# pod: An Optimal-Latency, Censorship-Free, and Accountable Generalized Consensus Layer

**Orestis Alpos**
Common Prefix, Athens, Greece

**Bernardo David**
IT University of Copenhagen (ITU), Denmark
Common Prefix, Athens, Greece

**Jakov Mitrovski**
Technical University of Munich, Germany
Common Prefix, Athens, Greece

**Odysseas Sofikitis**
Common Prefix, Athens, Greece
pod network, Athens, Greece

**Dionysis Zindros**
Common Prefix, Athens, Greece
pod network, Athens, Greece

## Abstract

This work addresses the inherent issues of high latency in blockchains and low scalability in traditional consensus protocols. We present pod, a novel notion of consensus whose first priority is to achieve the physically-optimal latency of $2\delta$, or one round-trip, *i.e.*, requiring only one network trip (duration $\delta$) for writing a transaction and one for reading it.

To accomplish this, we first eliminate inter-replica communication. Instead, clients send transactions directly to all replicas, which independently process transactions and append them to local logs. Replicas assign a timestamp and a sequence number to each transaction in their logs, allowing clients to extract valuable metadata about the transactions and the system state. Later on, clients retrieve these logs and extract transactions (and associated metadata) from them.

Necessarily, this construction achieves weaker properties than a total-order broadcast protocol, due to existing lower bounds. Our work models the primitive of pod and defines its security properties. We then show pod-core, a protocol that satisfies properties such as transaction confirmation within $2\delta$, censorship resistance against Byzantine replicas, and accountability for safety violations. We show that single-shot auctions can be realized using the pod notion and observe that it is also sufficient for other popular applications.

**2012 ACM Subject Classification** Security and privacy → Distributed systems security; Computer systems organization → Dependable and fault-tolerant systems and networks

**Keywords and phrases** consensus, censorship resistance, accountability, auctions

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2025.4

**Related Version** *Full Version*: https://arxiv.org/abs/2501.14931 [2]

## 1 Introduction

Despite the widespread adoption of blockchains, a significant challenge remains unresolved: they are inherently slow. The latency from the moment a client submits a transaction to when it is confirmed in another client's view of the blockchain can be prohibitively long for certain applications. Notice that we define latency in terms of the blockchain *liveness* property, referring to finalized, non-reversible outputs: once a transaction is received by a reader, it


remains in the protocol's output permanently. Moreover, we do not assume "optimistic" or "happy path" scenarios, where transactions might finalize faster under favorable conditions (such as having honest leaders or optimal network conditions).

Indeed, Nakamoto-style blockchain protocols require a large number of rounds in order to achieve consensus on a new block, even when considering the best known bounds [15]. On the other hand, it is known that permissioned protocols for $n$ parties (out of which $t$ are corrupted) realizing traditional notions of broadcast and Byzantine agreement require at least $t+1$ rounds in the synchronous case [1] and at least $2n/(n-t)$ rounds in the asynchronous case [14], even when allowing for digital signatures and probabilistic termination.

In a model where *replicas* maintain the network, *writers* submit transactions, and *readers* read the network, the minimum latency is one network round trip, or $2\delta$, letting $\delta$ denote the actual network delay, as the information must travel from the writers to the replicas and then to the readers. More importantly, we want that any transaction from an honest writer appears in the output of honest readers within $2\delta$ time, regardless of the current value of $\delta$ and corrupted parties' actions. In this context, we are motivated by the following question:

*Can we realize tasks that blockchains are commonly used for with optimal latency?*

We give a positive answer to this question with a protocol realizing pod, a new notion of consensus that trades off traditional agreement properties for optimal latency, while retaining sufficient security guarantees to realize important tasks (*e.g.*, decentralized auctions).

## 1.1 Our Contributions

In order to motivate the notion of pod, we first introduce the architecture of our protocol, pod-core, which realizes this notion. To achieve the single-round-trip latency, our first key design decision is to eliminate inter-replica communication entirely. Instead, writers send their transactions directly to all replicas. Each replica maintains its own *replica log*, processes incoming transactions independently, and transmits its log to readers on request. Readers then process these replica logs to extract transactions and relevant associated information. See Figure 1 for a summary of the pod-core architecture.

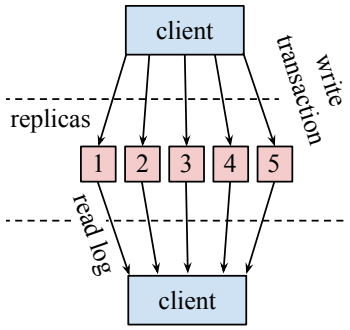


**Figure 1** pod-core's simple architecture. A writing client (top) sends a transaction to all replicas (middle). Each replica appends it to its own log and transmits it to the reading client (bottom).

This design raises two important questions. First, what meaningful information can readers derive from replica logs when replicas operate in isolation? Second, given that in two rounds even randomized authenticated broadcast is proven impossible [14], what capabilities can this – necessarily weaker – primitive offer? We demonstrate that, by incorporating simple mechanisms, such as assigning timestamps and sequence numbers to transactions, replicas

can enable readers to extract valuable information beyond mere low-latency guarantees. Furthermore, we show how the properties of pod can enable various applications, including auctions (as shown in Section 6). Specifically, a secure pod delivers the following guarantees (formally defined in Section 3):

- Transaction confirmation within $2\delta$, with each transaction assigned a *confirmed round*: we say that the transaction becomes *confirmed* at the time indicated by the *confirmed round*.
- Censorship resistance when facing up to $\beta$ Byzantine and $\gamma$ omission-faulty replicas, ensuring all confirmed transactions appear in every honest reader's output.
- A *past-perfect round* can be computed by readers, such that the reader is guaranteed to have received all transactions that are or will be confirmed prior to this round, even though not all transactions are strictly ordered.
- Accountability for all safety violations, *i.e.*, if any safety property is violated, at least $\beta + 1$ replicas can be identified as misbehaving.

In particular, our Protocol pod-core, presented in Section 4, realizes the notion of pod with the properties above, supporting a continuum of two adversarial models: up to $\beta$ Byzantine replicas and up to $\gamma$ omission-faulty replicas, out of a total of $n > 5\beta + 3\gamma$ replicas. Protocol pod-core requires no expensive cryptographic primitives or setup beyond digital signatures and a PKI registering replicas' public keys. We showcase pod-core's efficiency by means of experiments with a prototype implementation presented in Section 5. Our experiments show that even with 1000 replicas distributed around the world, the latency achieved by our protocol is just under double (resp. about 5 times) the round-trip time between writer and reader clients with security against omission-faulty (resp. Byzantine) replicas.

## 1.2 Technical Overview

We consider that time proceeds in *rounds*, and that parties (replicas and clients) know the current round, so we can express *timestamps* in terms of rounds. The output of pod associates each transaction tx with timestamp values $\mathsf{r}_{\min} \geq 0$ (minimum round), $\mathsf{r}_{\max} \leq \infty$ (maximum round) and $\mathsf{r}_{\text{conf}}$ (*confirmed round*). We call these values the *trace* of tx, and they evolve over time. Initially we have $\mathsf{r}_{\text{conf}} = \perp$ but later we get $\mathsf{r}_{\text{conf}} \neq \perp$, when a transaction is *confirmed*. The protocol guarantees *confirmation within $u$ rounds*, meaning that, at most $u$ rounds after tx was written, every party who reads the pod will see tx as confirmed with some $\mathsf{r}_{\text{conf}} \neq \perp$. The protocol also guarantees that $\mathsf{r}_{\min} \leq \mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\max}$, a property we call *confirmation bounds*: while each party reads different values $\mathsf{r}_{\min}, \mathsf{r}_{\max}, \mathsf{r}_{\text{conf}}$ for the same tx, pod guarantees that values read by different parties stay within these limits.

When clients *read* the pod, they obtain a pod data structure $D = (\mathsf{T}, \mathsf{r}_{\text{perf}})$, where T is the set of transactions and their traces and $\mathsf{r}_{\text{perf}}$ is a *past-perfect* round. The *past-perfection* safety property guarantees that T contains *all* transactions that every other honest party will ever read with a confirmed round smaller than $\mathsf{r}_{\text{perf}}$. A pod also guarantees *past-perfection within $w$*, meaning that $\mathsf{r}_{\text{perf}}$ is at most $w$ rounds in the past.

In summary, pod provides *past-perfection* and *confirmation bounds* as safety properties, ensuring parties cannot be blindsided by transactions suddenly appearing as confirmed too far in the past, and that the different (and continuously changing) transaction timestamps stay in a certain range. The liveness properties of *confirmation within $u$* and *past-perfection within $w$* ensure that new transactions get confirmed within a bounded delay, and that each party's past-perfect round must be constantly progressing.

Besides introducing the notion of pod, we present protocol pod-core, which realizes this notion while requiring minimal interaction among parties and achieving optimal latency, *i.e.*, optimal parameters $u = 2\delta$ and $w = \delta$, where $\delta$ is the current network delay (not a delay upper bound, which we assume to be unknown). The only communication is between each client and the replicas. Writing a transaction tx to pod-core only requires clients to send tx to the replicas, who each assign a timestamp ts (their current time) and a *sequence number* sn to tx and return a signature on $(\mathsf{tx}, \mathsf{ts}, \mathsf{sn})$. When reading the pod, the client simply requests each replica's log of transactions, validates the responses, and determines $\mathsf{r}_{\min}$, $\mathsf{r}_{\max}$, and $\mathsf{r}_{\text{conf}}$ from the received timestamps. Protocol pod-core supports a continuum of mixed adversarial models, tolerating up to $\beta$ Byzantine *and* at the same time up to $\gamma$ additional omission-faulty replicas.

## 1.3 Related work

Many previous works have lowered the latency of ordering transactions. HotStuff [27] uses three rounds of all-to-leader and leader-to-all communication pattern, which results in a latency (measuring from the moment a client submits a transaction until in appears in the output of honest replicas) of $8\delta$ in the happy path. Jolteon [16], Ditto [16], and HotStuff-2 [19] are two-round versions of HotStuff with end-to-end latency of $5\delta$. MoonShot [10] allows leaders to send a new proposal every $\delta$ time, before receiving enough votes for the previous one, but still achieves an end-to-end latency of $5\delta$. In the "DAG-based" line of word, Tusk [8] achieves and end-to-end latency of $7\delta$, the partially-synchronous version of BullShark [23] an end-to-end latency of $5\delta$, and Mysticeti [3] an end-to-end latency of $4\delta$. All these protocols aim at total-order properties and have their lower latency is inherently restricted by lower bounds, whereas pod starts from the single-round-trip latency requirement and explores the properties that can be achieved.

The redundancy of consensus for implementing payment systems has been recognized by previous works [18, 22, 7, 5]. The insight is that total transaction order is not required in the case that each account is controlled by one client. Instead, a partial order is sufficient, ensuring that, if transactions $\mathsf{tx}_1$ and $\mathsf{tx}_2$ are created by the same client, then every party outputs them in the same order. This requirement was first formalized by Guerraoui *et al.* [18] as the *source-order property*. The constructions of Guerraoui *et al.* [18] and FastPay [5] require clients to maintain sequence numbers. ABC [22] requires clients to reference all previous transaction in a DAG (including its own last transaction). Cheating clients might lose liveness [5, 18, 22], but equivocating is not possible. To the best of our knowledge, previous work in the consensusless literature has not considered or achieved a property like our *past-perfection*, which we show sufficient for implementing auctions.

# 2 Preliminaries

**Notation.** We denote by $\mathbb{N}$ the set of natural numbers including 0. Letting $L$ be a bounded sequence, we denote by $L[i]$ the $i^{\text{th}}$ element (starting from 0), and by $|L|$ its length. Negative indices address elements from the end, so $L[-i]$ is the $i^{\text{th}}$ element from the end, and $L[-1]$ in particular is the last. The notation $L[i{:}]$ means the subarray of $L$ from $i$ onwards, while $L[{:}j]$ means the subsequence of $L$ up to (but not including) $j$. We denote an empty sequence by $[\,]$. We denote the concatenation of sequences $L_1$ and $L_2$ by $L_1 \,\|\, L_2$.

**Pseudocode notation.** Command "**require** $P$" causes a function to terminate immediately and return false if $P$ evaluates to false. Notation "**upon** $e$" causes a block of code to be executed when event $e$ occurs. Notations "$\langle MSG \rangle \leftarrow p$" and "$\langle MSG \rangle \rightarrow p$" denote receiving

and sending a message $MSG$ from and to party $p$, respectively. Finally, $x : a \in A \to b \in B$ denotes that variable $x$ is a map from elements of type $A$ to elements of type $B$. When obvious from the context, we do not explicitly write the types $A$ or $B$. For a map $x$, the operations $x.keys()$ and $x.values()$ return all keys and all values in $x$, respectively. With $\emptyset$ we denote an empty map.

**Parties.** We consider $n$ *replicas* $R = \{R_1, \ldots, R_n\}$ and an unknown number of *clients*. Parties are *stateful*, *i.e.*, store *state* between executions of different algorithms. We assume that replicas are known to all parties and register their public keys (for which they have corresponding secret keys) in a Public Key Infrastructure (PKI). Clients do not register keys in the PKI.

**Adversarial model.** We call a party (replica or client) *honest*, if it follows the protocol, and *malicious* otherwise. We assume *static corruptions*, *i.e.*, the set of malicious replicas is decided before the execution starts and remains constant. This work uses a combination of two adversarial models, the *Byzantine* and the *omission* models. In the *Byzantine* model, corrupted replicas are malicious and may deviate arbitrarily from the protocol. The adversary has access to the internal state and secret keys of all corrupted parties. We denote by $\beta \in \mathbb{N}$ the number of Byzantine replicas in an execution. The Byzantine adversary is modelled as a probabilistic polynomial time overarching entity that is invoked in the stead of every corrupted party. That is, whenever the turn of a corrupted party comes to be invoked by the environment, the adversary is invoked instead. In the *omission* model, corrupted replicas may only deviate from the protocol by dropping messages that they were supposed to send, but follow the protocol otherwise. Observe that this includes crash faults, where replicas crash (*i.e.* stop execution) and remain crashed until the end of the execution of an algorithm. We denote by $\gamma \in \mathbb{N}$ the number of omission-faulty replicas in an execution.

**Modeling time.** We assume that time proceeds in discrete *rounds*, and parties have clocks allowing them to determine the current round. For simplicity, our analysis will assume *synchronized clocks*. Notice that although we assume synchronized clocks as a setup, clock synchronization can be achieved in partially synchronous networks [11] using existing techniques [21], also in the case where replicas gradually join the network [26]. By *timestamp* we refer to a round number assigned to some event.

**Modeling network.** We denote by $\delta \in \mathbb{N}$ the actual delay (measured in number of rounds) it takes to deliver a message between two honest parties, a number which is *finite* but *unknown* to all parties. We denote by $\Delta \in \mathbb{N}$ an upper bound on this delay, *i.e.*, $\delta \leq \Delta$, which is also *finite*. In the *synchronous* model, $\Delta$ is *known* to all parties. In the *partially synchronous* model [11], $\Delta$ is *unknown* but still finite, *i.e.*, all messages are eventually delivered. A protocol is called *responsive* if it does not rely on knowledge of $\Delta$ and its liveness guarantees depend only on the actual network delay $\delta$.

### Digital signatures

We assume that replicas (and auctioneers in bidset-core) authenticate their messages with digital signatures. A digital signature scheme consists of the following three algorithms, satisfying the EUF-CMA security [17]: (1) $KeyGen(1^\kappa)$: The key generation algorithm takes as input a security parameter $\kappa$ and outputs a secret key $\mathsf{sk}$ and a public key $\mathsf{pk}$. (2) $Sign(\mathsf{sk}, m) \to \sigma$: The signing algorithm takes as input a private key $\mathsf{sk}$ and a message

$m \in \{0,1\}^*$ and returns a signature $\sigma$. (3) $\mathit{Verify}(\mathsf{pk}, m, \sigma) \to b \in \{0,1\}$: The verification algorithm takes as input a public key $\mathsf{pk}$, a message $m$, and a signature $\sigma$, and outputs a bit $b \in \{0,1\}$. We say $\sigma$ is a *valid* signature on $m$ with respect to $\mathsf{pk}$ if $\mathit{Verify}(pk, m, \sigma) = 1$.

### Accountable safety

Taking a similar approach as Neu, Tas, and Tse [20, Def. 4], we define *accountable safety* through an *identification function*.

▶ **Definition 1** (Transcript and partial transcript). *We define as* transcript *the set of all network messages sent by all parties in an execution of a protocol. A* partial transcript *is a subset of a transcript.*

▶ **Definition 2** ($\beta$-Accountable safety). *A protocol satisfies* accountable safety *with* resilience $\beta$ *if its interface contains a function* $\mathsf{identify}(T) \to \tilde{R}$*, which takes as input a partial transcript* $T$ *and outputs a set of replicas* $\tilde{R} \subset R$*, such that the following conditions hold except with negligible probability.*

**Correctness:** *If safety is violated, then there exists a partial transcript* $T$*, such that* $\mathsf{identify}(T) \to \tilde{R}$ *and* $|\tilde{R}| > \beta$*.*

**No-framing:** *For any partial transcript* $T$ *produced during an execution of the protocol, the output of* $\mathsf{identify}(T)$ *does not contain honest replicas.*

▶ Remark 3. For simplicity, we have defined the transcript based on messages sent by *all* replicas. We can also define a *local transcript* as the set of messages observed by a single party. As will become evident from the implementation of $\mathsf{identify}()$, in practice, adversarial behavior can be identified from the local transcripts of a single party or of a pair of parties.

## 3 Modeling pod

In this section, we introduce the notion of a $\mathsf{pod}$, a distributed protocol where clients can *read* and *write* transactions. We first define the basic data structures of a $\mathsf{pod}$ protocol.

▶ **Definition 4** (Transaction trace and trace set). *The* transaction trace *of a transaction* $\mathsf{tx} \in \{0,1\}^*$ *is a tuple containing the values* $(\mathsf{tx}, r_{min}, r_{max}, r_{conf})$*, which change during the execution of a* $\mathsf{pod}$ *protocol. We call* $r_{min} \in \mathbb{N}$ *the* minimum round*,* $r_{max} \in \mathbb{N} \cup \{\infty\}$ *the* maximum round*,* $r_{conf} \in \mathbb{N} \cup \{\perp\}$ *the* confirmed round*. We denote by* $r_{max} = \infty$ *an unbounded maximum round and by* $r_{conf} = \perp$ *an undefined confirmed round. We also denote these values as* $\mathsf{tx}.r_{min}, \mathsf{tx}.r_{max}$*, and* $\mathsf{tx}.r_{conf}$*. A* trace set $\mathsf{T}$ *is a set of transaction traces* $\{(\mathsf{tx}, r_{min}, r_{max}, r_{conf}) \mid \mathsf{tx} \in \{0,1\}^*\}$.

▶ **Definition 5** (Confirmed transaction). *A transaction with confirmed round* $r_{conf}$ *is called* confirmed *if* $r_{conf} \neq \perp$*, and* unconfirmed *otherwise.*

▶ **Definition 6** (Pod data structure). *A* pod data structure $D$ *is a tuple* $(\mathsf{T}, r_{perf})$*, where* $\mathsf{T}$ *is a trace set and* $r_{perf}$ *is a round number called the* past-perfect round.

We denote the components of a pod data structure as $D.\mathsf{T}$ and $D.\mathsf{r}_{\mathrm{perf}}$. We write $\mathsf{tx} \in D.\mathsf{T}$ if an entry $(\mathsf{tx}, \cdot, \cdot, \cdot)$ exists in $D.\mathsf{T}$. We remark that transactions in $\mathsf{T}$ may be confirmed on unconfirmed. Moreover, $\mathsf{r}_{\mathrm{perf}}$ will be used to define a completeness property on $\mathsf{T}$ (the *past-perfection* property of $\mathsf{pod}$).

▶ **Definition 7** (Auxiliary data). *We associate with a pod data structure $D$ some auxiliary data $C$, which will be used to validate $D$. The exact implementation of $C$ is irrelevant for the definition of pod; however, it is helpful to mention that in pod-core it will be a tuple $C = (C_{pp}, \mathbb{C}_{tx})$, where $C_{pp}$ will be a* past-perfection certificate *and $\mathbb{C}_{tx}$ a map from each transaction $\mathsf{tx}$ in $D.\mathsf{T}$ to a* transaction certificate *$C_{tx}$ for $\mathsf{tx}$. Both contain digital signatures.*

▶ **Definition 8** (Interface of a pod). *A pod protocol has the following interface.*

- *write($\mathsf{tx}$): It writes a transaction $\mathsf{tx}$ to the pod.*
- *read() $\to (D, C)$: It outputs a pod data structure $D = (\mathsf{T}, r_{perf})$ and auxiliary data $C$.*

We say that a client *reads the* pod when it calls *read*(). If $\mathsf{tx}$ appears in $\mathsf{T}$, we say that the client *observes* $\mathsf{tx}$ and, if $\mathsf{tx}.\mathsf{r}_{\text{conf}} \neq \perp$, we say that the client *observes $\mathsf{tx}$ as confirmed.*

▶ **Definition 9** (Validity function). *Apart from its interface functions, a pod protocol also specifies a computable, deterministic, and non-interactive function valid($D$, $C$) that takes as input a pod data structure $D$ and auxiliary data $C$ and outputs a boolean value. We say that a pod data structure $D$ is* valid *if valid$(D, C) = \mathsf{true}$.*

▶ **Definition 10** (View of the pod). *We call* view of the *pod and denote by $D^{\mathsf{c}}_{\mathsf{r}}$ the data structure returned by read(), where read() is invoked by client $\mathsf{c}$ and the output is produced at round $\mathsf{r}$. We remark that $\mathsf{r}$ denotes the round when read() outputs, as the client may have invoked it at an earlier round.*

We now introduce the basic definition of a *secure* pod protocol, as well as some additional properties (*timeliness* and *monotonicity*) that it may satisfy.

▶ **Definition 11** (Secure pod). *A protocol is a* secure *pod if it implements the pod interface of Definition 8 and specifies a validity function valid(), such that the following properties hold.*

**(Liveness) Completeness:** *Honest clients always output a valid pod data structure. That is, if read() returns $(D, C)$ to an honest client, then valid$(D, C) = \mathsf{true}$.*

**(Liveness) Confirmation within $u$:** *Transactions of honest clients become confirmed after at most $u$ rounds. Formally, if an honest client $\mathsf{c}$ writes a transaction $\mathsf{tx}$ at round $\mathsf{r}$, then for any honest client $\mathsf{c}'$ (including $\mathsf{c} = \mathsf{c}'$) it holds that $\mathsf{tx} \in D^{\mathsf{c}'}_{r+u}$ and $\mathsf{tx}.r_{conf} \neq \perp$.*

**(Liveness) Past-perfection within $w$:** *Rounds become past-perfect after at most $w$ rounds. Formally, for any honest client $\mathsf{c}$ and round $\mathsf{r} \geq w$, it holds that $D^{\mathsf{c}}_{\mathsf{r}}.r_{perf} \geq \mathsf{r} - w$.*

**(Safety) Past-perfection:** *A valid pod $D$ contains all transactions that may ever obtain a confirmed round smaller than $D.r_{perf}$. Formally, the adversary cannot output $(D_1, C_1)$ and $(D_2, C_2)$ to the network, such that valid$(D_1, C_1) \wedge$ valid$(D_2, C_2)$ and there exists a transaction $\mathsf{tx}$ such that $(\mathsf{tx}, r^1_{min}, r^1_{max}, r^1_{conf}) \notin D_1.\mathsf{T}$ and $(\mathsf{tx}, r^2_{min}, r^2_{max}, r^2_{conf}) \in D_2.\mathsf{T}$ and $r^2_{conf} \neq \perp$ and $r^2_{conf} < D_1.r_{perf}$.*

**(Safety) Confirmation bounds:** *The values $r_{min}$ and $r_{max}$ bound the confirmed round that a transaction may ever obtain. Formally, the adversary cannot output $(D_1, C_1)$ and $(D_2, C_2)$ to the network, such that valid$(D_1, C_1) \wedge$ valid$(D_2, C_2)$ and there exists a transaction $\mathsf{tx}$ such that $(\mathsf{tx}, r^1_{min}, r^1_{max}, r^1_{conf}) \in D_1.\mathsf{T}$ and $(\mathsf{tx}, r^2_{min}, r^2_{max}, r^2_{conf}) \in D_2.\mathsf{T}$ and $r^1_{min} > r^2_{conf}$ or $r^1_{max} < r^2_{conf}$.*

The *confirmation bounds* property gives $\mathsf{r}^1_{\text{min}} \leq \mathsf{r}^2_{\text{conf}} \leq \mathsf{r}^1_{\text{max}}$, for $\mathsf{r}^1_{\text{min}}, \mathsf{r}^1_{\text{max}}, \mathsf{r}^2_{\text{conf}}$ computed by honest clients, but it does not guarantee anything about the values of $\mathsf{r}^1_{\text{min}}$ and $\mathsf{r}^1_{\text{max}}$ (for example, it could trivially be $\mathsf{r}^1_{\text{min}} = 0$ and $\mathsf{r}^1_{\text{max}} = \infty$). To this purpose we define an additional property of pod, called *timeliness*. Previous work has observed a similar property as orthogonal to safety and liveness [25].

▶ **Definition 12** (pod $\theta$-timeliness for honest transactions). *A pod protocol is* $\theta$-timely *if it is a secure pod, as per Definition 11, and for any honest clients* $c_1, c_2$*, if* $c_1$ *writes transaction* $tx$ *in round* $r$ *and* $c_2$ *has view* $D_{r'}^{c_2}$ *in round* $r'$*, such that* $(tx, r_{min}, r_{max}, r_{conf}) \in D_{r'}^{c_2}.T$*, then: (1)* $r_{conf} \in (r, r+\theta]$*; (2)* $r_{max} \in (r, r+\theta]$*; (3)* $r_{max} - r_{min} < \theta$*, implying* $r_{min} \neq 0$ *and* $r_{max} \neq \infty$.

Moreover, a pod protocol allows $\mathsf{r}_{\min}$, $\mathsf{r}_{\max}$, $\mathsf{r}_{\text{conf}}$ to change during an execution – for example, clients in construction pod-core will update them when they receive votes from replicas. In the full version of this paper [2, Appendix A] we define *monotonicity* properties that impose restrictions on how these values evolve.

We conclude this section with some visual examples in Figures 2 and 3.

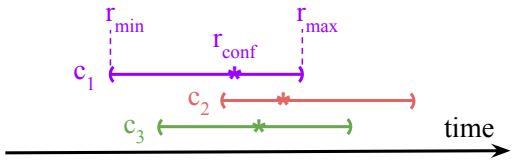


**Figure 2** The same transaction in the view of three different pod clients. Each client assigns it a minimum round $\mathsf{r}_{\min}$ and a maximum round $\mathsf{r}_{\max}$. If it gets confirmed, the confirmation round $\mathsf{r}_{\text{conf}}$ will be between these two values. The $\mathsf{r}_{\text{conf}}$ that each client locally computes respects the bounds of each other client.


**Figure 3** A possible view of a single pod client. Transactions $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3$ are confirmed, $\mathsf{tx}_4$ is not yet confirmed. A client also derives a past-perfect round $\mathsf{r}_{\text{perf}}$. No transaction other than $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_4$ may obtain $\mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\text{perf}}$. There may exist $\mathsf{tx}_5$ for which the client has not received votes, but $\mathsf{tx}_5$ cannot obtain $\mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\text{perf}}$.

## 4 Protocol pod-core

Before we present protocol pod-core, we define basic concepts and structures.

▶ **Definition 13** (Vote). *A* vote *is a tuple* $\mathsf{vote} = (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R)$*, where* $\mathsf{tx}$ *is a transaction,* $\mathsf{ts}$ *is a timestamp,* $\mathsf{sn}$ *is a sequence number,* $\sigma$ *is a signature, and* $R$ *is a replica. A vote is* valid *if* $\sigma$ *is a valid signature on message* $m = (\mathsf{tx}, \mathsf{ts}, \mathsf{sn})$ *with respect to the public key* $\mathsf{pk}_R$ *of replica* $R$.

▶ Remark 14 (Sequence numbers, session identifiers, streaming algorithm). Honest clients process votes from each replica in the same order, namely in order of increasing timestamps. For this, a replica maintains a *sequence number* which it increments and includes every time it assigns a timestamp to a transaction. We also assume that all messages between clients and replicas are concatenated with a *session identifier (sid)*, which is unique for each concurrent execution of the protocol and included in all messages signed by the replicas. Finally, the client protocol we show in Protocol 1 is *streaming*, that is, clients maintain a connection to the replicas, and *stateful*, that is, they persist their state (received transactions and votes) across all invocations of *write*() and *read*().

**Past-perfection and transaction certificates.** In pod-core, clients store certain votes which they output upon *read*() as part of the *certificate* $C$, which will be used to prove the validity of the returned $D$ and for accountability in case of safety violations. Specifically, $C$ consists of two parts, $C = (C_{\text{pp}}, \mathbb{C}_{\text{tx}})$: the *past-perfection certificate* $C_{\text{pp}}$ contains, for each replica, the vote on the most recent timestamp received from that replica. It is implemented as a map from replicas to votes, i.e., $C_{\text{pp}} : R \to \mathsf{vote}$. The *transaction certificate* $\mathbb{C}_{\text{tx}}$ contains, for

each transaction, all valid votes received for it. It is implemented as a map from transactions to a map from replicas to votes, i.e., $\mathbb{C}_{\mathsf{tx}} : \mathsf{tx} \to C_{\mathsf{tx}}$ and $C_{\mathsf{tx}} : R \to \mathsf{vote}$. We remark that $C_{\mathrm{pp}}$ can be derived by taking the union of certificates $C_{\mathsf{tx}}$ for all transactions and keeping the most recent vote for each replica, but we define $C_{\mathrm{pp}}$ explicitly for clarity and readability.

▶ **Protocol 1** (pod-core). *Protocol pod-core is executed by $n$ replicas that follow the steps of Algorithm 1 and an unknown number of clients that follow the steps of Algorithms 2 and 3 with parameters $\beta$, $\gamma$ and $\alpha$, where $\beta$ denotes the number of Byzantine replicas and $\gamma$ the number of omission-faulty replicas (in addition to the Byzantine) and $\alpha = n - \beta - \gamma$ is the number of honest replicas.*

## 4.1 Replica code

**Algorithm 1** Protocol pod-core: Code for a replica $R_i$, where sk denotes its secret signing key.

```
1:  C                                              ▷ The set of all connected clients
2:  nextsn                                         ▷ The next sequence number to assign to votes
3:  replicaLog                                     ▷ The transaction log or the replica

4:  upon init() do                                 ▷ Called once when the replica is initialized
5:      C ← ∅; nextsn ← 0; replicaLog ← [ ]
6:  end upon

7:  upon ⟨CONNECT ⟩ ← c do                         ▷ Called when a new client c connects to the replica
8:      C ← C ∪ {c}
9:      for (tx, ts, sn, σ) ∈ replicaLog do
10:         ⟨VOTE (tx, ts, sn, σ, R_i)⟩ → c
11:     end for
12: end upon

13: upon ⟨WRITE tx⟩ ← c do                         ▷ Called when a client c writes a transaction tx
14:     if replicaLog[tx] ≠ ⊥ then return          ▷ Ignore duplicate transactions
15:     doVote(tx)
16: end upon

17: function doVote(tx)
18:     ts ← round(); sn ← nextsn; σ ← Sign(sk, (tx, ts, sn))
19:     replicaLog ← replicaLog ‖ (tx, ts, sn, σ)
20:     for c ∈ C do ⟨VOTE (tx, ts, sn, σ, R_i)⟩ → c
21:     nextsn ← nextsn + 1
22: end function

23: upon end round do                              ▷ Executed at the end of each round
24:     tx ← HEARTBEAT‖round()
25:     doVote(tx)
26: end upon
```

We show the pseudocode for the replica code in Algorithm 1. The state of a replica (lines 1–3) contains replicaLog, a log implemented as a sequence of votes $(\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_i)$ created by the replica, where ts is the timestamp assigned by the replica to tx, sn is a sequence number, and $\sigma$ its signature. When the replica receives $\langle CONNECT \rangle$ from a client c, it appends c to its set of connected clients and sends to c all entries in replicaLog (lines 7–12).

When it receives $\langle \mathit{WRITE}\ \mathsf{tx} \rangle$, a replica first checks whether it has already seen $\mathsf{tx}$, in which case the message is ignored. Otherwise, it assigns $\mathsf{tx}$ a timestamp $\mathsf{ts}$ equal its local round number and the next available sequence number $\mathsf{sn}$, and signs the message $(\mathsf{tx}, \mathsf{ts}, \mathsf{sn})$ (line 18). Honest replicas use incremental sequence numbers for each transaction, implying that a vote with a larger sequence number than a second vote will have a larger or equal timestamp than the second. The replica appends $(\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma)$ to $\mathsf{replicaLog}$, and sends it via a $\langle \mathit{VOTE}\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_i) \rangle$ message to all connected clients (line 20).

**Heartbeat messages.** Clients maintain a *most-recent timestamp* variable $\mathsf{mrt}[R_j]$ for each replica. This is updated every time they receive a vote and is crucial for computing the past-perfect round $\mathsf{r}_{\text{perf}}$. To make sure that clients update $\mathsf{mrt}[R_j]$ even when $R_j$ does not have any new transactions in a round, we have replicas send a vote on a dummy HEARTBEAT transaction the end of each round (lines 23–26). An obvious practical optimization is to send HEARTBEAT only for rounds when no other transactions were sent. When received by a client, a HEARTBEAT is handled as a vote (i.e., it triggers line 11). To avoid being considered a duplicate vote by clients (see line 33 in Algorithm 2), replicas append the round number to the HEARTBEAT transaction.

## 4.2 Client code

The state of a client is shown in Algorithm 2 in lines 1–6. Variable $\mathsf{tsps}$ is a map from transactions $\mathsf{tx}$ to a map from replicas $R$ to timestamps $\mathsf{ts}$. The state gets initialized in lines 7–10. At initialization the client also sends a $\langle \mathit{CONNECT} \rangle$ message to each replica, which initiates a streaming connection from the replica to the client.

**Receiving votes.** A client maintains a connection to each replica and receives votes through $\langle \mathit{VOTE}\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j) \rangle$ messages (lines 11–16). When a vote is received from replica $R_j$, the client first verifies the signature $\sigma$ under $R_j$'s public key (line 30). If invalid, the vote is ignored. Then the client verifies that the vote contains the next sequence number it expects to receive from replica $R_j$ (line 31). If this is not the case, the vote is *backlogged* and given again to the client at a later point (the backlogging functionality is not shown in the pseudocode). The client then checks the vote against previous votes received from $R_j$. First, $\mathsf{ts}$ must be greater or equal to $\mathsf{mrt}[R_j]$, the most recent timestamp returned by replica $R_j$. Second, the replica must have not previously sent a different timestamp for $\mathsf{tx}$. If both checks pass, the client updates $\mathsf{mrt}[R_j]$ and $\mathsf{tsps}[\mathsf{tx}][R_j]$ with $\mathsf{ts}$ (line 34). The client also updates $C_{\text{pp}}$ and $\mathbb{C}_{\text{tx}}$ for each valid vote (lines 13–14).

If any of these checks fail, the client ignores the vote, since both of these cases constitute *accountable* faults: In the first case, the client can use the message $\langle \mathit{VOTE}\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j) \rangle$ and the vote it received when it updated $\mathsf{mrt}[R_j]$ to prove that $R_j$ has misbehaved. In the second case, it can use $\langle \mathit{VOTE}\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j) \rangle$ and the previous vote it has received for $\mathsf{tx}$. The $\mathsf{identify}()$ function we show in Algorithm 8 can detect such misbehavior

**Writing to and reading from pod.** Clients interact with a $\mathsf{pod}$ using the *write*($\mathsf{tx}$) and *read*() functions. In order to write a transaction $\mathsf{tx}$, a client sends $\langle \mathit{WRITE}\ \mathsf{tx} \rangle$ to each replica (lines 17–19). Since the construction is stateful and streaming, the client state contains at all times the latest view the client has of the $\mathsf{pod}$. Hence, *read*() operates on the local state (lines 20–24). It returns all the transactions the client has received so far and their traces, and the current past-perfect round $\mathsf{r}_{\text{perf}}$. We will show the details of *computeTxSet*() in Algorithm 3. As per the pod interface, *read*() also returns auxiliary data $C$, which has two parts: the past-perfection certificate $C_{\text{pp}}$ and a list of transaction certificates $C_{\text{tx}}$ (line 23). Note that $\mathsf{tsps}$.*keys()* on Algorithm 3 returns all entries in $\mathsf{tsps}$.

**Algorithm 2** Protocol pod-core: Code for a client, part 1.

1: $\mathcal{R} = \{R_1, \ldots, R_n\}$; $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ ▷ All replicas and their public keys
2: $\mathsf{mrt} : R \to \mathsf{ts}$ ▷ The most recent timestamp returned by each replica
3: $\mathsf{nextsn} : R \to \mathsf{sn}$ ▷ The next sequence number expected by each replica
4: $\mathsf{tsps} : \mathsf{tx} \to (R \to \mathsf{ts})$ ▷ Timestamp received for each tx from each replica
5: $C_{\text{pp}} : R \to \mathsf{vote}$ ▷ Past-perfection certificate: most recent vote from each replica
6: $\mathbb{C}_{\text{tx}} : \mathsf{tx} \to C_{\text{tx}}$, where $C_{\text{tx}} : R \to \mathsf{vote}$ ▷ Transaction certificates

7: **upon** *init*() **do** ▷ Called once when the client is initialized
8: *initState*()
9: **for** $R_j \in \mathcal{R}$ **do:** $\langle CONNECT \rangle \to R_j$
10: **end upon**

11: **upon** $\langle VOTE\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)\rangle \leftarrow R_j$ **do** ▷ Called when client receives a vote
12: **if** *processVote*$(\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)$ **then**
13: $C_{\text{pp}}[R_j] \leftarrow (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)$ ▷ Keep most recent vote from $R_j$ in $C_{\text{pp}}$
14: $\mathbb{C}_{\text{tx}}[\mathsf{tx}][R_j] \leftarrow (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)$ ▷ Keep all votes for tx in $C_{\text{tx}}$
15: **end if**
16: **end upon**

17: **function** *write*(tx) ▷ Part of pod interface, used to write a new transaction
18: **for** $R_j \in \mathcal{R}$ **do:** $\langle WRITE\ \mathsf{tx}\rangle \to R_j$
19: **end function**

20: **function** *read*() ▷ Part of pod interface, used to read all transactions
21: $\mathsf{T} \leftarrow$ *computeTxSet*$(\mathsf{tsps}, \mathsf{mrt})$ ▷ Shown in Algorithm 3
22: $\mathsf{r}_{\text{perf}} \leftarrow$ *computePastPerfectRound*$(\mathsf{mrt})$ ▷ Shown in Algorithm 3
23: $D \leftarrow (\mathsf{T}, \mathsf{r}_{\text{perf}})$ ; $C \leftarrow (C_{\text{pp}}, \mathbb{C}_{\text{tx}})$ ; **return**$(D, C)$
24: **end function**

25: **function** *initState*()
26: $\mathsf{tsps} \leftarrow \emptyset$; $\mathbb{C}_{\text{tx}} \leftarrow \emptyset$
27: **for** $R_j \in \mathcal{R}$ **do:** $\mathsf{mrt}[R_j] \leftarrow 0$ ; $C_{\text{pp}}[R_j] \leftarrow \perp$ ; $\mathsf{nextsn}[R_j] = -1$
28: **end function**

29: **function** *processVote*$(\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)$ ▷ Validate vote and update local state
30: **require** *Verify*$(\mathsf{pk}_j, (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}), \sigma)$ ▷ Otherwise, vote is invalid
31: **require** $\mathsf{sn} = \mathsf{nextsn}[R_j]$ ▷ Otherwise, vote cannot be processed yet
32: **require** $\mathsf{ts} \geq \mathsf{mrt}[R_j]$ ▷ Otherwise, $R_j$ has sent old timestamp
33: **require** $\mathsf{tsps}[\mathsf{tx}][R_j] = \perp$ **or** $\mathsf{tsps}[\mathsf{tx}][R_j] = \mathsf{ts}$ ▷ Otherwise, vote is duplicate
34: $\mathsf{nextsn}[R_j] \leftarrow \mathsf{nextsn}[R_j] + 1$ ; $\mathsf{mrt}[R_j] \leftarrow \mathsf{ts}$ ; $\mathsf{tsps}[\mathsf{tx}][R_j] \leftarrow \mathsf{ts}$
35: **end function**

**Computing the trace values and the past-perfect round.** Function *computeTxSet*() (Algorithm 3), computes the current transaction set from the timestamps received so far. A transaction becomes confirmed when the client receives $\alpha$ votes for it, in which case $\mathsf{r}_{\text{conf}}$ is the median of all received timestamps (lines 6–8). The computation of $\mathsf{r}_{\text{min}}, \mathsf{r}_{\text{max}}$, and $\mathsf{r}_{\text{perf}}$ is done using the functions *minPossibleTs*(), *maxPossibleTs*(), and *computePastPerfectRound*(), respectively. Function *minPossibleTs*() gets the timestamps timestamps from each replica on tx and the most recent timestamps mrt. It fills a missing timestamp from replica $R_j$ with

**Algorithm 3** Protocol pod-core: Client code, part 2. Functions to compute trace values and past-perfect round. The code is parametrized with $\beta$, the number of Byzantine replicas expected by the client, and $\gamma$, the number of omission-faulty replicas, and $\alpha = n - \beta - \gamma$ for $n$ replicas.

```
1:  function computeTxSet(tsps, mrt)
2:      T ← ∅
3:      for tx ∈ tsps.keys() do                          ▷ loop over all received transactions
4:          r_min ← minPossibleTs(tsps[tx], mrt) ; r_max ← maxPossibleTs(tsps[tx])
5:          r_conf ← ⊥ ; timestamps ← [ ]
6:          if |tsps[tx].keys()| ≥ α then
7:              for R_j ∈ tsps[tx].keys() do: timestamps ← timestamps ‖ tsps[tx][R_j]
8:              r_conf ← median(timestamps)
9:          end if
10:         T ← T ∪ {(tx, r_min, r_max, r_conf)}
11:     end for
12:     return T
13: end function

14: function minPossibleTs(timestamps, mrt)
15:     for R_j ∈ R do
16:         if timestamps[R_j] = ⊥ then timestamps ← timestamps ‖ [mrt[R_j]]
17:     end for
18:     sort timestamps in increasing order of timestamps
19:     timestamps ← [0, β times ..., 0] ‖ timestamps          ▷ omitted altogether if β = 0
20:     return median(timestamps[: α])
21: end function

22: function maxPossibleTs(timestamps)
23:     for R_j ∈ R do
24:         if timestamps[R_j] = ⊥ then timestamps ← timestamps ‖ [∞]
25:     end for
26:     sort timestamps in increasing order of timestamps
27:     timestamps ← timestamps ‖ [∞, β times ..., ∞]          ▷ omitted altogether if β = 0
28:     return median(timestamps[−α :])
29: end function

30: function computePastPerfectRound(mrt)
31:     sort mrt in increasing order
32:     mrt ← [0, β times ..., 0] ‖ mrt                        ▷ omitted altogether if β = 0
33:     return median(mrt[: α])
34: end function

35: function median(Y) : return Y[⌊ |Y|/2 ⌋]
```

$\mathsf{mrt}[R_j]$ (line 16), the minimum timestamp that can ever be accepted from $R_j$ (see the check in line 32 of Algorithm 2). It then prepends $\beta$ times the 0 value, pessimistically assuming that up to $\beta$ replicas will try to bias $\mathsf{tx}$ by sending a timestamp 0 to other clients. It then returns the median of the $\alpha$ smallest timestamps, which, again pessimistically, are the smallest timestamps another client may use to confirm $\mathsf{tx}$. Function *maxPossibleTs*() is analogous, filling a missing vote with $\infty$ (line 24) and appending the $\infty$ value, the worst-case timestamp

that Byzantine replicas may send to other clients. Finally, *computePastPerfectRound*() is similar to *minPossibleTs*() but it operates on timestamps mrt. As honest clients will not accept a timestamp smaller than mrt on any future transaction (line 32 of Algorithm 2), the returned value bounds from below the confirmed round of *any* transaction *not yet seen*.

## 4.3 Validation function

The validation function *valid*() allows a pod client to verify that a given pod data structure $D$ satisfies the security properties of pod (Definition 11) *without* necessarily communicating with pod replicas. The function *valid*() for pod-core is shown in Section A. The verifier repeats the logic of an honest client: it is initialized in the same way as a pod client, goes through the votes found in $\mathbb{C}_{\text{tx}}$, and checks that the resulting values match the ones in $D$.

## 4.4 Analysis

▶ **Theorem 15** (pod-core security). *Assume that the network is partially synchronous with actual network delay $\delta$, that $\beta$ is the number of Byzantine replicas, $\gamma$ the number of omission-faulty replicas, $\alpha = n - \beta - \gamma$ the confirmation threshold, and $n \geq 5\beta + 3\gamma + 1$ the total number of replicas. Protocol pod-core (Protocol 1), instantiated with a EUF-CMA secure signature scheme, the valid() function shown in Algorithm 7, and the* identify() *function described in Algorithm 8, is a responsive secure pod (Definition 11) with Confirmation within $u = 2\delta$, Past-perfection within $w = \delta$ and $\beta$-accountable safety (Definition 2), except with negligible probability.*

**Proof.** Shown in Appendix B. ◀

# 5 Evaluation

To validate our theoretical results regarding optimal latency in Protocol pod-core, we implement[1] a prototype pod-core in Rust 1.85. Our benchmarks measure the end-to-end confirmation latency of a transaction from the moment it is written by client until it is read as confirmed by another client in a different continent, both interacting with replicas distributed around the world. Specifically, the latency is computed as the difference between the timestamp recorded by the reading client upon receiving sufficiently many votes (quorum size $\alpha$) from different replicas and the initial timestamp recorded by the writing client. We present the results in Figure 4.

The implementation follows a client-server architecture where each replica maintains two TCP listening sockets: one for the reading client connection and one for the writing client connection. Upon receiving a transaction payload from a writer, the replica creates a tuple containing the payload, a sequence number, and the current local timestamp. The replica then signs this tuple using a Schnorr signature[2] on secp256k1 curve, appends it to its local log, and forwards the signed tuple to the reading client. Replicas are deployed round-robin across seven AWS regions: Frankfurt, London, N. Virginia, N. California, Canada, Mumbai, and Seoul. Each replica is deployed on a t2.medium EC2 instance (2 vCPUs, 4GB RAM) and is initialized with user data that contains the replica's unique secret signing key.

[1] Our prototype implementation is available at `https://github.com/commonprefix/pod-experiments`
[2] `https://crates.io/crates/secp256k1`

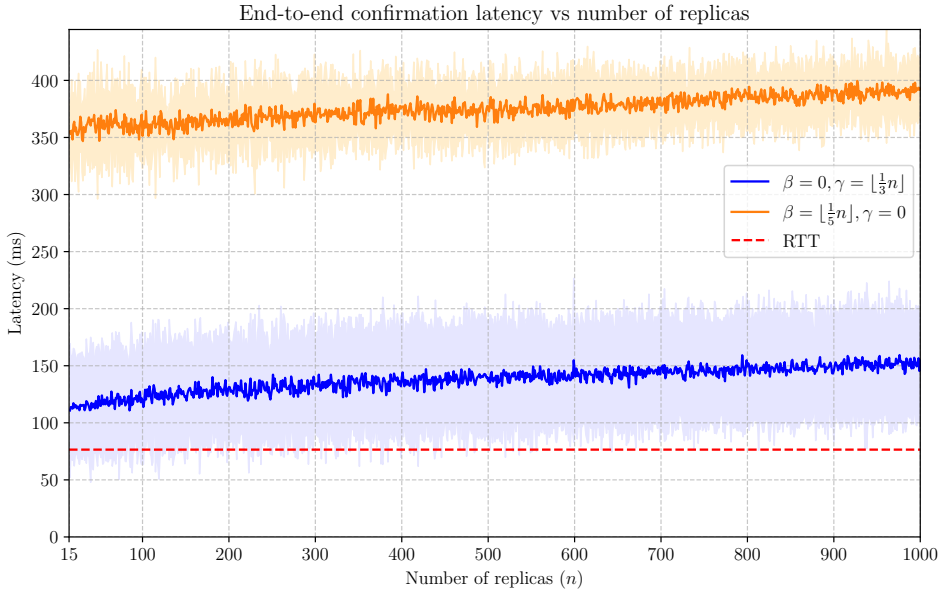


**Figure 4** End-to-end confirmation latency from a writing client to a reading client as a transaction traverses across $n = 15, \ldots, 1000$ replicas, for two reading clients: (1) a client that expects up to $\gamma = \lfloor \frac{1}{3} n \rfloor$ omission faults (blue line, below), and (2) a client that expects up to $\beta = \lfloor \frac{1}{5} n \rfloor$ Byzantine faults (orange line, above). The physical network round-trip time between the reading client and the writing client is also shown (dashed red line, 76ms). A 95% confidence interval is shown (shaded).

We implement two types of clients. The writing client establishes connections to all replicas, records the timestamp (in its local view) right before sending the transaction and sends transaction payloads to each replica. The reading client maintains connections to all replicas, validates incoming signed transactions, and records the timestamp (in its local view) upon receiving a quorum of valid signatures for a particular transaction. We deploy the reading client in London and the writing client in N. Virginia, both initialized with the complete list of replica information (IP addresses, public keys).

We conduct experiments with two different values for the quorum size $\alpha = 1 - \beta - \gamma$: (1) $\beta = 0$ and $\gamma = \lfloor \frac{1}{3} n \rfloor$, for a client that only expects omission faults, and (2) $\beta = \lfloor \frac{1}{5} n \rfloor$ and $\gamma = 0$, for a client that expects Byzantine faults. We repeat the experiments for different numbers of replicas ($n = 15, \ldots, 1000$). We repeat each experiment five times and report the mean latency and a 95% confidence interval.

As shown in Figure 4, our experimental results demonstrate that the latency remains largely independent of the number of replicas. The reading client reports a transaction as confirmed as soon as the fastest $\alpha$ replicas have responded, which gives rise to the happy artifact that the 1 - $\alpha$ slowest replicas do not slow down confirmation. This also explains why the omission-fault experiment exhibits lower latency than the Byzantine experiment. Even with 1000 replicas the mean confirmation latency is 138ms for the omission-fault experiment and 375ms for the Byzantine experiment. This approximates the physical network round-trip time between the reading client and the writing client that stands at 76ms.

## 6 Auctions on pod through the bidset protocol

In this section, we show how single-shot distributed auctions can be implemented on top of pod. This is achieved through bidset, a primitive for collecting a set of bids. The idea is as follows. A pre-appointed *sequencer* – which can be any party, even a pod replica – runs the auction, but the bids are collected from pod using a bidset protocol. The past-perfection property of pod renders the sequencer unable to censor bids: when it creates an output, all timely and honestly-written bids *must* be in it, otherwise the sequencer has provably misbehaved and can be held accountable.

▶ **Definition 16** (bidset protocol). *A bidset* protocol *has a* starting time *parameter* $t_0$ *and exposes the following interfaces to* bidder *and* consumer *parties:*

- *function submitBid(b): It is called by a bidder at round* $t_0$ *to submit a bid* $b$*.*
- *event result*$(B, C_{\text{bid}})$*: It is an event generated by a consumer. It contains a* bid-set $B$*, which is a set of bids, and auxiliary information* $C_{bid}$*.*

A bidset protocol satisfies the following liveness and safety properties:

**(Liveness) Termination within $W$:** An honest consumer generates an event *result*$(B, C_{bid})$ by round $t_0 + W$.

**(Safety) Censorship resistance:** If an honest bidder calls *submitBid*$(b)$ and an honest consumer generates an event *result*$(B, \cdot)$, then $b \in B$.

**(Safety) Weak consistency:** If two honest consumers generate *result*$(B_1, \cdot)$ and *result*$(B_2, \cdot)$ events, such that $B_1 \neq \emptyset$ and $B_2 \neq \emptyset$, then $B_1 = B_2$.

▶ **Protocol 2** (bidset-core). *Construction bidset-core is parameterized by an integer* $\Delta$ *(looking ahead, we will prove security in synchrony, i.e., assuming the network delay* $\delta$ *is smaller than* $\Delta$*) and assumes digital signatures and a pod with* $\delta$*-timeliness,* $w = \delta$ *and* $u = 2\delta$*. At time* $t_0$*, bidders execute Algorithm 4, the sequencer Algorithm 5, and the consumers Algorithm 6.*

**Algorithm 4** bidset-core: Code for a bidder. It runs a client for a pod-core instance *pod*.

```
1: function submitBid(b)
2:     pod.write(b)
3: end function
```

**Algorithm 5** bidset-core: Code for the sequencer. It runs a client for a pod-core instance *pod*, and $\mathsf{sk}_a$ denotes the secret key of the sequencer.

```
1: function readBids()
2:     ((T, r_perf), (C_pp, C_tx)) ← pod.read()
3:     while r_perf ≤ t_0 + Δ do: ((T, r_perf), (C_pp, C_tx)) ← pod.read()
4:     B ← {tx | (tx, ·, ·, ·) ∈ T}; C_bid ← C_pp
5:     σ ← Sign(sk_a, (B, C_bid))
6:     tx ← ⟨BIDS (B, C_bid, σ)⟩
7:     pod.write(tx)
8: end function
```

A bidder (Algorithm 4) submits a bid by writing it on the pod at round $t_0$. The sequencer (Algorithm 5) waits until the pod returns a past-perfect round larger than $t_0 + \Delta$ (Algorithm 5) and then constructs the bid-set $B$ from the set of transactions in $\mathsf{T}$ (Algorithm 5). The sequencer concludes by signing $B$ and $C_{\text{bid}}$ (which can be used as evidence, in case of a safety violation) and writing $\langle BIDS\ (B, C_{\text{bid}}, \sigma)\rangle$ on *pod*. The consumer (Algorithm 6) waits until

**Algorithm 6** bidset-core: Code for a consumer. It runs a client for a pod-core instance *pod*.

```
1: function readResult()
2:     loop
3:         ((T, r_perf), (C_pp, C_tx)) ← pod.read()
4:         if ∃(tx, ·, ·, r_conf, ·) ∈ T : tx = ⟨BIDS (B, C_bid, σ)⟩ and r_conf ≤ t_0 + 3Δ then
5:             output event result(B, C_bid)
6:         else if r_perf > t_0 + 3Δ then
7:             output event result(∅, C_pp)
8:         end if
9:     end loop
10: end function
```

one of the following two conditions is met. First, a *confirmed* transaction $\langle BIDS\ (B, C_{\text{bid}}, \sigma)\rangle$ appears in $\mathsf{T}$, for which $\mathsf{r}_{\text{conf}} \leq t_0 + 3\Delta$ (Algorithm 6), in which case it outputs bid-set $B$ as result. Second, a round higher than $t_0 + 3\Delta$ becomes past-perfect in pod (Algorithm 6) without a confirmed $\langle BIDS\ \rangle$ transaction appearing, in which case it outputs $B = \emptyset$.

▶ **Theorem 17** (Bidset security)**.** *Assuming a synchronous network where $\delta \leq \Delta$, protocol bidset-core (Construction 2) instantiated with a digital signature and a secure pod protocol that satisfies the* past-perfection within $w = \delta$, confirmation within $u = 2\delta$ *and* $\delta$-timeliness *properties, is a secure bidset protocol satisfying* termination within $W = 3\Delta + \delta$*. It satisfies accountable safety with an* identifySequencer() *function that identifies a malicious sequencer.*

**Proof.** Additional intuition, formal proofs, and identifySequencer() are shown in the full version of this paper [2, Appendix D]. ◀

▶ Remark 18. Observe that bidset-core terminates within $W = 3\Delta + \delta$ in the worst case, but, if the sequencer is honest, then it terminates within $W = \Delta + 3\delta$. Moreover, bidset-core is not responsive because Algorithm 5 waits for a fixed $\Delta$ interval. This step can be optimized if the set of bidders is known (e.g., requiring them to pre-register), which allows for the protocol to be made optimistically responsive (*i.e.*, $W = 4\delta$) when all parties are honest.

▶ Remark 19 (Implicit sub-session identifiers). We assume that each instance of the bidset-core protocol is identified by a unique sub-session identifier (ssid). All messages written to the underlying pod are concatenated with the ssid.

## 7 Conclusion

In this work we present pod, a novel consensus layer that finalizes transactions with the optimal one-round-trip latency by eliminating communication among replicas. Instead, clients read the system state by performing lightweight computation on logs retrieved from the replicas. As no validator has a particular role in pod (as compared to leaders, block proposers, miners, etc. in similar protocols), pod achieves censorship resistance by default, without any extra mechanisms or additional cost. Furthermore, validator misbehavior, such as voting in incompatible ways or censoring confirmed transactions, is accountable.

As an application, we present an efficient and censorship-resistant mechanism for single-shot first-price and second-price open auctions, which leverages pod as a bulletin board. We show how the accountability, offered by pod, is also inherited by applications built on it – the auctioneer cannot censor confirmed bids without being detected. In the full version [2] of this paper we discuss further applications, including payments in the style of Fastpay [5].

We conjecture that single-shot sealed bid auction protocols, such as those of [12, 4, 9, 24, 6, 13], can also be instantiated on top of a bidset protocol. Intuitively, this holds because such protocols first agree on a set of sealed bids and then determine the winner. A formal analysis of sealed-bid auction protocols based on bidset is left as future work. The architecture of pod also motivates the design of light clients, which would not have to connect to all replicas or to download all transactions. This is achieved through cryptographic primitives such as Merkle mountain ranges and segment trees. We leave the formal description as future work.

Finally, we remark that pod differs from standard notions of consensus because it does not offer an agreement property, neither to validators nor to clients. A client reading the pod obtains a past-perfect round $\mathsf{r}_{\text{perf}}$, and it is guaranteed to have received all transactions that obtained a confirmed round $\mathsf{r}_{\text{conf}}$ such that $\mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\text{perf}}$, or that may obtain such an $\mathsf{r}_{\text{conf}}$ in the future, even though the transaction presently appears to the client as unconfirmed. However, the client cannot tell which unconfirmed transactions will become confirmed. Moreover, a transaction might appear confirmed to one client and unconfirmed to another (in this case, this is a transaction written by a malicious client).

## References


1. Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that $t$-resilient consensus requires $t + 1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999. doi:10.1016/S0020-0190(99)00100-3.
2. Orestis Alpos, Bernardo David, and Dionysis Zindros. Pod: An optimal-latency, censorship-free, and accountable generalized consensus layer. *CoRR*, abs/2501.14931, 2025. doi:10.48550/arXiv.2501.14931.
3. Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency DAG consensus with fast commit path. *CoRR*, abs/2310.14821, 2023. doi:10.48550/arXiv.2310.14821.
4. Samiran Bag, Feng Hao, Siamak F. Shahandashti, and Indranil Ghosh Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15:2042–2052, 2020. doi:10.1109/TIFS.2019.2955793.
5. Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *AFT*, pages 163–177. ACM, 2020. doi:10.1145/3419614.3423249.
6. Tarun Chitra, Matheus V. X. Ferreira, and Kshitij Kulkarni. Credible, Optimal Auctions via Public Broadcast. In Rainer Böhme and Lucianna Kiffer, editors, *6th Conference on Advances in Financial Technologies (AFT 2024)*, volume 316 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.AFT.2024.19.
7. Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38. IEEE, 2020. doi:10.1109/DSN48063.2020.00023.
8. George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In *EuroSys*, pages 34–50. ACM, 2022. doi:10.1145/3492321.3519594.
9. Bernardo David, Lorenzo Gentile, and Mohsen Pourpouneh. FAST: Fair auctions via secret transactions. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 727–747. Springer, Cham, June 2022. doi:10.1007/978-3-031-09234-3_36.

10 Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader BFT via optimistic proposals. *CoRR*, abs/2401.01791, 2024. doi:10.48550/arXiv.2401.01791.

11 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

12 Hisham S. Galal and Amr M. Youssef. Trustee: Full privacy preserving vickrey auction on top of ethereum. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 190–207, Cham, 2020. Springer International Publishing.

13 Chaya Ganesh, Shreyas Gupta, Bhavana Kanukurthi, and Girisha Shankar. Secure vickrey auctions with rational parties. Cryptology ePrint Archive, Paper 2024/1011, 2024. To appear at CCS 2024. doi:10.1145/3658644.3670311.

14 Juan A. Garay, Jonathan Katz, Chiu-Yuen Koo, and Rafail Ostrovsky. Round complexity of authenticated broadcast with a dishonest majority. In *FOCS*, pages 658–668. IEEE Computer Society, 2007. doi:10.1109/FOCS.2007.44.

15 Peter Gazi, Ling Ren, and Alexander Russell. Practical settlement bounds for longest-chain consensus. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part I*, volume 14081 of *Lecture Notes in Computer Science*, pages 107–138. Springer, 2023. doi:10.1007/978-3-031-38557-5_4.

16 Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022. doi:10.1007/978-3-031-18283-9_14.

17 Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988. doi:10.1137/0217017.

18 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. *Distributed Comput.*, 35(1):1–15, 2022. doi:10.1007/S00446-021-00399-2.

19 Dahlia Malkhi and Kartik Nayak. Extended abstract: Hotstuff-2: Optimal two-phase responsive BFT. *IACR Cryptol. ePrint Arch.*, page 397, 2023. URL: https://eprint.iacr.org/2023/397.

20 Joachim Neu, Ertem Nusret Tas, and David Tse. The availability-accountability dilemma and its resolution via accountability gadgets. In Ittay Eyal and Juan A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 541–559. Springer, Cham, May 2022. doi:10.1007/978-3-031-18283-9_27.

21 Barbara Simons. An overview of clock synchronization. In Barbara Simons and Alfred Spector, editors, *Fault-Tolerant Distributed Computing*, pages 84–96, New York, NY, 1990. Springer New York.

22 Jakub Sliwinski and Roger Wattenhofer. ABC: asynchronous blockchain without consensus. *CoRR*, abs/1909.10926, 2019. arXiv:1909.10926.

23 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version. *CoRR*, abs/2209.05633, 2022. doi:10.48550/arXiv.2209.05633.

24 Nirvan Tyagi, Arasu Arun, Cody Freitag, Riad Wahby, Joseph Bonneau, and David Mazières. Riggs: Decentralized sealed-bid auctions. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 1227–1241. ACM Press, November 2023. doi:10.1145/3576915.3623182.

25 Apostolos Tzinas, Srivatsan Sridhar, and Dionysis Zindros. On-chain timestamps are accurate. Cryptology ePrint Archive, Report 2023/1648, 2023. URL: https://eprint.iacr.org/2023/1648.

26 Josef Widder. Booting clock synchronization in partially synchronous systems. In Faith Ellen Fich, editor, *Distributed Computing*, pages 121–135, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-39989-6_9.

27 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

## A Validation function for pod-core

In this section we present the function *valid*(), which allows a pod client, which is not necessarily communicating with the pod replicas, to verify that a given pod data structure $D$ satisfies the security properties of pod (Definition 11).

**Algorithm 7** Function *valid*($D$, $C$) for pod-core. Code for a *verifier*, which can be a pod client not communicating with the pod replicas.

1: **State:** Same as in Algorithm 2, includes $\{R_1, \ldots, R_n\}$, tsps, mrt.

2: **function** *valid*($D$, $C$)
3: $(C_{\text{pp}}, \mathbb{C}_{\text{tx}}) \leftarrow C$ ▷ $C_{\text{pp}} : R \to \text{vote}$, $\mathbb{C}_{\text{tx}} : \text{tx} \to C_{\text{tx}}$, $C_{\text{tx}} : R \to \text{vote}$
4: *initState*() ▷ shown in Algorithm 2
5: allVotes $\leftarrow \bigcup_{\text{tx} \in \mathbb{C}_{\text{tx}}} (\mathbb{C}_{\text{tx}}[\text{tx}].\mathit{values}())$
6: **for** $(\text{tx}, \text{ts}, \text{sn}, \sigma, R_j) \in$ allVotes **in increasing order of** sn **do**
7: **require** *processVote*($\text{tx}, \text{ts}, \text{sn}, \sigma, R_j$) ▷ shown in Algorithm 2, updates tsps, mrt
8: **end for**
9: **require** $D.\mathsf{T} = \mathit{computeTxSet}(\text{tsps}, \text{mrt})$ ▷ shown in Algorithm 3
10: **require** $D.\mathsf{r}_{\text{perf}} = \mathit{computePastPerfectRound}(\text{mrt})$ ▷ shown in Algorithm 3
11: **for** $(\text{tx}, \text{ts}, \text{sn}, \sigma, R_j) \in C_{\text{pp}}.\mathit{values}()$ **do**
12: **require** $(\text{tx}, \text{ts}, \text{sn}, \sigma, R_j) \in$ allVotes
13: **require** $\text{sn} = \mathit{max}_{\text{sn}'}((\cdot, \cdot, \text{sn}', \cdot, R_j) \in \text{allVotes})$
14: **end for**
15: **end function**

The function *valid*() for pod-core is shown in Algorithm 7. The idea is to have the verifier repeat the logic of an honest client. The verifier is initialized in the same way as in Algorithm 2 – importantly, it knows the identifiers and public keys of pod replicas. Function *valid*() takes as input a pod data structure $D$ and auxiliary data $C$, which contains two parts, a *past-perfection certificate* $C_{\text{pp}}$ and a collection of *transaction certificates* $\mathbb{C}_{\text{tx}}$, one for each transaction in $D.\mathsf{T}$. Both contain vote messages, as constructed by a pod client in lines 13 and 14 of Algorithm 2. The verifier processes each vote in order of increasing sequence number sn using function *processVote*(). If any vote is invalid, *valid*() returns false. Observe that if the votes are valid the verifier will have updated its local tsps and mrt variables with the same values as the pod client that constructed $D$. Finally, the verifier computes the transaction set $\mathsf{T}$ and the past-perfect round $\mathsf{r}_{\text{perf}}$ (using its local tsps and mrt variables) and requires that the values match the ones in $D$ (lines 9–10).

Finally, the verifier also verifies the past-perfection certificate. Given that the previous checks have passed, we require that each vote in $C_{\text{pp}}$ is contained in one of the transaction certificates in $\mathbb{C}_{\text{tx}}$ and has the maximum sequence number received from the client that sent the vote (lines 11–14). As we have remarked earlier, $C_{\text{pp}}$ can be derived from $\mathbb{C}_{\text{tx}}$ by taking the union of certificates $\mathbb{C}_{\text{tx}}$ for all transactions and keeping the most recent vote for each replica, in which case the checks on lines 11–14 can be omitted. We maintain the past-perfection certificate for readability and simplicity in the proofs.

## B Security of Protocol pod-core under a Continuum of Byzantine and Omission faults

In order to prove Theorem 15 and establish the security of Protocol pod-core shown Construction 1, we first prove some useful intermediate results. We remind that $n = \alpha + \beta + \gamma$, where $n$ denotes the total number of replicas, $\beta$ denotes the number of Byzantine replicas, $\gamma$ denotes the number of omission-faulty replicas in an execution, and $\alpha$ denotes the number of replicas required to confirm a transaction.

▶ **Lemma 20** (The values for minimum, maximum and confirmed rounds)**.** *Regarding Algorithm 3, we have the following. Consider the list of all timestamps received by a client for a particular transaction, replacing a missing vote from $R_j$ with a special value ($\mathsf{mrt}[R_j]$ for computing $r_{min}$, $\infty$ for computing $r_{max}$), to get $n$ values in total, sorted in increasing order. Assume mrt is also sorted in increasing order of timestamps.*

1. *$r_{min}$ is the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of this list.*
2. *$r_{max}$ is the timestamp at index $n - \alpha + \lfloor \alpha/2 \rfloor + \beta$ of this list.*
3. *$r_{perf}$ is the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of mrt.*

**Proof.** Functions *minPossibleTs*() and *computePastPerfectRound*() prepend $\beta$ times the 0 value in the beginning of the list and return the median of the first $\alpha$ values, hence they return the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$. Function *maxPossibleTs*() appends $\beta$ times the $\infty$ value at the end of the list and returns the median of the last $\alpha$ values of that list, that is, it ignores the first $n - \alpha + \beta$ values and returns the timestamp at index $n - \alpha + \beta + \lfloor \alpha/2 \rfloor$. ◀

▶ **Lemma 21** ($\mathsf{r}_{\mathsf{perf}}$ bounded by honest timestamp)**.** *Assuming $n \geq 5\beta + 3\gamma + 1$ (equiv., $\alpha \geq 4\beta + 2\gamma + 1$), for a valid pod $D$ with auxiliary data $C = (C_{pp}, \mathbb{C}_{tx})$, there exists some honest replica $R_j$, such that the most-recent timestamp mrt from $R_j$ included in $C_{pp}$ satisfies $\mathsf{mrt} \leq D.r_{perf}$.*

**Proof.** Since *valid*$(D, C)$ = true, the past-perfect round $D.\mathsf{r}_{\mathrm{perf}}$ is the value returned by *computePastPerfectRound*() of Algorithm 3. From Lemma 20 we have that $\mathsf{r}_{\mathrm{perf}}$ is the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of sorted mrt. The condition $\alpha \geq 4\beta + 2\gamma + 1$ implies that $\beta + \gamma \leq \lfloor \alpha/2 \rfloor - \beta$, hence the number of not honest replicas $(\beta + \gamma)$ cannot fill all positions between 0 and $\lfloor \alpha/2 \rfloor - \beta$, hence at least one of the indexes between 0 and $\lfloor \alpha/2 \rfloor - \beta$ (inclusive) will contain the timestamp created and sent by an honest replica. ◀

We now recall Theorem 15, which we prove through a series of lemmas.

▶ **Theorem 15** (pod-core security)**.** *Assume that the network is partially synchronous with actual network delay $\delta$, that $\beta$ is the number of Byzantine replicas, $\gamma$ the number of omission-faulty replicas, $\alpha = n - \beta - \gamma$ the confirmation threshold, and $n \geq 5\beta + 3\gamma + 1$ the total number of replicas. Protocol pod-core (Protocol 1), instantiated with a EUF-CMA secure signature scheme, the* valid() *function shown in Algorithm 7, and the* identify() *function described in Algorithm 8, is a responsive secure pod (Definition 11) with Confirmation within $u = 2\delta$, Past-perfection within $w = \delta$ and $\beta$-accountable safety (Definition 2), except with negligible probability.*

**Proof.** Follows from Lemmas 22–26, presented and proven in the remainder of this section. ◀

▶ **Lemma 22** (Confirmation within $u$)**.** *For the conditions stated in Theorem 15, Protocol 1 satisfies the* confirmation within $u$ *property (Definition 11) for $u = 2\delta$.*

**Proof.** Assume an honest client $\mathsf{c}$ calls *write*($\mathsf{tx}$) at round $\mathsf{r}$. It sends message $\langle \mathit{WRITE}\ \mathsf{tx}\rangle$ to all replicas at round $\mathsf{r}$ (line 18). An honest replica receives this by round $\mathsf{r}+\delta$ and sends a $\langle \mathit{VOTE}\ \rangle$ message back to all connected clients (line 20). An honest client $\mathsf{c}'$ receives the vote by round $\mathsf{r}+2\delta$. As are at least $\alpha$ honest replicas, $\mathsf{c}'$ receives at least $\alpha$ such votes, hence the condition in line 6 is satisfied and $\mathsf{c}'$ observes $\mathsf{tx}$ as confirmed. ◀

▶ **Lemma 23** (Past-perfection within $w$)**.** *For the conditions stated in Theorem 15, Protocol 1 satisfies the* past-perfection within $w$ *property (Definition 11) for* $w = \delta$.

**Proof.** Assume an honest client $\mathsf{c}$ at round $\mathsf{r}$ has view $D^c_{\mathsf{r}}$. From Lemma 21, there exists some honest replica $R_j$, such that the most-recent timestamp $\mathsf{mrt}[R_j]$ that $R_j$ has sent to $\mathsf{c}$ satisfies $D^c_{\mathsf{r}}.\mathsf{r}_{\text{perf}} \geq \mathsf{mrt}[R_j]$. The honest replica $R_j$ sends at least one heartbeat or vote message per round (line 25), which arrives within $\delta$ rounds, and an honest client updates $\mathsf{mrt}[R_j]$ when it receives the heartbeat or vote message. Hence, $\mathsf{c}$ will have $\mathsf{mrt}[R_j] \geq \mathsf{r} - \delta$. All together, $D^c_{\mathsf{r}}.\mathsf{r}_{\text{perf}} \geq \mathsf{r} - \delta$. ◀

▶ **Lemma 24** (Past-perfection safety)**.** *For the conditions stated in Theorem 15, Protocol 1 satisfies the* past-perfection *safety property (Definition 11), except with negligible probability.*

**Proof.** Assume the adversary outputs valid $(D_1, C_1)$ and $(D_2, C_2)$ that violate the property, i.e., there exists a transaction $\mathsf{tx}$ such that $(\mathsf{tx}, \mathsf{r}^1_{\text{min}}, \mathsf{r}^1_{\text{max}}, \mathsf{r}^1_{\text{conf}}) \notin D_1.\mathsf{T}$ and $(\mathsf{tx}, \mathsf{r}^2_{\text{min}}, \mathsf{r}^2_{\text{max}}, \mathsf{r}^2_{\text{conf}}) \in D_2.\mathsf{T}$ and $\mathsf{r}^2_{\text{conf}} \neq \perp$ and $\mathsf{r}^2_{\text{conf}} < D_1.\mathsf{r}_{\text{perf}}$. Let $C_1 = (C^1_{\text{pp}}, \mathbb{C}^1_{\text{tx}})$ and $C_2 = (C^2_{\text{pp}}, \mathbb{C}^2_{\text{tx}})$.

Let $\mathcal{R}_1$ be the set of replicas $R_i$ for which $C^1_{\text{pp}}$ contains a vote with timestamp $\mathsf{mrt}_i \geq D_1.\mathsf{r}_{\text{perf}}$. From Lemma 20 ($\mathsf{r}_{\text{perf}}$ is computed as the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of sorted $\mathsf{mrt}$), and since $D_1$ is valid, there exist at least $n - \lfloor a/2 \rfloor + \beta$ such replicas, hence $|\mathcal{R}_1| \geq n - \lfloor a/2 \rfloor + \beta$. For each $R_i \in \mathcal{R}_1$, the transaction certificates $\mathbb{C}^1_{\text{tx}}$ contain the whole log of $R_i$ with timestamps up to $\mathsf{mrt}_i$ (line 31 of Algorithm 2 does not allow gaps in the sequence number of the received votes). That is, for each $R_i \in \mathcal{R}_1$ the certificates $\mathbb{C}^1_{\text{tx}}$ contains votes

$$(\mathsf{tx}_{i,1}, \mathsf{ts}_{i,1}, 1, \sigma_{i,1}, R_i), (\mathsf{tx}_{i,2}, \mathsf{ts}_{i,2}, 2, \sigma_{i,2}, R_i), \ldots, (\mathsf{tx}_{i,k_i}, \mathsf{ts}_{i,k_i}, k_i, \sigma_{i,k_i}, R_i), \tag{1}$$

where $k_i$ is the smallest sequence number for which $\mathsf{ts}_{i,k_i} \geq D_1.\mathsf{r}_{\text{perf}}$, and $\mathsf{tx}_{i,j}$ are transactions.

Since $\mathsf{tx}$ is confirmed in $D_2$ and $\mathsf{r}^2_{\text{conf}} < D_1.\mathsf{r}_{\text{perf}}$, the transaction certificate $\mathbb{C}^2_{\text{tx}}[\mathsf{tx}]$ must contain votes on $\mathsf{tx}$ with timestamp $\mathsf{ts}_i$, such that $\mathsf{ts}_i < D_1.\mathsf{r}_{\text{perf}}$, from at least $\lfloor \alpha/2 \rfloor + 1$ replicas. Let $\mathcal{R}_2$ be the set of these replicas, with $|\mathcal{R}_2| \geq \lfloor \alpha/2 \rfloor + 1$. For each $R_i \in \mathcal{R}_2$, certificate $\mathbb{C}^2_{\text{tx}}[\mathsf{tx}]$ contains a vote

$$(\mathsf{tx}, \mathsf{ts}_i, \mathsf{sn}_i, \sigma_i, R_i), \tag{2}$$

such that $\mathsf{ts}_i < D_1.\mathsf{r}_{\text{perf}}$. We will show that, if at most $\beta$ replicas are Byzantine, this leads to a contradiction. Observe from the cardinality of $\mathcal{R}_1$ and $\mathcal{R}_2$ that at least $\beta + 1$ replicas must be in both sets, hence at least one honest replica must be in both sets (except if the adversary forges a signature under the public key of an honest replica, which happens with negligible probability). For that replica, the vote in (2) must be one of the votes in (1) since $\mathsf{ts}_i < D_1.\mathsf{r}_{\text{perf}}$ and $\mathsf{ts}_{i,m_i} \geq D_1.\mathsf{r}_{\text{perf}}$. Hence, one of the $\mathsf{tx}_{i,j}$ in (1) is $\mathsf{tx}$, and $\mathsf{tx}$ must appear in $D_1.\mathsf{T}$, a contradiction. ◀

▶ **Lemma 25** (Confirmation bounds)**.** *For the conditions stated in Theorem 15, Protocol 1 satisfies the* confirmation bounds *property (Definition 11), except with negligible probability.*

**Algorithm 8** The identify() function for Protocol pod-core (Protocol 1).

```
1: function identify(T)
2:     R̃ ← ∅
3:     for ⟨VOTE (tx_1, ts_1, sn_1, σ_1, R_1)⟩ ∈ T do
4:         if not Verify(pk_1, (tx_1, ts_1, sn_1), σ_1) then continue
5:         for ⟨VOTE (tx_2, ts_2, sn_2, σ_2, R_2)⟩ ∈ T do
6:             if not Verify(pk_2, (tx_2, ts_2, sn_2), σ_2) then continue
7:             if R_1 = R_2 and sn_1 = sn_2 and (tx_1 ≠ tx_2 or ts_1 ≠ ts_2) then
8:                 R̃ ← R̃ ∪ {R_1}
9:             end if
10:        end for
11:    end for
12: end function
```

**Proof.** Assume the adversary outputs $(D_1, C_1)$ and $(D_2, C_2)$, such that $\mathit{valid}(D_1, C_1) \wedge \mathit{valid}(D_2, C_2)$ and there exists a transaction $\mathsf{tx}$ such that $(\mathsf{tx}, \mathsf{r}^1_{\min}, \mathsf{r}^1_{\max}, \mathsf{r}^1_{\text{conf}}) \in D_1.\mathsf{T}$ and $(\mathsf{tx}, \mathsf{r}^2_{\min}, \mathsf{r}^2_{\max}, \mathsf{r}^2_{\text{conf}}) \in D_2.\mathsf{T}$. Let $C_1 = (C^1_{\text{pp}}, \mathbb{C}^1_{\text{tx}})$ and $C_2 = (C^2_{\text{pp}}, \mathbb{C}^2_{\text{tx}})$, and $C^1_{\text{tx}} = \mathbb{C}^1_{\text{tx}}[\mathsf{tx}]$ and $C^2_{\text{tx}} = \mathbb{C}^2_{\text{tx}}[\mathsf{tx}]$.

First assume $\mathsf{r}^1_{\min} > \mathsf{r}^2_{\text{conf}}$. From Lemma 20, $C^1_{\text{tx}}$ can include at most $\lfloor \alpha/2 \rfloor - \beta$ votes with a timestamp for $\mathsf{tx}$ smaller than $\mathsf{r}^1_{\min}$. Allowing up to $\beta$ replicas to equivocate, the adversary can obtain at most $\lfloor \alpha/2 \rfloor$ votes on $\mathsf{tx}$ with a timestamp smaller than $\mathsf{r}^1_{\min}$, except if it forges a digital signature from an honest replica, which happens with negligible probability. In order to compute $\mathsf{r}^2_{\text{conf}} < \mathsf{r}^1_{\min}$ for $\mathsf{tx}$, the adversary must include in $C^2_{\text{tx}}$ timestamps smaller than $\mathsf{r}^1_{\min}$ from at least $\lfloor \alpha/2 \rfloor + 1$ replicas.

Now assume $\mathsf{r}^1_{\max} < \mathsf{r}^2_{\text{conf}}$. Using Lemma 20, $C^1_{\text{tx}}$ can include at most $\alpha - \lfloor \alpha/2 \rfloor - \beta - 1$ votes with a timestamp larger than $\mathsf{r}_{\max}$, hence the number of honest replicas, from which a vote with timestamp larger than $\mathsf{r}_{\max}$ can be included in $C^2_{\text{tx}}$ is at most $\alpha - \lfloor \alpha/2 \rfloor - 1$ (since $\beta$ are malicious). If $\alpha$ is odd, this upper bound becomes $\alpha - \lfloor \alpha/2 \rfloor - 1 = \lfloor \alpha/2 \rfloor$, while at least $\lfloor \alpha/2 \rfloor + 1$ votes larger that $\mathsf{r}_{\max}$ are required to compute a median larger than $\mathsf{r}_{\max}$, and if $\alpha$ is even, then $\alpha - \lfloor \alpha/2 \rfloor - 1 = \lfloor \alpha/2 \rfloor - 1$, while at least $\lfloor \alpha/2 \rfloor$ votes larger that $\mathsf{r}_{\max}$ are required to compute a median larger than $\mathsf{r}_{\max}$. (We remind that Algorithm 3 returns as median the value at position $\lfloor \alpha/2 \rfloor$). In either case, we get a contradiction, except for the negligible probability that the adversary forges a digital signature from an honest replica. ◀

▶ **Lemma 26** ($\beta$-Accountable safety)**.** *For the conditions stated in Theorem 15, Protocol 1 satisfies accountable safety (Definition 2) with resilience $\beta$, except with negligible probability.*

**Proof.** We show that identify() (Algorithm 8) satisfies the *correctness* and *no-framing* properties required by Definition 2, in three steps.

**1.** If the past-perfection safety property (Definition 11) is violated, there exists a partial transcript $T$, such that identify() on input $T$ returns at least $\beta$ replicas.
*Proof:* We resume the proof of Lemma 24. There, we constructed sets $\mathcal{R}_1, \mathcal{R}_2$, such that $\mathcal{R}_1 \cap \mathcal{R}_2 \geq \beta + 1$. We saw that, for each $R_i \in \mathcal{R}_1 \cap \mathcal{R}_2$, certificates $\mathbb{C}^1_{\text{tx}}$ contain the replica log shown in (1), containing all votes with timestamp up to $\mathsf{ts}_{i,k_i} \geq \mathsf{r}_{\text{perf}}$. In a similar logic, certificates $\mathbb{C}^2_{\text{tx}}$ contains the following $k'_i$ votes from $R_i$ (possibly more, but we care for the votes up to transaction $\mathsf{tx}$)

$$(\mathsf{tx}'_{i,1}, \mathsf{ts}'_{i,1}, 1, \sigma'_{i,1}, R_i), (\mathsf{tx}'_{i,2}, \mathsf{ts}'_{i,2}, 2, \sigma'_{i,2}, R_i), \ldots, (\mathsf{tx}'_{i,k'_i}, \mathsf{ts}'_{i,k'_i}, k'_i, \sigma'_{i,k'_i}, R_i), \tag{3}$$

with $\mathsf{tx}'_{i,k'_i} = \mathsf{tx}$ and $\mathsf{ts}'_{i,k'_i} < \mathsf{r}_{\text{perf}}$. Obviously, for an honest $R_i$, the replica logs of (1) and (3) must be identical, i.e., $\mathsf{tx}_{i,j} = \mathsf{tx}'_{i,j}$ and $\mathsf{ts}_{i,j} = \mathsf{ts}'_{i,j}$, for $j \in [1, \min(k_i, k'_i)]$. We will show that they differ in at least one sequence number. If $k_i > k'_i$, then the replica logs differ at sequence number $k'_i$, because the transaction $\mathsf{tx}_{i,k_i}$ in (1) cannot be $\mathsf{tx}$, as $D_1.\mathsf{T}$ does not contain $\mathsf{tx}$, and $\mathsf{tx}'_{i,k'_i} = \mathsf{tx}$. If $k_i \leq k'_i$, the log of (1) should be identical with the first $k_i$ positions of the log of (3), which would imply that $\mathsf{ts}_{i,k_i} = \mathsf{ts}'_{i,k_i}$ and, since a valid pod only accepts non-decreasing timestamps, $\mathsf{ts}'_{i,k_i} \leq \mathsf{ts}'_{i,k'_i}$, and all together $\mathsf{ts}_{i,k_i} \leq \mathsf{ts}'_{i,k'_i}$. This is impossible, because $\mathsf{ts}_{i,k_i} > \mathsf{r}_{\text{perf}}$ and $\mathsf{ts}'_{i,k'_i} < \mathsf{r}_{\text{perf}}$. Hence, the two logs will contain a different timestamp for some sequence number in $[1, k'_i]$.

Summarizing, we have shown for at least $\beta + 1$ replicas $R_i \in \mathcal{R}_1 \cap \mathcal{R}_2$, certificate $C_1$ and $C_2$ contain votes $(\mathsf{tx}_1, \mathsf{ts}_1, \mathsf{sn}_1, \sigma_1, R_i)$ and $(\mathsf{tx}_2, \mathsf{ts}_2, \mathsf{sn}_2, \sigma_2, R_i)$, such that $\mathsf{sn}_1 = \mathsf{sn}_2$ but $\mathsf{tx}_1 \neq \mathsf{tx}_2$ or $\mathsf{ts}_1 \neq \mathsf{ts}_2$. On input a set $T$ that contains these votes, function $\mathsf{identify}(T)$ returns $\mathcal{R}_1 \cap \mathcal{R}_2$.

**2.** If the confirmation-bounds property (Definition 11) is violated, there exists a partial transcript $T$, such that Algorithm 8 on input $T$ returns at least $\beta$ replicas.

*Proof:* As in the proof of Lemma 25, assume the adversary outputs $(D_1, C_1)$ and $(D_2, C_2)$, such that $\mathit{valid}(D_1, C_1) \wedge \mathit{valid}(D_2, C_2)$ and there exists a transaction $\mathsf{tx}$ such that $(\mathsf{tx}, \mathsf{r}^1_{\text{min}}, \mathsf{r}^1_{\text{max}}, \mathsf{r}^1_{\text{conf}}) \in D_1.\mathsf{T}$, $(\mathsf{tx}, \mathsf{r}^2_{\text{min}}, \mathsf{r}^2_{\text{max}}, \mathsf{r}^2_{\text{conf}}) \in D_2.\mathsf{T}$, and $\mathsf{r}^1_{\text{min}} > \mathsf{r}^2_{\text{conf}} \vee \mathsf{r}^1_{\text{max}} < \mathsf{r}^2_{\text{conf}}$ Let $C_1 = (C^1_{\text{pp}}, \mathbb{C}^1_{\text{tx}})$ and $C_2 = (C^2_{\text{pp}}, \mathbb{C}^2_{\text{tx}})$, and $C^1_{\text{tx}} = \mathbb{C}^1_{\text{tx}}[\mathsf{tx}]$ and $C^2_{\text{tx}} = \mathbb{C}^2_{\text{tx}}[\mathsf{tx}]$.

Let's take the case $\mathsf{r}^1_{\text{min}} > \mathsf{r}^2_{\text{conf}}$ first. From Lemma 20 ($\mathsf{timestamps}$ contains at least $n - \lfloor \alpha/2 \rfloor + \beta$ timestamps $\mathsf{ts}$ such that $\mathsf{ts} \geq \mathsf{r}_{\text{min}}$), there is a set $\mathcal{R}_1$ with at least $n - \lfloor \alpha/2 \rfloor + \beta$ replicas $R_i$, from each of which $\mathbb{C}^1_{\text{tx}}$ contains votes

$$(\mathsf{tx}_{i,1}, \mathsf{ts}_{i,1}, 1, \sigma_{i,1}, R_i), (\mathsf{tx}_{i,2}, \mathsf{ts}_{i,2}, 2, \sigma_{i,2}, R_i), \ldots, (\mathsf{tx}_{i,m_i}, \mathsf{ts}_{i,m_i}, m_i, \sigma_{i,m_i}, R_i), \tag{4}$$

up to some sequence number $m_i$, such that $\mathsf{ts}_{i,m_i} \geq \mathsf{r}_{\text{min}}$ and either $\mathsf{tx}_{i,m_i} = \mathsf{tx}$ (i.e., a vote from $R_i$ on $\mathsf{tx}$ is included in $C^1_{\text{tx}}$, and we only consider the votes up to this one), or $\mathsf{tx}_{i,j} \neq \mathsf{tx}, \forall j \leq m_i$ (i.e., a vote from $R_i$ on $\mathsf{tx}$ is not included in $C^1_{\text{tx}}$, in which case $\mathsf{timestamps}$ contains the timestamp $R_i$ has sent on $\mathsf{tx}_{i,m_i} \neq \mathsf{tx}$).

Now, for a valid $D_2$ to output $\mathsf{r}^2_{\text{conf}} < \mathsf{r}^1_{\text{min}}$, certificate $C^2_{\text{tx}}$ must contain timestamps smaller than $\mathsf{r}_{\text{min}}$ from at least $\lfloor \alpha/2 \rfloor + 1$ replicas. Call this set $\mathcal{R}_2$. From each of these replicas, certificates $\mathbb{C}^2_{\text{tx}}$ must contain votes

$$(\mathsf{tx}'_{i,1}, \mathsf{ts}'_{i,1}, 1, \sigma'_{i,1}, R_i), (\mathsf{tx}'_{i,2}, \mathsf{ts}'_{i,2}, 2, \sigma'_{i,2}, R_i), \ldots, (\mathsf{tx}, \mathsf{ts}'_{i,m'_i}, m'_i, \sigma'_{i,m'_i}, R_i), \tag{5}$$

considering only votes up to $\mathsf{tx}$, for which $\mathsf{ts}'_{i,m'_i} < \mathsf{r}_{\text{min}}$.

By counting arguments there are at least $\beta + 1$ replicas in $\mathcal{R}_1 \cap \mathcal{R}_2$. For each one, we make the following argument. Since $\mathsf{ts}_{i,m_i} \geq \mathsf{r}_{\text{min}}$ and $\mathsf{ts}'_{i,m'_i} < \mathsf{r}_{\text{min}}$, we get $\mathsf{ts}'_{i,m'_i} < \mathsf{ts}_{i,m_i}$, and it must be the case that $m'_i < m_i$ (otherwise, the two logs will differ at a smaller sequence number, similar to the previous case). But in this case the two logs differ at sequence number $m'_i$, i.e., $\mathsf{tx}_{i,m'_i} \neq \mathsf{tx}'_{i,m'_i} = \mathsf{tx}$. This is because the log of (4) either does not contain $\mathsf{tx}$, or contains it at sequence number $m_i > m'_i$, in which case it must contain a different transaction at sequence number $m'_i$. On input a set $T$ that contains all votes for replicas in $\mathcal{R}_1$ and $\mathcal{R}_2$ votes, function $\mathsf{identify}(T)$ returns $\mathcal{R}_1 \cap \mathcal{R}_2$.

For the case $\mathsf{r}^1_{\text{max}} < \mathsf{r}^2_{\text{conf}}$, similar arguments apply. In order to compute $\mathsf{r}^2_{\text{conf}} > \mathsf{r}^1_{\text{max}}$, certificate $C^2_{\text{tx}}$ must contain at least $\lfloor \alpha/2 \rfloor$ or $\lfloor \alpha/2 \rfloor + 1$ (depending on the parity of $\alpha$) votes on $\mathsf{tx}$ with timestamp larger than $\mathsf{r}_{\text{max}}$. On the other hand, from Lemma 20 certificate $C^1_{\text{tx}}$ contains at least $n - \alpha + \lfloor \alpha/2 \rfloor + \beta$ votes on $\mathsf{tx}$ with a timestamp smaller or equal than $\mathsf{r}_{\text{max}}$. As before, the replicas in the intersection of these two sets have sent conflicting votes for some sequence numbers.

**3.** The identify() function never outputs honest replicas.
*Proof:* The function only adds a replica to $\tilde{R}$ if given as input two vote messages from that replica, where the same sequence number is assigned to two different votes (line 7 on Algorithm 8). An honest replica always increments nextsn after each vote it inserts to its log (line 21 on Algorithm 1), hence, the adversary can only construct valid votes by forging a signature under the public key of an honest replica, which happens with negl. probability. ◀