any of these protocols, nodes construct a *certified DAG*, where each block

Algorithm 2 Procedure to update the DAG.

```
procedure UpdateDAG():
    $\langle s, i\rangle \leftarrow$ Now()
    $\langle s', i'\rangle \leftarrow$ BeforeTime($\langle s, i\rangle$)
    if $B_{\text{own}}$.time $\neq \langle s', i'\rangle$ then
        return
    $\sigma \leftarrow$ CurrentDigest()                                    // see Line 21 of Alg. 3
    for $B \in$ Buffer s.t. Cone($B$) $\subset$ Buffer $\wedge$ $B$.time $= \langle s', i'\rangle \wedge B$.digest $= \sigma \wedge B$.node $\notin$ EqSet do
        $check \leftarrow 1$
        for $C \in$ Cone($B$) s.t. $C$.slot $= s$ do
            if $C$.slot $\in$ EqSet($\sigma$) then
                $check \leftarrow 0$
                break
        for $C \in$ Cone($B$) $\setminus \mathcal{D}$ s.t. $C$.slot $\leq s - 1 \wedge \sigma$ does not commit $C$ do
            if ReachNumber($C, B$) $< i - 1$ then
                $check \leftarrow 0$
                break
        if $check \wedge$ IsValid(Cone($B$)) then
            $\mathcal{D} \leftarrow \mathcal{D} \cup$ Cone($B$)
// The function outputs the number of nodes that created a block in Cone($B$) at slot $B$.slot from which one can reach $C$
function ReachNumber($C, B$):
    $S \leftarrow \{\}$                                                    // Set of nodes
    for $E \in$ Cone($B$) s.t. $E$.slot $= B$.slot do
        if $C \in$ Cone($E$) then
            $S \leftarrow S \cup \{C.\text{node}\}$
    return $|S|$
// The function outputs the previous timestamp
function BeforeTime($\langle s, i\rangle$):
    if $i > 1$ then
        return $\langle s, i - 1\rangle$
    else
        return $\langle s - 1, f + 2\rangle$
// The function verifies if a DAG is graph-, time-, and digest valid
function IsValid($\mathcal{C}$):
    return $\mathcal{C}$ is valid                                           // See Def. 25
```

comes with its certificate, a quorum of signatures, which does not allow any equivocating blocks to appear in the DAG. The latency for non-leader blocks in the above protocols suffers from introducing *waves*, a number of consecutive rounds with one designated leader block. Shoal [27] reduces the latency of non-leader blocks by interleaving two instances of Bullshark.

All the above protocols use broadcast primitives for every block, leading to an increased latency compared to the state-of-the-art chain-based consensus protocols like HotStuff [20, 29]. To reduce this latency and simplify the consensus logic, BBCA-Chain [22] suggested using a new broadcast primitive, called Byzantine Broadcast with Complete-Adopt (BBCA), only for leader blocks, and Best-Effort Broadcast (BEB) for all other blocks. In addition, BBCA-Chain makes all rounds symmetric by assigning leaders every round. Note that there has been a similar effort; specifically, Sailfish [25] that assigns a leader node for every round and allows committing even before BRB instances deliver voting blocks. This allows improving the latency in the *happy-case* when all nodes are correct.

Hashgraph [4] is the first DAG-based leaderless BFT protocol in which nodes use BEB for disseminating all their blocks. Nodes construct an *optimistic* DAG, in which equivocating blocks could appear. By logical interpretation of the DAG, nodes exclude equivocations and run an inefficient binary agreement protocol that orders blocks by the received median timestamps, leading to very high latency in the worst case. Cordial Miners [16] use a similar mechanism to exclude equivocations. However, the latency is significantly improved by assigning leaders in each 3-round wave and introducing a special commit rule for leader blocks on such an optimistic DAG. Mysticeti [2] is the most recent improvement of Cordial

Miners that introduces pipelined leader blocks. Mysticeti allows every node at every round to be a leader, which significantly reduces the latency, especially in case of crash nodes.

Our protocol, Slipstream, uses BEB for disseminating all blocks in the DAG and uses time slots (not rounds) for the partition of the DAG into slices. On the one hand, the happy-case latency of committing blocks in slices is linear with the number of Byzantine nodes, which is much higher than the constant average latency in many other DAG-based solutions. On the other hand, all existing DAG-based BFT protocols halt in case of network partitions (e.g., when 1/3 of nodes are offline), whereas our protocol allows for optimistic committing the DAG in such cases and tolerating a higher fraction of Byzantine nodes under a synchronous slot-sleepy network model. In addition, during synchrony, our protocol is leaderless meaning nodes never attempt to switch their chain using a leader derived via a common random coin.

Payment systems on a DAG: While the total ordering of transactions is known to require solving the consensus problem, it is shown in [14] that a system that enables participants to make simple payments from one account to another needs to solve a simpler task. In cases where payments are independent of one another (e.g., single-owned token assets or UTXO transactions), ordering payments becomes unnecessary, and a partial ordering suffices. This was later observed in multiple papers [5, 6], and different solutions that utilize DAGs were suggested. In Flash [18], nodes create blocks to approve transactions included in prior blocks and a quorum of approvals is sufficient to commit a UTXO transaction. While this solution achieves very low communication complexity and latency even in asynchronous settings, this and other prior solutions don't address some practical concerns.

One of the major issues of consensusless payment systems is a potential equivocation on a client's side. When a double-spend is created by a client's wallet, e.g., due to a software bug, the assets might get locked forever. In Sui Lutris [7] and Mysticeti [2], this problem is solved by proper reconfiguration between epochs in which validators in the next committee unlock previously locked owned objects. In Sui Lutris [7], a client is responsible for getting a quorum of signatures for a given transaction and forming a transaction certificate, which ensures that no double-spend for the owned object will ever attempt to be committed by validators. An owned-object transaction is executed immediately after arriving of its certificate. Nevertheless, all transaction certificates are committed through the consensus path, which uses Narwhal [9] and Bullshark [28]. The latter allows for making proper checkpoints needed for synchronization and epoch reconfiguration. In Mysticeti-FPC [2], clients submit transactions directly to nodes without constructing certificates. Nodes include explicit approval votes in their blocks about transactions in prior blocks. While a quorum of approvals is sufficient for the safe execution of owned-object transactions, nodes use implicit transaction certificates derived from the DAG structure for committing such transactions.

In our protocol, UTXO transactions can be executed once a quorum of transaction certificates is derived from the DAG. The structure of transaction certificates is very similar to the one of Mysticeti-FPC, with the only difference being that votes about transactions are not explicitly written in blocks, but derived from the DAG (similar to Flash [18]). During synchronous periods, two instants suffice for quorum construction. To resolve the problem with potentially locked UTXOs, we give a short two-slot period for certificates to appear. If a transaction doesn't get certificates in time, then after finalizing some slots, the total order is used to resolve double-spending. We argue that a similar technique could be used in Mysticeti-FPC: instead of voting on pure transactions (and summing up the votes for a transaction in different blocks), nodes could vote on pairs of transactions and blocks, where blocks' round numbers are used as anchors in the DAG; if no transaction certificate for a given transaction in a given block appears in the next few rounds, then the total order from the committed leader block could resolve double-spends or unlock locked owned objects.

References

1 Dick Alstein. *Distributed consensus and hard real-time systems*. Computing science reports. Technische Universiteit Eindhoven, 1994.

2 Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

3 Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3363213.

4 Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 34:9–11, 2016.

5 Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

6 Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. Zef: low-latency, scalable, private payments. In *Proceedings of the 22nd Workshop on Privacy in the Electronic Society*, pages 1–16, 2023.

7 Same Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus. *arXiv preprint arXiv:2310.18042*, 2023.

8 Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

9 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

10 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

11 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

12 Matthias Fitzi, Peter Ga, Aggelos Kiayias, and Alexander Russell. Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition. *Cryptology ePrint Archive*, 2018.

13 Adam Gkagol and Michał Świketek. Aleph: A leaderless, asynchronous, byzantine fault tolerant consensus protocol. *arXiv preprint arXiv:1810.05256*, 2018.

14 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

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

16 Idit Keidar, Oded Naor, and Ehud Shapiro. Cordial miners: A family of simple, efficient and self-contained consensus protocols for every eventuality. *arXiv preprint arXiv:2205.09174*, 2022.

17 Leslie Lamport, Robert Shostak, and Marshall Pease. *The Byzantine generals problem*, page 203–226. Association for Computing Machinery, New York, NY, USA, 2019. URL: https://doi.org/10.1145/3335772.3335936.

18 Andrew Lewis-Pye, Oded Naor, and Ehud Shapiro. Flash: An asynchronous payment system with good-case linear communication complexity. *arXiv preprint arXiv:2305.03567*, 2023.

19 Chenxing Li, Fan Long, and Guang Yang. Ghast: Breaking confirmation delay barrier in nakamoto consensus via adaptive weighted blocks. *arXiv e-prints*, pages arXiv–2006, 2020.

20 Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

21 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

22 Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: One-message, low latency bft consensus on a dag. *arXiv preprint arXiv:2310.06335*, 2023.

23 Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 446–465. IEEE, 2021.

24 Rafael Pass and Elaine Shi. The sleepy model of consensus. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II 23*, pages 380–409. Springer, 2017.

25 Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving latency of dag-based bft. *Cryptology ePrint Archive*, 2024.

26 Yu Song, Yu Long, Xian Xu, and Dawu Gu. Fabft: Flexible asynchronous bft protocol using dag. *Cryptology ePrint Archive*, 2023.

27 Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

28 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

29 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

30 Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105. IEEE, 2020.

A Formal model

A.1 Honest and cautious accounts

▶ **Definition 16** (Cautious account). *A cautious account is an account that creates a UTXO transaction only if all the inputs to transactions are already confirmed by one correct node. Transactions issued by cautious accounts are called cautious transactions and cautious transactions are supposed to be sent to at least one such correct node.*

▶ Remark 17. In a Byzantine environment, it is in general not known which nodes are correct and which are not. However, having at least $f+1$ nodes confirming a transaction is sufficient for knowing that at least one correct node is confirming the transaction. Then the transaction can be sent to these $f+1$ nodes.

▶ Remark 18. In the protocol itself, an account is not necessarily running a node. However, for simplicity, we can assume either an account has its own node, or a node is able to produce a proof of the transaction confirmation that can be verified by the account.

▶ **Definition 19** (Honest account). *An honest account is an account that uses any given UTXO associated with this account only once to create a UTXO transaction. Transactions issued by honest accounts are called honest transactions.*

A.2 Network and communication model

We focus on two models in the paper: an *eventual lock-step model* and a *slot-sleepy model*. Before explaining them, we recall a standard lock-step synchronous model.

▶ **Definition 20** (Lock-step synchronous model). *A consensus protocol works under a lock-step synchronous model if and only if an execution consists of a sequence of instants, and an instant includes a sequence of the three phases:*

- *A* Receive phase*, in which nodes receive messages sent by other nodes. In the current instant, a node receives all messages from correct nodes that are sent in the previous instant.*
- *A* State update phase*, in which a node takes an action based on the received messages from the receive phase.*
- *A* Send phase*, in which nodes send messages to other nodes.*

▶ Remark 21. Compared to the original definition of a lock-step synchronous model [1], we changed the order of the phases by moving the send phase to the end. This is done intentionally in order to simplify the description of a slot-sleepy model (see Def. 23). In particular, for the described sequence of phases, nodes can skip the receive and state update phases at the very first instant in the execution of the protocol.

In an eventual lock-step synchronous model, the network behaves as a lock-step synchronous model after an (unknown) finite asynchronous period ends.

▶ **Definition 22** (Eventual lock-step synchronous model). *A consensus protocol works under an eventual lock-step synchronous (ELSS) model if there is a Global Stabilization Time, unknown to nodes and denoted as instant GST, such that after GST the protocol follows a lock-step synchronous model. In addition,*

- *Before instant GST, nodes also have three phases for each instant, but messages from the correct nodes can be delayed arbitrarily, i.e., in the receive phase of an instant, messages from any prior instant are not necessarily received.*

- *In the received phase of instant GST, correct nodes receive all messages from the correct nodes that were sent at or before GST.*

In a slot-sleepy model, each node goes through sleep-wake cycles each of which corresponds to a sequence of slots.

▶ **Definition 23** (Slot-sleepy model). *A consensus protocol works under a slot-sleepy* (SS) *model if for each slot $s \in \mathbb{N}$, each node is either slot-s awake, or slot-s asleep. If a node is slot-s awake, then the node has all three (receive, state update, and send) phases at every instant of slot s. If a node is slot-s asleep, then the node does not have any phase at any instant of slot s.*

▶ **Definition 24** (Oracle). *An oracle takes a slot number s as an input and sends at the send phase of instant $\langle s-1, f+2\rangle$ to each node the same random value $r = r(s)$. This value is unpredictable to any node before the receive phase of instant $\langle s, 1\rangle$ and has a uniform distribution over the set $\{1, \dots, n\}$. Every correct node receives this value at the next receive phase after GST.*

B Valid DAG

The concept of a valid DAG is needed to capture the essential properties of a DAG that is maintained by a correct node during the execution of Slipstream in one of the models considered in the paper. The following three sections delve into different definitions of valid DAGs, whereas the last one shows why a correct node always maintains a valid DAG.

B.1 Graph-valid DAG

A DAG $\mathcal{D} = (V, E)$ is considered *graph-valid* if it satisfies the following properties:

1. the vertex set V contains a unique vertex, denoted as $Genesis$, with no outgoing edges, i.e., which does not contain any hash references;
2. for any vertex $B \in V$, the set V contains all blocks that are referenced by B in B.refs;
3. for any two vertices B and C with $C \in B$.refs, the edge set E contains the directed edge (B, C).

B.2 Time-valid DAG

A DAG $\mathcal{D}$ is called *time-valid* if it satisfies the following property:

1. for any two vertices $B, C \in \mathcal{D}$ with $C \in B$.refs, it holds C.time $< B$.time.
2. there is only one vertex $Genesis \in \mathcal{D}$, that is created at slot 0; specifically, $Genesis.time = \langle 0, f+2\rangle$.

B.3 Digest-valid DAG

A DAG $\mathcal{D}$ is called *digest-valid* if it satisfies the following digest validity (DV) properties:

1. the digest of genesis $Genesis$.digest $= 0$;
2. every block $B \in \mathcal{D}$ with B.instant $< f+2$ contains a digest to slot B.slot -2, i.e., B.digest.slot $= B$.slot -2. If B.instant $= f+2$, B contains digest to slot B.slot -1;
3. for any block $B \in \mathcal{D}$, $B \neq Genesis$ it holds
   a. if B.instant $\notin \{1, f+2\}$, all referenced blocks $C \in B$.refs must satisfy C.digest $= B$.digest;

  b. if B.instant $= 1$, all blocks in B.refs can be divided into two groups Ref_1, $|\text{Ref}_1| \geq 1$, and Ref_2, $|\text{Ref}_2| \geq 0$, such that for any $C \in \text{Ref}_1$, C.digest $= B$.digest and for every $F \in \text{Ref}_2$ the digest field is the same (but different from B.digest);

  c. if B.instant $= f + 2$, all blocks $C \in B$.refs referenced by B must satisfy C.digest $=$ BeforeDigest(B.digest); see Def. 9;

4. a digest of every block $B \in \mathcal{D}$ with B.instant $= f + 2$ is computed correctly based on Cone(B) according to Def. 9.

B.4 Valid DAG of correct nodes

▶ **Definition 25** (Valid DAG). *A DAG is called valid if it is graph, time, and digest-valid.*

▶ **Lemma 26.** *For any given correct node, it holds that its local DAG $\mathcal{D}$ at the end of the state update phases is valid.*

Proof. Recall that the local DAG $\mathcal{D}$ at the end of the state update coincides with the past cone Cone(B_{own}) of the latest created block. This is straightforward to check for graph- and time-validity. In the remainder, we focus on digest-validity.

There are four different cases in the Alg. 1 to check. The first case is when instant $i \notin \{1, f + 2\}$ and UpdateDAG() is used. Note that before adding Cone(B) to the local DAG in line 18, we check the validity of Cone(B), see Def. 25. It remains to verify that the union of two valid DAGs is again a valid DAG. Conditions DV.1, DV.2, and DV.4 follow also directly from the fact that Cone(B) and $\mathcal{D}$ do satisfy them. Condition DV.3 is satisfied since we require in line 7 that B has the same digest as the local DAG. As the node is supposed to be honest and to follow the protocol, the new block B_{own}, created in line 30 in Alg. 1, only references blocks in $\mathcal{D} \cup$ Cone(B) which have the same digest and hence Cone(B_{own}) is valid.

The second case is where $i = 1$ and the node wakes up. Similar to the arguments in the UpdateDAG(), the validity properties are conserved by merging two valid DAGs. Note hereby that property DV.3 is true since we only accept blocks B having the same digest.

The third case, where $i = 1$ and the node switches the chain, is treated analogously. Let us only note here that, for property DV.3, we have to check that the new block B_{own}, satisfies DV.3.b. However, this condition is satisfied, as at most one block with a different digest is added.

The fourth case is where $i = f + 2$ and the backbone chain gets updated. This has no influence on graph and time validity; however, the new block B_{own} will contain a different digest. For an honest node following procedure UpdateChain, see line 25 in Alg. 3, properties DV.2, DV.3, and DV.4 hold true. ◀

C Formal results

▶ **Theorem 27** (Safety and liveness for Order$_{\text{own}}$). *Assume the network operates in the* SS *model, e.g., GST $= 0$. Assume a majority of slot-s awake correct nodes for all slots $s \in \mathbb{N}$, i.e., the number of correct nodes is strictly larger than the number of Byzantine nodes for every slot. Then it holds*

Safety: *Every slot-s awake correct node adopts the same digest for slot $s - 2$, i.e., for any two sequences of blocks maintained by any two correct nodes in* Order$_{\text{own}}$*, one sequence must be a prefix of the other.*

Liveness: *Every block created by a correct node at slot s is included in the digest of slot s, i.e., it appears in* Order$_{\text{own}}$ *after slot $s + 1$.*

▶ **Theorem 28** (Safety and liveness for $\textsc{Order}_{\text{final}}$). *Assume the network operates in the* ELSS *model (e.g., $GST > 0$ is unknown to nodes). Assume a supermajority of correct nodes, i.e., $n = 3f + 1$. Then it holds*
Safety: *For any two sequences of final blocks maintained by any two correct nodes in $\textsc{Order}_{\text{final}}$, one sequence must be a prefix of the other.*
Liveness: *Before GST, if a correct node creates a block, then the block eventually appears in $\textsc{Order}_{\text{final}}$. After GST, there is a slot s_{same} after which every block created by a correct node at slot $s \geq s_{\text{same}}$ appears in $\textsc{Order}_{\text{final}}$ after instant $\langle s+2, 2\rangle$. The number of slots $s_{\text{same}} - GST$ is stochastically dominated by $\xi_1 + \xi_2$, where ξ_1 and ξ_2 are independent and identically distributed geometric random variables with parameter $p = \frac{1}{n}$. In particular, the expected number of slots to sync correct nodes with the same slot digest satisfies $\mathbb{E}(s_{\text{same}}) \leq GST + 2n$.*

▶ **Theorem 29** (Safety, liveness, and consistency for Ledger). *Assume the network operates in an* ELSS *model. Assume a supermajority of correct nodes, i.e., $n = 3f + 1$. Then it holds*
Safety: *Two UTXO transactions that constitute a double-spend never both appear in* Ledger*.*
Liveness: *Every cautious honest UTXO transaction eventually added to* Ledger *through the consensus path. There is an instant after GST such that every cautious honest UTXO transaction, included in a block by a correct node at instant i, gets fast-path confirmed and appears in* Ledger *after instant $i + 2$.*
Consistency: *If a UTXO transaction is fast-path confirmed by a correct node, then it eventually appears in* Ledger *of every correct node.*

▶ Remark 30. The amortized communication complexity (for one transaction and big block size) in Slipstream in the ELSS model is $O(n^2)$, which is similar to all other DAG-based BFT protocols that don't use special erasure-code-based BRB primitives such as [8, 10]. Indeed, because of potentially n hash references to previous blocks, the block size in our protocol is (at least) linear with n, the number of nodes. Thereby, one can put n transactions in a block without affecting the order of the block size. Each block needs to be sent by every correct node to every other correct node, i.e., each block makes $O(n^2)$ trips between nodes. After GST, consider all transactions in s consecutive slots. For confirming the $s(f+2)n(n-f)$ transactions, committed to these s slots by $n - f$ correct nodes, it takes $O((s+1)(f+2)n^4)$ bits to be communicated, i.e., $O(n^2)$ bits per transaction.

D Proofs

D.1 Proof of Theorem 27

The key ingredient for Theorem 27 is the following key lemma, which can be applied to both the ELSS model and the SS model. It states that under synchrony, two correct nodes starting the slot with the same slot digest will generate identical digests at the end of the slot. The proof of this lemma is inspired by [11], and it makes use of the lock-step nature of the network models considered in the paper.

▶ **Lemma 31.** *Suppose slot s occurs after GST. If two correct nodes have adopted the same digest for slot $s-2$ after completing the state update phase of instant $\langle s, 1\rangle$, then they generate the same digest for slot $s - 1$ after completing the state update phase of instant $\langle s, f+2\rangle$.*

Proof. Let η and μ be two correct nodes that adopted the slot digest σ_{s-2} after the state update phase of instant $\langle s, 1\rangle$. Consider an arbitrary block B which is created at slot $s - 1$ or before and included by node η in the digest σ_{s-1} for slot $s - 1$. Toward a contradiction, assume that B is not included by μ in the digest for slot $s - 1$. Let i be the first instant of slot s during the state update phase, of which η has added B to its maintained DAG. Note that B was added together with the past cone of a certain block, say block E (see line 18 of Alg. 2). Technically, that could happen at slot $s-1$, and we treat i as 1 in this case. The block B satisfies the condition U3 in the perception of node η. Thus, it holds that $\textsc{ReachNumber}(B, E) \geq i-1$. Next, consider two cases depending on the value of i.

First case: Consider $1 \leq i \leq f+1$. Then, at the send phase of instant $\langle s, i\rangle$, node η creates block C such that $\text{Cone}(C)$ contains E (and consequently B) and sends it. At the next instant $\langle s, i+1\rangle$, node μ receives the block C with its past cone and updates the DAG with B, which leads to the contradiction. Indeed, the buffer of μ contains $\text{Cone}(C)$ and all the U-conditions (see Sec. 4.6) are met:
U1: No block at slot s in $\text{Cone}(C)$ is created by equivocator from $\text{EqSet}(\sigma_{s-2})$ since $C.\text{node} = \eta$ and η is a correct node that checked the same condition in all the previous instants of slot s.
U2: The past cone of block C is valid due to Lem. 26.
U3: The reachable number $\textsc{ReachNumber}(B, C) \geq i$, as the node η was not counted previously. Similarly, one can show that the reachable number of all other blocks in $\text{Cone}(C)$, which are not yet included in the DAG of μ, satisfies the same condition U3.

Second case: Consider $i = f+2$. In this case, at the time of adding B by η to its maintained DAG, the reachable number $\textsc{ReachNumber}(B, E) \geq i-1 = f+1$. This means that B was added to the DAG by at least one correct node at or before instant $\langle s, f+1\rangle$. By applying the arguments from the first case, we conclude that μ has included B in its maintained DAG at or before instant $\langle s, f+2\rangle$. This leads to a contradiction.

The above arguments imply that after the state update phase of the instant $\langle s, f+2\rangle$ both nodes η and μ have in their DAGs the same set of blocks that are created at slot $s-1$ or before. Thereby, they generate the identical digest σ_{s-1}. ◀

Proof of Theorem 27. First, we prove the safety property. Let η be a slot-$(s+1)$ awake correct node. We proceed by an induction on s. The base case $s=0$ holds as the digest σ_{-1} is fixed and known to all nodes. In the following, consider $s \geq 1$. Consider two cases depending on whether node η was awake or asleep at the previous slot.

First case: Suppose η was slot-s awake and adopted digest σ_{s-2} in slot s. By Lem. 31, η generated the same digest for slot $s-1$ at the end of slot s as all other awake correct nodes who adopted σ_{s-2} at the beginning of slot s. By the inductive hypothesis, the number of such correct nodes holds the majority among all slot-s awake nodes, i.e., $2N_{\text{same}} \geq N_{\text{total}} + 1$. By the inductive hypothesis, the indicator I_{ELSS} will never change its value from 0 in the SS model. Indeed, this indicator can be changed in lines 27 and 58 of Alg. 4. As for line 27, it is not possible in the SS model to construct a digest certificate conflicting with the current digest of a correct node, i.e., $\textsc{IsConflict}(\textsc{Digest}(\text{DC}_{\text{leader}}), B_{\text{own}}.\text{digest}) = 0$. As for line 58, it is not possible to have two groups of blocks each of size at least $f+1$ including different digests at the end of any slot because all awake correct nodes always generate the same digests. Thus, when node η will call $\textsc{SwitchChain}()$ (see Alg. 4), it will not pass the if-condition (see line 34) as $I_{\text{ELSS}} = 0$. Consequently, it will not switch the backbone chain after the state update phase of instant $\langle s+1, 1\rangle$.

Second case: If η was slot-s asleep, then at the beginning of slot $s+1$, η calls the procedure $\textsc{WakeUpChain}()$ (see Alg. 4) and adopts the same digest as the majority of nodes generated at the end of slot s.

In both cases, η adopts the same digest as the majority of nodes produced at the end of slot s. This completes the proof of the inductive step and the proof of the safety property.

The liveness property follows directly from the synchronous nature of the SS model and the fact that every block sent by a correct node is received and added to a local DAG by any other correct node at the next instant. ◀

D.2 Proof of Theorem 28

D.2.1 Safety

We start with a simple observation concerning digest certificates. We will make use of the property that $2f+1$ out of $3f+1$ nodes are correct.

▶ **Lemma 32.** *For any two digest certificates A and B with A.slot $= B$.slot, the digests certified by A and B are the same.*

Proof. Let $s = A$.slot $= B$.slot. By Def. 13, a quorum of blocks created at instants $\langle s, 1\rangle, \dots, \langle s, f+1\rangle$ is needed for a digest certificate. Thus, at least one correct node created a block (or blocks) at slot s that is (are) included in both Cone(A) and Cone(B). This implies that A and B certify the same digest. ◀

▶ **Lemma 33.** *If a correct node has finalized a slot digest σ at slot s, then at least $f+1$ correct nodes have created blocks at slot s serving as digest certificates for σ.*

Proof. The statement follows directly from Def. 14. Indeed, the finalization of digest σ could happen only when the correct node has in its DAG a quorum, i.e., $2f+1$, of blocks such that each block from this quorum is a DC for σ. Among these $2f+1$ blocks, $f+1$ were created by correct nodes. ◀

▶ **Lemma 34** (Safety of finality)**.** *Suppose a correct node has finalized a slot digest σ. Then no other correct node finalizes a slot digest conflicting with σ.*

Proof. Toward a contradiction, assume that a conflicting digest θ is final for another correct node. Without loss of generality, assume σ gets final at an earlier instant, say slot s, by a given correct node. By Lem. 33, $f+1$ correct nodes created digest certificates for σ at slot s. By Lem. 32, all digest certificates at slot s certify the same digest. By the chain switching rule (procedure SwitchChain() in Alg. 4), no node of the $f+1$ correct nodes, created DCs, could switch its backbone chain to a one conflicting with σ unless it received a digest certificate issued at a slot greater than s.

Let B be a digest certificate created at a slot r such that Digest(B) is conflicting with σ and the slot $r > s$ is minimal among all such digest certificates. Such a certificate should exist since the conflicting digest θ gets final at some slot. At least one of the said $f+1$ correct nodes had to contribute a block to form a digest certificate B. This means that this correct node has switched its chain at or before slot r to the one conflicting with σ. But this could happen only if there would exist a digest certificate that conflicts with σ and is issued before slot r. This contradicts the minimality of r. ◀

D.2.2 Liveness

▶ **Lemma 35.** *Suppose after instant GST, all correct nodes adopt the same slot digest σ_{s-2} for slot s. Then all correct nodes finalize the digest σ_{s-2} at the state update phase of instant $\langle s, 3\rangle$.*

Proof. Every block created by one of the correct nodes at instant i is added to the DAG maintained by any other correct node at instant $i+1$ by the arguments of Lem. 31. Correct nodes adopt the digest σ_{s-2} for the whole slot s and reference all previous blocks by the correct nodes. Hence, the $2f+1$ blocks created by correct nodes at instant $\langle s, 2\rangle$ form a quorum of digest certificates for σ_{s-2}. By Def. 14, all correct nodes will finalize σ_{s-2} at the state update phase of instant $\langle s, 3\rangle$. ◀

Now we proceed with a statement showing when all correct node switch to the same backbone chain after GST.

▶ **Lemma 36** (Liveness of finality). *After GST, let s_{same} denote the first slot when all correct nodes adopt the same slot digest. Then $s_{\text{same}} - GST$ is stochastically dominated by a random variable which is a sum of two independent random variables having geometric distribution* $\text{Geo}(1/n)$*. In particular, the expected number of slots it takes until all correct nodes follow the same backbone chain satisfies $\mathbb{E}(s_{\text{same}}) \le GST + 2n$.*

Proof. Let $s+1$ be an arbitrary slot after GST and before moment s_{same}. Let μ be a correct node with a digest certificate generated at the highest slot number among all correct nodes. Note that after GST, the function LEADER() returns one of the nodes (not $\perp$). With probability $1/n$, node μ will be chosen as a leader based on the value of a common random coin for slot $s+1$. For every other correct node η, it holds $\text{DC}_{\text{leader}}.\text{slot} \ge \text{DC}_{\text{own}}.\text{slot}$. Let us check all potential scenarios when both η and μ will not end up on the same backbone chain after running at $\langle s+1, 1\rangle$ procedure SWITCHCHAIN(), see Alg. 4:

1. node η has finalized a digest at the previous slot, i.e., $s_{\text{final}} = s - 2$ (see line 20 in Alg. 4);
2. node η has $s_{\text{final}} \neq s - 2$ and $2N_{\text{same}} \ge N_{\text{total}} + 1$ (see line 34 in Alg. 4).

In the first case, η and μ have digest certificates created at the same slot s and certifying the same digest σ_{s-2} by Lem. 32. In addition, they generated the same digests for slot $s-1$ by Lem. 31, i.e., η will adopt the same digest for slot $s+1$ as μ.

In the second case, after GST, there could be some *majority* groups of correct nodes that perceive $2N_{\text{same}} \ge N_{\text{total}} + 1$, where correct nodes within each group have the same digest. Note that in this case, $N_{\text{total}} \ge 2f + 1$ since blocks from all correct nodes will be received after GST. Therefore, N_{same} has to be at least $f+1$. If there are two such majority groups, then all correct nodes will learn about that at the slot $s+1$ and change the indicator I_{ELSS} to 1 when running SWITCHCHAIN() at the next slot $s+2$ (see line 18). Thus, we can assume that there is only one majority group of correct nodes (including η) that perceive $2N_{\text{same}} \ge N_{\text{total}} + 1$ and η has $I_{\text{ELSS}} = 0$ (the latter condition does not allow η to switch its chain to the one of the leader μ). However, this also means that digests DIGEST($\text{DC}_{\text{leader}}$) and B_{own}.digest are not conflicting (see line 27). In such a case, we note that with probability $1/n$, node η could be selected as a leader for slot $s+1$ and then μ (as all other correct nodes outside the majority group) would switch its backbone chain as all required if-conditions in line 29 are satisfied.

It remains to compute the expected number of slots after GST, which is sufficient for all correct nodes to switch to one chain. With a probability of at least $1/n$, a proper correct node will be selected as a leader, resulting in switching all correct nodes to one chain. However, the adversary controlling the Byzantine nodes has a one-time opportunity to create at least two majority groups that will not allow all correct nodes to adopt one chain. In the worst case, the adversary could use this one-time option only when a proper correct leader is chosen. In this case, all correct nodes set their indicators I_{ELSS} to 1 at the end of the slot and switch their backbone chains to the one of a proper correct leader next time. Thereby, the moment $s_{\text{same}} - GST$ is stochastically dominated by a sum of two independent random variables having a geometric distribution with probability $1/n$. ◀

▶ **Theorem 37** (Liveness of finality). *Let $s_{\text{same}} > GST$ be defined as in* Lem. 36*. Then after finishing any slot $s \ge s_{\text{same}}$, all correct nodes finalize the digest for slot $s - 2$.*

Proof. By Lem. 36, all correct nodes have adopted the same backbone chain before starting slot s_{same}. Then they produce the same digest for slot $s_{\text{same}} - 1$ by Lem. 31 and finalize the digest for slot $s_{\text{same}} - 2$ by Lem. 35. The same arguments can be applied for any other slot $s > s_{\text{same}}$. ◀

D.3 Proof of Theorem 29

D.3.1 Liveness

First, we show the liveness property of fast-path confirmation. By Lem. 36, at some moment s_{same} after GST all correct nodes end up on the same backbone chain. Then any cautious honest transaction tx, included in a block B after slot $s \geq s_{\text{same}}$, gets confirmed by ConfirmTxsFastPath() at the state update phase of instant B.time + 3. Indeed all correct nodes stay on the same backbone chain by Lem. 31 and add blocks created by correct nodes at one instant at the state update phase of the next instant. Since the transaction is honest and cautious, the blocks of correct nodes at instant B.time + 2 serve as a quorum of transaction certificates for tx in B, resulting in confirmation of tx in procedure ConfirmTxsFastPath().

Second, we prove the liveness property of the consensus-path confirmation. Any transaction included in a block B will be processed in the procedure ConfirmTxsConsensusPath() (see Alg. 5), when the finality time of slot B.slot is determined. Due to Lem. 36, the finality time progresses after GST, specifically, after slot s_{same}. If a cautious honest transaction has a transaction certificate in the DAG (restricted to the respected digest), then it will be added to the final ledger in the first for-loop of ConfirmTxsConsensusPath(); if not, it will happen in the second for-loop. For a cautious transaction, its inputs are already in Ledger; for an honest transaction, no conflicting transactions could appear in Ledger before processing this transaction.

D.3.2 Consistency

Let tx be a transaction in a block B such that tx in B gets confirmed through the fast path by a correct node. Hence, there exists a quorum of transaction certificates for tx in B, where each transaction certificate is created at slot B.slot or B.slot + 1.

Next we show that every correct node adds tx to Ledger through the consensus path. Let the finality time for B.slot be τ, i.e., FinalTime(B.slot) $= \tau$, see Def. 15. After the moment when a correct node updates the finality time with τ, this node will call ConfirmTxsConsensusPath() at line 48 in Alg. 3. When processing transaction tx in B in the first for-loop (see line 10) in ConfirmTxsConsensusPath(), there will be at least one transaction certificate TC in $\mathcal{D}(\sigma_\tau)$. Indeed, one correct node must contribute to both a quorum of transaction certificates for tx in B and a quorum of digest certificates when finalizing digest $\sigma_{\tau-2}$ at slot τ. By definition of approvals (Def. 6), the transactions creating inputs of tx must be already in Ledger and, thus, the if-condition at line 20 will be passed. It remains to check that tx is not conflicting with Ledger.

Toward a contradiction, assume that a conflicting transaction tx' is already in Ledger. First, recall that both tx and tx' can not have transaction certificates due to quorum intersection by at least one correct node. Thus tx' had a chance to get confirmed only when it was processed with block B' in the second for-loop (see line 17) in ConfirmTxsConsensusPath() at some point before, i.e., when the partition of the DAG corresponding to one of the previous finality times τ' with $\tau' < \tau$ was processed. There should be at least one correct node that contributed to both the finalization of digest $\sigma_{\tau'-2}$ (which commits block B' with tx') at slot τ' and approval of tx in block B. However, this could potentially happen only if B.slot $\leq \tau' - 1$ because of the definition of transaction approval (see Def. 6). This means both slots B.slot and B.slot + 1 are at most τ', and there should be at least one correct node with a transaction certificate for tx in B and contributing to the finality of $\sigma_{\tau'-2}$ at slot τ'. However, this means that tx would get final when processing the first loop (see line 10 in ConfirmTxsConsensusPath()) before tx' in B', i.e., tx' can not be added to Ledger.

D.3.3 Safety

Due to quorum intersection by at least one correct node, two conflicting transactions can not have transaction certificates and can not get both to Ledger through the fast path. When adding a transaction to Ledger in procedure ConfirmTxsConsensusPath(), we check a potential double-spend for the transaction in the current state of the ledger.

It remains to check the impossibility of the remaining case when a transaction tx is added to Ledger through the fast path and at the time of updating the ledger, tx is conflicting with some transaction $tx' \in \text{Ledger}$. Such a check is absent in procedure ConfirmTxsFastPath(). However, the same case was considered in the proof of consistency above, and its impossibility was already shown.

E Algorithms

Algorithm 3 Procedures to create a new block, update a backbone chain, and finalize a digest.

```
Local variables:
    $\mathcal{D} \leftarrow \{Genesis\}$                         // DAG maintained by the node
    $\text{Chain}_{\text{own}} \leftarrow (0)$                   // backbone chain adapted by the node
    $\text{pk}, \text{sk}$                                       // Public key and Private key of the node
    $s_{\text{final}} \leftarrow 0$                              // Slot index of the last final slot digest
    $s_{\text{pre}} \leftarrow 0$                                // Slot index of the last final slot digest in $\mathcal{D}(s_{\text{final}})$, see Def. 15
    $B_{\text{own}}$                                             // Last block created by the node
    $\textsc{Order}_{\text{own}}, \textsc{Order}_{\text{final}}$ // Optimistic and final orders of blocks
function $\textsc{CreateBlock}()$ :
    $B \leftarrow \{\}$
    $B.\text{refs} \leftarrow \textsc{Hash}(\textsc{Tips}())$   // Hash references to all unreferenced blocks in $\mathcal{D}$
    $B.\text{txs} \leftarrow \textsc{Payload}()$                // Subset of transactions
    $B.\text{digest} \leftarrow \textsc{CurrentDigest}()$
    $B.\text{time} \leftarrow \textsc{Now}()$                   // Current slot and instant indices
    $B.\text{node} \leftarrow \text{pk}$
    $B.\text{sign} \leftarrow \textsc{Sign}_{\text{sk}}(B)$      // Sign the content of block
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{B\}$              // Update the maintained DAG
    return $B$
function $\textsc{Tips}()$:
    return $\{B \in \mathcal{D} : \nexists C \in \mathcal{D} : \textsc{Hash}(B) \in C.\text{refs}\}$
function $\textsc{CurrentDigest}()$:
    return $\text{Chain}_{\text{own}}.last$                      // Last element of backbone chain
function $\textsc{IsQuorum}(Set)$:
    return $|B.\text{node} : B \in Set| \geq 2f + 1$
procedure $\textsc{UpdateChain}()$:
    $\langle s + 1, f + 2 \rangle \leftarrow \textsc{Now}()$
    $\sigma_{s-1} \leftarrow \textsc{CurrentDigest}()$
    $r \leftarrow \{\}$
    $Blocks_{\leq s} \leftarrow \{B \in \mathcal{D} \setminus \mathcal{D}(\sigma_{s-1}) : B.\text{slot} \leq s\}$
    for $B \in Blocks_{\leq s}$ do
        $r \leftarrow r || \textsc{Hash}(B)$
    $\sigma_s = \textsc{Hash}(\sigma_{s-1}, r)$
    $\text{Chain}_{\text{own}} = \text{Chain}_{\text{own}} || \sigma_s$   // Append the computed digest to the end
    $\textsc{Order}_{\text{own}} \leftarrow \textsc{Order}(\sigma_s)$
procedure $\textsc{FinalizeSlots}()$:
    $\langle s, i \rangle \leftarrow \textsc{Now}()$
    for $t \in [s_{\text{final}} + 3, s]$ do
        $\sigma_t \leftarrow \text{Chain}_{\text{own}}[t]$
        $Blocks_{=t} \leftarrow \{B \in \mathcal{D}(\sigma_t) : B.\text{slot} = t\}$
        $\text{DC}_{=t} \leftarrow \{B \in Blocks_{=t} : B \text{ is a DC for } \sigma_{t-2}\}$   // DCs from slot $t$ (see Def. 13)
        if $\textsc{IsQuorum}(\text{DC}_{=t})$ then
            $s_{\text{final}} \leftarrow t - 2$
            $\textsc{Order}_{\text{final}} \leftarrow \textsc{Order}(\sigma_{t-2})$
            $\tau \leftarrow \textsc{FinalTime}(s_{\text{pre}} + 1)$   // see Def. 15
            while $\tau \neq \perp$ do
                $\sigma \leftarrow \text{Chain}_{\text{own}}[\tau]$
                $s_{\text{pre}} \leftarrow \textsc{LastFinal}(\mathcal{D}(\sigma))$   // Slot index of last final digest in $\mathcal{D}(\sigma)$
                $\textsc{ConfirmTxsConsensusPath}()$                  // see Alg. 5
                $\tau \leftarrow \textsc{FinalTime}(s_{\text{pre}} + 1)$
```

Algorithm 4 Procedures to adopt a chain.

```
Local variables:
    $\mathcal{D}$                                             // DAG maintained by the node
    $\text{Chain}_{\text{own}}$                               // backbone chain adapted by the node
    $B_{\text{own}}$                                          // Last block created by the node
    $I_{\text{ELSS}} \leftarrow 0$                            // Indicator that detects an ELSS model
    $s_{\text{final}}$                                        // Slot index of the last final slot digest
    EqSet                                                     // Set of equivocators known to the node
// The procedure is used to adopt a chain when the node was slot-$s$ awake
procedure SwitchChain() :
    $\langle s+1, 1\rangle \leftarrow \text{Now}()$
    $\sigma_{s-1} \leftarrow \text{CurrentDigest}()$
    $N_{\text{same}} \leftarrow 0, N_{\text{total}} \leftarrow 0$
    for $node \in \{1,2,\dots,n\} \setminus \text{EqSet}$ do
        $B \leftarrow \text{LastBlockFrom}(node)$
        if $B.\text{time} = \langle s, f+2\rangle$ then
            $N_{\text{total}} \leftarrow N_{\text{total}} + 1$
            if $B.\text{digest} = \sigma_{s-1}$ then
                $N_{\text{same}} \leftarrow N_{\text{same}} + 1$
    CheckELSS()
    $\text{leader} \leftarrow \text{Leader}()$
    if $s_{\text{final}} \neq s-2 \wedge \text{leader} \neq \perp$ then
        $B_{\text{leader}} \leftarrow \text{LastBlockFrom}(\text{leader})$
        if $\neg\text{IsValid}(\text{Cone}(B_{\text{leader}}))$ then
            break
        $\text{DC}_{\text{leader}} \leftarrow \text{LastDigestCertificate}(B_{\text{leader}})$    // Last DC created by the block creator in block's causal history
        $\text{DC}_{\text{own}} \leftarrow \text{LastDigestCertificate}(B_{\text{own}})$
        if $\text{IsConflict}(\text{Digest}(\text{DC}_{\text{leader}}), B_{\text{own}}.\text{digest})$ then
            $I_{\text{ELSS}} \leftarrow 1$
        if $2N_{\text{same}} \leq N_{\text{total}}$ then
            if $\neg\text{IsConflict}(B_{\text{leader}}.\text{digest}, \text{Digest}(\text{DC}_{\text{own}})) \vee \text{DC}_{\text{leader}}.\text{slot} \geq \text{DC}_{\text{own}}.\text{slot}$ then
                $\mathcal{D} \leftarrow \mathcal{D} \cup \text{Cone}(B_{\text{leader}})$
                $\text{Chain}_{\text{own}} \leftarrow \text{Chain}(B_{\text{leader}}.\text{digest})$
                $\text{ORDER}_{\text{own}} \leftarrow \text{ORDER}(B_{\text{leader}}.\text{digest})$
        else
            if $I_{\text{ELSS}} = 1 \wedge \text{DC}_{\text{leader}}.\text{slot} \geq \text{DC}_{\text{own}}.\text{slot}$ then
                $\mathcal{D} \leftarrow \mathcal{D} \cup \text{Cone}(B_{\text{leader}})$
                $\text{Chain}_{\text{own}} \leftarrow \text{Chain}(B_{\text{leader}}.\text{digest})$
                $\text{ORDER}_{\text{own}} \leftarrow \text{ORDER}(B_{\text{leader}}.\text{digest})$
// The procedure to adopt a chain when the node was slot-$s$ asleep
procedure WakeUpChain():
    $\langle s+1, 1\rangle \leftarrow \text{Now}()$
    $M \leftarrow \{\}$                                       // Multiset of digests
    for $node \in \{1,2,\dots,n\} \setminus \text{EqSet}$ do
        $B \leftarrow \text{LastBlockFrom}(node)$
        if $B.\text{time} = \langle s, f+2\rangle$ then
            $M \leftarrow M \cup \{B.\text{digest}\}$
    $\sigma_{s-1} \leftarrow \text{Mode}(M)$                  // Most present element in the multiset with deterministic tiebreaker
    $\text{Chain}_{\text{own}} \leftarrow \text{Chain}(\sigma_{s-1})$
    $\text{ORDER}_{\text{own}} \leftarrow \text{ORDER}(\sigma_{s-1})$
    for $node \in \{1,2,\dots,n\} \setminus \text{EqSet}$ do
        $B \leftarrow \text{LastBlockFrom}(node)$
        if $\text{IsValid}(\text{Cone}(B)) \wedge B.\text{digest} = \sigma_{s-1}$ then
            $\mathcal{D} \leftarrow \mathcal{D} \cup \text{Cone}(B)$
// The procedure is used to update the variable indicating the ELSS model
procedure CheckELSS():
    $\langle s+1, 1\rangle \leftarrow \text{Now}()$
    $M \leftarrow \{\}$                                       // Multiset of digests
    for $B \in \text{Buffer}$ s.t. $B.\text{time} = \langle s-1, f+2\rangle$ do
        $M \leftarrow M \cup \{B.\text{digest}\}$
    if $\exists \sigma, \theta \in M$ s.t. $\text{Num}(\sigma, M) \geq f+1 \wedge \text{Num}(\theta, M) \geq f+1$ then
        $I_{\text{ELSS}} \leftarrow 1$
// The function returns the number of repetitions of an element in a multiset
function Num($\sigma, M$):
    $count \leftarrow 0$
    for $\beta \in M$ do
        if $\beta = \sigma$ then
            $count \leftarrow count + 1$
    return $count$
```

Algorithm 5 Procedures to update the ledger.

```
Local variables:
    $\mathcal{D}$                                              // DAG maintainained by the node
    Ledger                                                     // Confirmed ledger maintained by the node
    $s_{\text{final}} \leftarrow 0$                            // Slot index of the last final slot digest
    $s_{\text{pre}} \leftarrow 0$                              // Slot index of the last final slot digest in $\mathcal{D}(s_{\text{final}})$, see Def. 15
    ProcTxCertificate $\leftarrow \{\}$                        // Processed blocks with transactions for which transactions certificates were checked
    ProcTotalOrder $\leftarrow \{\}$                           // Processed blocks with transactions which were resolved using the total order
// Procedure to confirm transaction through the consensus path
procedure ConfirmTxsConsensusPath():
    $\tau \leftarrow$ FinalTime($s_{\text{pre}}$)              // see Def. 15
    // First add transactions with transaction certificates (see Def. 7). Use Order$_{\text{final}}$ for the for-loop.
    for $B \in \mathcal{D}(\sigma_\tau) \setminus$ ProcTxCertificate s.t. $B$.slot $\leq \tau - 2$ do
        ProcTxCertificate $\leftarrow$ ProcTxCertificate $\cup \{B\}$
        for $tx \in B$.txs do
            if $\exists C \in \mathcal{D}(\sigma_\tau)$ s.t. $C$ is a TC for $tx$ in $B$ then
                if $tx$ has inputs in Ledger then
                    if $tx$ is not conflicting with Ledger then
                        Ledger $\leftarrow$ Ledger $\cup \{tx\}$
    // Use total order to resolve remaining conflicts
    for $B \in \mathcal{D}(\sigma_{\tau-2}) \setminus$ ProcTotalOrder do
        ProcTotalOrder $\leftarrow$ ProcTotalOrder $\cup \{B\}$
        for $tx \in B$.txs do
            if $tx$ has inputs in Ledger then
                if $tx$ is not conflicting with Ledger then
                    Ledger $\leftarrow$ Ledger $\cup \{tx\}$
// Procedure to confirm transaction through the fast-path
procedure ConfirmTxsFastPath() :
    for $B \in \mathcal{D}$ do
        for $tx \in B$.txs do
            TC$_{\text{all}} \leftarrow$ AllTxCertificates($tx, B$)
            if IsQuorum(TC$_{\text{all}}$) then
                Ledger $\leftarrow$ Ledger $\cup \{tx\}$
function AllTxCertificates($tx, B$):
    $M \leftarrow \{\}$                                        // Multiset of transaction certificates
    for $C \in \mathcal{D}$ s.t. $0 \leq C$.slot $- B$.slot $\leq 1$ do
        if $C$ is a TC for $tx$ in $B$ then
            $M \leftarrow M \cup \{C\}$
    return $M$
```