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

Orestis Alpos[1], Bernardo David[1,2], Jakov Mitrovski[1,3], Odysseas Sofikitis[1,4], and Dionysis Zindros[1,4]

[1] Common Prefix
[2] IT University of Copenhagen (ITU)
[3] Technical University of Munich
[4] pod network

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 2δ, or one round-trip, *i.e.*, requiring only one network trip (duration δ) 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 assigns 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 2δ, 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.

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 [14]. 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 [13], 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 2δ, letting δ 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 2δ time, regardless of the current value of δ and corrupted parties' actions. In this context, this work is 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.

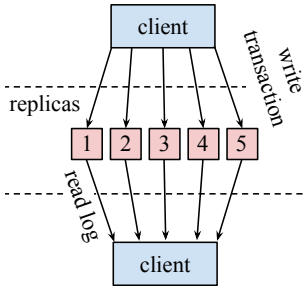

Fig. 1: pod-core's simple architecture. A writing client (top) sends a transaction to all replicas (middle). Each replica appends it to its own log and transmits it to the reading client (bottom).

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 [13], 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 2δ, 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 β Byzantine and γ 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 β Byzantine replicas and up to γ 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 r_{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 r_{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 r_{perf}. A pod also guarantees *past-perfection within w*, meaning that r_{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 δ is the current network delay (not a delay upper bound, which we assume to be unknown). Our construction relies on a set of n *replicas* to maintain a pod data structure, which can be read by an unknown number of clients. The only communication is between each client and the replicas, not among clients nor among 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 $\mathsf{r}_{\min}$ and $\mathsf{r}_{\max}$ from the received timestamps. If the client receives responses from enough replicas, r_{conf} is determined by taking the median of the timestamps received from these replicas.

Protocol pod-core supports a continuum of mixed adversarial models, tolerating up to β Byzantine *and* at the same time up to γ additional omission-faulty replicas.

Applications. The efficiency of pod has the potential to allow for a plethora of distributed applications to be implemented with low latency. In Section 6 we show how auctions can be run on top of pod, achieved through bidset, a new primitive for collecting a set of bids in a censorship resistant manner. It is straightforward to realize single-shot open bid auctions using our bidset primitive based on pod. We also conjecture that protocols for distributed sealed bid auctions based on public bulletin board can also be recast over this primitive. Moreover, we conjecture that consensusless payment systems, such as Fastpay [4], can also be easily realized over pod.

1.3 Related work

Reducing latency. Many previous works have lowered the latency of ordering transactions. HotStuff [26] 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 8δ in the happy path. Jolteon [15], Ditto [15], and HotStuff-2 [18] are two-round versions of HotStuff with end-to-end latency of 5δ. MoonShot [9] allows leaders to send a new proposal every δ time, before receiving enough votes for the previous one, but still achieves an end-to-end latency of 5δ. In the "DAG-based" line of word, Tusk [7] achieves and end-to-end latency of 7δ, the partially-synchronous version of BullShark [22] an end-to-end latency of 5δ, and Mysticeti [2] an end-to-end latency of 4δ. 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.

Auctions. The pod notion offers the *past-perfection* property: a *read*() operation outputs a timestamp r_{perf}, and it is *guaranteed* that the output of *read*() contains all transactions that can ever be confirmed with a timestamp smaller than r_{perf} in the view of any reading client, regardless of the network conditions. This implies that reading clients (such as an auctioneer) cannot claim not having received a transaction when reading the pod, as this is detectable by any other client who reads the pod. To the best of our knowledge, previous work in the consensusless literature has not considered or achieved this property, hence it cannot readily support auctions.

Consensusless payments. The redundancy of consensus for implementing payment systems has been recognized by previous works [4,6,17,21]. 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 tx_1 and 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.* [17] as the *source-order property*. The constructions of Guerraoui *et al.* [17] and FastPay [4] require clients to maintain sequence numbers. ABC [21] requires clients to reference all previous transaction in a DAG (including its own last transaction). Cheating clients might lose liveness [4,17,21], but equivocating is not possible.

2 Preliminaries

Notation. We denote by $\mathbb{N}$ the set of natural numbers including 0. Let L be a sequence, we denote by $L[i]$ the i^{th} element (starting from 0), and by $|L|$ its length. Negative indices address elements from the end, so $L[-i]$ is the i^{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 $L_1 \,\|\, L_2$.

2.1 Execution Model

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 any two honest parties, their clocks can be at most ϕ rounds apart. For simplicity, our analysis will assume *synchronized clocks*, that is, $\phi = 0$. Notice that although we assume synchronized clocks as a setup, clock synchronization can be achieved in partially synchronous networks [10] using existing techniques [20], also in the case where replicas gradually join the network [25]. 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, Δ is *known* to all parties. In the *partially synchronous* model [10], Δ 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 Δ and its liveness guarantees depend only on the actual network delay δ.

2.2 Cryptographic primitives

Digital Signatures. We assume that replicas (and auctioneers in bidset-core) authenticate their messages with digital signatures. A digital signature scheme is a triple of algorithms satisfying the EUF-CMA security [16] as defined below:

- $\mathit{KeyGen}(1^\kappa)$: The key generation algorithm takes as input a security parameter κ and outputs a secret key sk and a public key pk.
- $\mathit{Sign}(\mathsf{sk}, m) \to \sigma$: The signing algorithm takes as input a private key sk and a message $m \in \{0,1\}^*$ and returns a signature σ.
- $\mathit{Verify}(\mathsf{pk}, m, \sigma) \to b \in \{0,1\}$: The verification algorithm takes as input a public key pk, a message m, and a signature σ, and outputs a bit $b \in \{0,1\}$.

We say σ is a *valid* signature on m with respect to pk if $\mathit{Verify}(pk, m, \sigma) = 1$.

2.3 Accountable safety

Taking a similar approach as Neu, Tas, and Tse [19, 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 (β-Accountable safety). *A protocol satisfies* accountable safety *with* resilience β *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 1. For the sake of 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 pod, a distributed protocol where clients can *read* and *write* transactions. We first define basic data structures and the interface of a pod protocol.

Definition 3 (Transaction trace and trace set). *The* transaction trace *of a transaction* $\mathsf{tx} \in \{0,1\}^*$ *is a tuple containing the values* $(\mathsf{tx}, \mathsf{r}_{min}, \mathsf{r}_{max}, \mathsf{r}_{conf})$*, which change during the execution of a* pod *protocol. We call* $\mathsf{r}_{min} \in \mathbb{N}$ *the* minimum round*,* $\mathsf{r}_{max} \in \mathbb{N} \cup \{\infty\}$ *the* maximum round*,* $\mathsf{r}_{conf} \in \mathbb{N} \cup \{\perp\}$ *the* confirmed round*. We denote by* $\mathsf{r}_{max} = \infty$ *an unbounded maximum round and by* $\mathsf{r}_{conf} = \perp$ *an undefined confirmed round. We also denote these values as* $\mathsf{tx}.\mathsf{r}_{min}$*,* $\mathsf{tx}.\mathsf{r}_{max}$*, and* $\mathsf{tx}.\mathsf{r}_{conf}$*. A* trace set T *is a set of transaction traces* $\{(\mathsf{tx}, \mathsf{r}_{min}, \mathsf{r}_{max}, \mathsf{r}_{conf}) \mid \mathsf{tx} \in \{0,1\}^*\}$*.*

Definition 4 (Confirmed transaction). *A transaction with confirmed round r_{conf} is called* confirmed *if $\mathsf{r}_{conf} \neq \perp$, and* unconfirmed *otherwise.*

Definition 5 (Pod data structure). *A* pod data structure *D is a tuple $(\mathsf{T}, \mathsf{r}_{perf})$, where 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 T may be confirmed on unconfirmed. Moreover, $\mathsf{r}_{\mathrm{perf}}$ will be used to define a completeness property on T (the *past-perfection* property of pod).

Definition 6 (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 may be helpful to mention that in pod-core it will be a tuple $C = (C_{pp}, \mathbb{C}_{tx})$. C_{pp} will be called the* past-perfection certificate *and $\mathbb{C}_{tx}$ will be a map from each transaction tx in $D.\mathsf{T}$ to a* transaction certificate *C_{tx} for tx. Both will contain digital signatures.*

Definition 7 (Interface of a pod). *A pod protocol has the following interface.*

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

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

Definition 8 (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 9 (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 c and the output is produced at round r. We remark that 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, which we later use for some applications.

Definition 10 (Secure pod). *A protocol is a* secure *pod if it implements the pod interface of Definition 7 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 c writes a transaction tx at round r, then for any honest client c' (including $\mathsf{c} = \mathsf{c}'$) it holds that $\mathsf{tx} \in D^{\mathsf{c}'}_{\mathsf{r}+u}$ and $\mathsf{tx}.\mathsf{r}_{conf} \neq \perp$.*
(Liveness) Past-perfection within w: *Rounds become past-perfect after at most w rounds. Formally, for any honest client c and round $\mathsf{r} \geq w$, it holds that $D^{\mathsf{c}}_{\mathsf{r}}.\mathsf{r}_{perf} \geq \mathsf{r} - w$.*
(Safety) Past-perfection: *A valid pod D contains all transactions that may ever obtain a confirmed round smaller than $D.\mathsf{r}_{perf}$. 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 tx such that $(\mathsf{tx}, \mathsf{r}^1_{min}, \mathsf{r}^1_{max}, \mathsf{r}^1_{conf}) \notin D_1.\mathsf{T}$ and $(\mathsf{tx}, \mathsf{r}^2_{min}, \mathsf{r}^2_{max}, \mathsf{r}^2_{conf}) \in D_2.\mathsf{T}$ and $\mathsf{r}^2_{conf} \neq \perp$ and $\mathsf{r}^2_{conf} < D_1.\mathsf{r}_{perf}$.*
(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 tx such that $(\mathsf{tx}, \mathsf{r}^1_{min}, \mathsf{r}^1_{max}, \mathsf{r}^1_{conf}) \in D_1.\mathsf{T}$ and $(\mathsf{tx}, \mathsf{r}^2_{min}, \mathsf{r}^2_{max}, \mathsf{r}^2_{conf}) \in D_2.\mathsf{T}$ and $\mathsf{r}^1_{min} > \mathsf{r}^2_{conf}$ or $\mathsf{r}^1_{max} < \mathsf{r}^2_{conf}$.*

The *confirmation bounds* property gives $\mathsf{r}^1_{\min} \leq \mathsf{r}^2_{\text{conf}} \leq \mathsf{r}^1_{\max}$, for $\mathsf{r}^1_{\min}, \mathsf{r}^1_{\max}, \mathsf{r}^2_{\text{conf}}$ computed by honest clients, but it does not guarantee anything about the values of $\mathsf{r}^1_{\min}$ and $\mathsf{r}^1_{\max}$ (for example, it could trivially be $\mathsf{r}^1_{\min} = 0$ and $\mathsf{r}^1_{\max} = \infty$). 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 [24].

Definition 11 (pod θ-timeliness for honest transactions). *A pod protocol is* θ-timely *if it is a secure pod, as per Definition 10, and for any honest clients* $\mathsf{c}_1, \mathsf{c}_2$*, if* c_1 *writes transaction* tx *in round* r *and* c_2 *has view* $D^{\mathsf{c}_2}_{r'}$ *in round* r'*, such that* $(\mathsf{tx}, r_{min}, r_{max}, r_{conf}) \in D^{\mathsf{c}_2}_{r'}.T$*, then:*

1. $r_{conf} \in (r, r+\theta]$
2. $r_{max} \in (r, r+\theta]$
3. $r_{max} - r_{min} < \theta$*, implying that* $r_{min} \neq 0$ *and* $r_{max} \neq \infty$.

Moreover, a pod protocol allows the values $\mathsf{r}_{\min}$, $\mathsf{r}_{\max}$, r_{conf} to change during an execution – for example, clients in construction pod-core will update them when they receive votes from replicas. The properties we have defined so far do not impose any restriction on how they evolve. For this reason, in Appendix A we define the additional property of pod *monotonicity*.

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

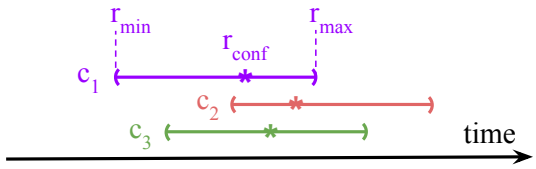

Fig. 2: The same transaction in the view of three different pod clients. Each client assigns it a minimum round $\mathsf{r}_{\min}$ and a maximum round $\mathsf{r}_{\max}$. If it gets confirmed, the confirmation round r_{conf} will be between these two values. The r_{conf} that each client locally computes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

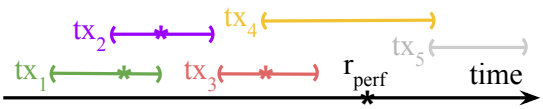

Fig. 3: A possible view of a single pod client. Transactions $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3$ are confirmed, tx_4 is not yet confirmed. A client also derives a past-perfect round r_{perf}. No transaction other than $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_4$ may obtain $\mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\text{perf}}$. There may exist tx_5 for which the client has not received votes, but 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 12 (Vote). *A* vote *is a tuple* $\mathsf{vote} = (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R)$*, where* tx *is a transaction,* ts *is a timestamp,* sn *is a sequence number,* σ *is a signature, and* R *is a replica. A vote is* valid *if* σ *is a valid signature on message* $m = (\mathsf{tx}, \mathsf{ts}, \mathsf{sn})$ *with respect to the public key* pk_R *of replica* R.

Remark 2 (Processing votes in order). We require that clients process votes from each replica in the same order, namely in order of increasing timestamps. For this we employ *sequence numbers*. Each replica maintains a sequence number, which it increments and includes every time it assigns a timestamp to a transaction.

Remark 3 (Implicit session identifiers). We 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. Moreover, the sid is implicitly included in all messages signed by the replicas.

Remark 4 (Streaming construction). 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_{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., $\mathbb{C}_{\text{tx}} : \mathsf{tx} \to C_{\text{tx}}$ and $C_{\text{tx}} : R \to \mathsf{vote}$. We remark that C_{pp} can be derived by taking the union of certificates C_{tx} for all transactions and keeping the most recent vote for each replica, but we define C_{pp} explicitly for clarity and readability.

Pseudocode notation. The notation '**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.

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 β, γ and α, where β denotes the number of Byzantine replicas and γ 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

The state of a replica (lines 1–3 of Algorithm 1) 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 σ is its signature. When the replica receives a $\langle CONNECT \rangle$ message 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 $\langle WRITE\ \mathsf{tx} \rangle$, a replica first checks whether it has already seen tx, in which case the message is ignored. Otherwise, it assigns tx a timestamp ts equal its local round number and the next available sequence number 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 replicaLog, and sends it via a $\langle VOTE\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_i) \rangle$ message to all connected clients (line 21).

Heartbeat messages. As we will see, 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 r_{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 25–28). 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 13 in Algorithm 2). To avoid being considered a duplicate vote by clients (see line 38 in Algorithm 2), replicas append the round number to the HEARTBEAT transaction.

4.2 Client code

Initialization. The state of a client is shown in Algorithm 2 in lines 2–8. The state contains the identifiers and public keys of all replicas, mrt, nextsn, tsps, D, C_{pp}, and C_{tx}. Variable tsps is a map from transactions tx to a map from replicas R to timestamps ts. The state gets initialized in lines 9–12. At initialization the client also sends a $\langle CONNECT \rangle$ message to each replica, which initiates a streaming connection from the replica to the client.

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))   ▷ round() returns the current round
19:     replicaLog ← replicaLog ‖ (tx, ts, sn, σ)
20:     for c ∈ 𝒞 do
21:         ⟨VOTE (tx, ts, sn, σ, R_i)⟩ → c
22:     end for
23:     nextsn ← nextsn + 1
24: end function

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

Receiving votes. A client maintains a connection to each replica and receives votes through $\langle VOTE\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)\rangle$ messages (lines 13–18). When a vote is received from replica R_j, the client first verifies the signature σ under R_j's public key (line 33). 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 34). 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, ts must be greater or equal to mrt_j, the most recent timestamp returned by replica R_j (line 36). Second, the replica must have not previously sent a different timestamp for tx (line 38). If both checks pass, the client updates $\mathsf{mrt}[j]$ (line 37) and $\mathsf{tsps}[\mathsf{tx}][R_j]$ (line 39) with ts. The client also updates C_{pp} and $\mathbb{C}_{\text{tx}}$ (lines 15 and 16) for each valid vote.

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 $\langle VOTE\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)\rangle$ 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 $\langle VOTE\ (\mathsf{tx}, \mathsf{ts}, \mathsf{sn}, \sigma, R_j)\rangle$ and the previous vote it has received for tx. The identify() function we show in Algorithm 8 can detect such misbehavior. However, in this paper we formalize accountability conditioned on safety being violated (Definition 2), hence we do not further explore this.

Writing to and reading from pod. Clients interact with a pod using the *write*(tx) and *read*() functions. In order to write a transaction tx, a client sends $\langle WRITE\ \mathsf{tx}\rangle$ to each replica (lines 19–21). Since the construction is stateful and streaming, the client state contains at all times the latest view the client has of the pod. Hence, *read*() operates on the local state (lines 22–27). It returns all the transactions the client has received so far and their traces, and the current past-

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

1: **State:**
2: $\mathcal{R} = \{R_1, \ldots, R_n\}$; $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ ▷ All replicas and their public keys
3: $\mathsf{mrt} : R \to \mathsf{ts}$ ▷ The most recent timestamp returned by each replica
4: $\mathsf{nextsn} : R \to \mathsf{sn}$ ▷ The next sequence number expected by each replica
5: $\mathsf{tsps} : \mathsf{tx} \to (R \to \mathsf{ts})$ ▷ Timestamp received for each tx from each replica
6: $D = (\mathsf{T}, \mathsf{r}_{\text{perf}})$ ▷ The pod observed by the client so far
7: $C_{\text{pp}} : R \to \mathsf{vote}$ ▷ Past-perfection certificate: the most recent vote from each replica
8: $\mathbb{C}_{\text{tx}} : \mathsf{tx} \to C_{\text{tx}}$, where $C_{\text{tx}} : R \to \mathsf{vote}$ ▷ Transaction certificate: for each transaction, all votes

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

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

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

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

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

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

perfect round r_{perf}. We will show the details of *computeTxSet*() in Algorithm 3. As per thepod interface, *read*() also returns auxiliary data C, which in the implementation of pod-core has two parts: the past-perfection certificate C_{pp} and a list of transaction certificates C_{tx} (line 25). Note that tsps.*keys()* on line 3 returns all entries in tsps.

Computing the trace values and the past-perfect round. In Algorithm 3 we show function *computeTxSet*(), used to compute the current transaction set from the timestamps tsps received so far. A transaction becomes confirmed when the client receives α votes for tx from different replicas (line 7), in which case r_{conf} is the median of all received timestamps (line 9). The computation of $\mathsf{r}_{\text{min}}, \mathsf{r}_{\text{max}}$, and r_{perf} is done using the functions *minPossibleTs*(), *maxPossibleTs*(), and *computePastPerfectRound*(), respectively.

Algorithm 3 Protocol pod-core: Client code, part 2. Functions to compute trace values and past-perfect round. The code is parametrized with β, the number of Byzantine replicas expected by the client, and γ, 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)
5:         r_max ← maxPossibleTs(tsps[tx])
6:         r_conf ← ⊥; timestamps = [ ]
7:         if |tsps[tx].keys()| ≥ α then
8:             for R_j ∈ tsps[tx].keys() do:  timestamps ← timestamps || tsps[tx][R_j]
9:             r_conf ← median(timestamps)
10:        end if
11:        T ← T ∪ {(tx, r_min, r_max, r_conf)}
12:    end for
13:    return T
14: end function

15: function minPossibleTs(timestamps, mrt)      ▷ timestamps : R → ts, contains timestamps on tx
16:    for R_j ∈ R do                             ▷ mrt : R → ts, most recent tsp from each replica
17:        if timestamps[R_j] = ⊥ then timestamps ← timestamps || [mrt[R_j]]
18:    end for
19:    sort timestamps in increasing order of timestamps
20:    timestamps ← [0, β times ..., 0] || timestamps                ▷ omitted altogether if β = 0
21:    return median(timestamps[: α])
22: end function

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

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

36: function median(Y)
37:    return Y[⌊ |Y|/2 ⌋]
38: end function
```

Function *minPossibleTs*() gets as input the timestamps timestamps from each replica on tx and the most recent timestamps mrt from the replicas. It fills a missing timestamp from replica R_j with $\mathsf{mrt}[R_j]$ (line 17), the minimum timestamp that can ever be accepted from R_j (smaller values will not pass the check in line 36 of Algorithm 2). It then prepends β times the 0 value (line 20), pessimistically assuming that up to β replicas will try to bias tx by sending a timestamp 0 to other clients, which only happens if replicas may be Byzantine, i.e., if $\beta > 0$. It then returns the median of the α smallest timestamps, which, again pessimistically, are the smallest timestamps another client may use to confirm tx.

Function *maxPossibleTs*() is analogous, filling a missing vote with ∞ (line 25) and appending the ∞ value (line 28), the worst-case timestamp that Byzantine replicas may send to other clients, and returning the median of the α largest timestamps.

Finally, *computePastPerfectRound*() is similar to *minPossibleTs*() but it operates on the timestamps mrt, instead of votes on a specific transaction. Hence, since an honest client will not

accept a timestamp smaller than mrt on any future transaction (line 36 of Algorithm 2), the returned value bounds from below the confirmed round that *any* honest client can ever assign to a transaction *not yet seen*.

4.3 Validation function

The purpose of the validation function *valid*() is to allow a 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 10).

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

The function *valid*() for pod-core is shown in Algorithm 4. 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 is expected to contain two parts, a *past-perfection certificate* C_{pp} and a collection of *transaction certificates* $\mathbb{C}_{\text{tx}}$, once for each transaction in $D.\mathsf{T}$ (line 3). Both contain vote messages, as constructed by a pod client in lines 15 and 16 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 T and the past-perfect round r_{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_{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_{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.

4.4 Analysis

Theorem 1 (pod-core security). *Assume that the network is partially synchronous with actual network delay δ, that β is the number of Byzantine replicas, γ 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 4, and the identify() *function described in Algorithm 8, is a responsive secure* pod *(Definition 10) with Confirmation within $u = 2\delta$, Past-perfection within $w = \delta$ and β-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[5] 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 α) from different replicas and the initial timestamp recorded by the writing client. We present the results in Figure 4.

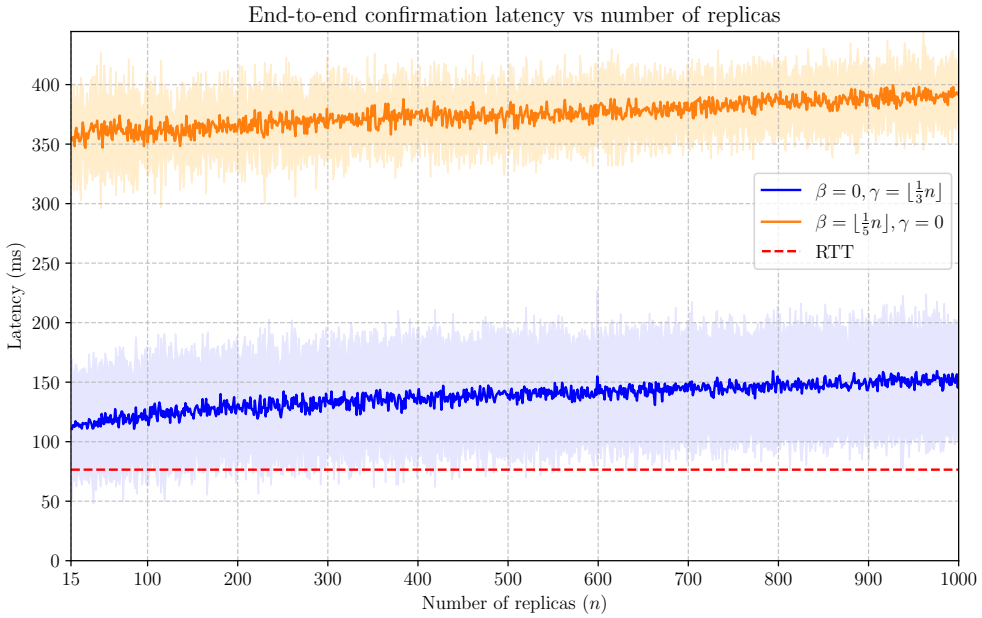

Fig. 4: End-to-end confirmation latency from a writing client to a reading client as a transaction traverses across $n = 15, \ldots, 1000$ replicas, for two reading clients: (1) a client that expects up to $\gamma = \lfloor \frac{1}{3}n \rfloor$ omission faults (blue line, below), and (2) a client that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

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[6] 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: eu-central-1 (Frankfurt), eu-west-2 (London), us-east-1 (N. Virginia), us-west-1 (N. California), ca-central-1 (Canada), ap-south-1 (Mumbai), and ap-northeast-2 (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.

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

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 eu-west-2 (London) and the writing client in us-east-1 (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 α replicas have responded, which gives rise to the happy artifact that the 1 - α 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* 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. We first define bidset and then construct it using an underlying pod.

Remark 5 (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.

Definition 13 (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_{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)**.** *Protocol bidset-core is parameterized by an integer* Δ *(looking ahead, we will prove security in synchrony, i.e., assuming the network delay* δ *is smaller than* Δ*) and assumes digital signatures and a pod with* δ*-timeliness,* $w = \delta$ *and* $u = 2\delta$*. At time* t_0*, all parties start executing Algorithms 5–7. A pre-appointed* sequencer *is responsible to reading the pod and writing back to it when a specific condition is met. For example, when instantiating bidset-core on top of pod-core, a replica can act as sequencer.*

Algorithm 5 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 6 bidset-core: Code for the sequencer. It runs a client for a pod-core instance *pod*, and 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
4:         ((T, r_perf), (C_pp, C_tx)) ← pod.read()
5:     end while
6:     B ← {tx | (tx, ·, ·, ·) ∈ T}; C_bid ← C_pp
7:     σ ← Sign(sk_a, (B, C_bid))
8:     tx ← ⟨BIDS (B, C_bid, σ)⟩
9:     pod.write(tx)
10: end function
```

Algorithm 7 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
```

A bidder (Algorithm 5) submits a bid by writing it on the pod at round t_0. The sequencer (Algorithm 6) waits until the pod returns a past-perfect round larger than $t_0 + \Delta$ (line 3) and then constructs the bid-set B from the set of transactions in T (line 6). The sequencer concludes by signing B and C_{bid} (which can be used as evidence, in case of a safety violation) and writing $\langle \mathit{BIDS}\ (B, C_{\text{bid}}, \sigma)\rangle$ on *pod*.

The code for a consumer is shown in Algorithm 7. The consumer waits until one of the following two conditions is met. First, a *confirmed* transaction $\langle \mathit{BIDS}\ (B, C_{\text{bid}}, \sigma)\rangle$ appears in T, for which $\mathsf{r}_{\text{conf}} \leq t_0 + 3\Delta$ (line 4), in which case it outputs bid-set B as result. Second, a round higher than $t_0 + 3\Delta$ becomes past-perfect in pod (line 6) without a confirmed $\langle \mathit{BIDS}\ \rangle$ transaction appearing, in which case it outputs $B = \emptyset$.

As an intuition on how bidset-core achieves censorship resistance, we observe the following. The *δ-timeliness property* of pod (Definition 11), given that $\delta \leq \Delta$, ensures that bids of honest parties will have a confirmed round $\mathsf{r}_{\text{conf}} \leq t_0 + \Delta$. Now, the sequencer may only produce a valid bid-set when pod returns a past-perfect round larger than $t_0 + \Delta$ (line 3), and the output *must* contain a certificate C_{pp} that proves this. However, if the certificate is valid, then the sequencer must have *provably* seen the bids of honest parties in T (we remind that the votes of replicas on pod-core are chained using sequence numbers), and thus B must contain all bids with $\mathsf{r}_{\text{conf}} \leq t_0 + \Delta$. If any party presents a transaction certificate C_{tx} for some transaction tx^* with $\mathsf{r}^*_{\text{conf}} \leq t_0 + \Delta$, but $\mathsf{tx}^* \notin B$, then the sequencer can be held accountable. We show the detailed proof in Lemma 9 and Lemma 11.

Regarding liveness, line 3 of Algorithm 6 becomes true in the view of sequencer by round $t_0 + \Delta + \delta$ (from the *past-perfection within $w = \delta$* property of pod-core), hence Algorithm 6 for an honest sequencer terminates by that round. Observe also that the transaction $\langle \mathit{BIDS}\ (B, C_{\text{bid}}, \sigma)\rangle$ becomes confirmed in the view of all honest clients by round $t_0 + \Delta + 3\delta$ (from the *confirmation within $u = 2\delta$* property), and it will have a confirmed round $\mathsf{r}_{\text{conf}} \leq t_0 + \Delta + 2\delta$ (from

the δ-*timeliness* property). Hence, if the network is synchronous and the sequencer honest, the condition in line 4 of Algorithm 7 becomes true at round at most $t_0 + \Delta + 3\delta$, Even if the sequencer is malicious, from the *past-perfection within* $w = \delta$ property of pod, the condition in line 6 will become true at latest at round $t_0 + 3\Delta + \delta$, hence bidset-core achieves *termination within* $W = 3\Delta + \delta$.

Theorem 2 (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* δ-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. The proof and identifySequencer() are shown in Appendix D. □

Remark 6. 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 6 waits for a fixed Δ interval. This step can be optimized if the set of bidders is known (i.e., by requiring them to pre-register), which allows for the protocol to be made optimistically responsive (*i.e.*, $W = 4\delta$) when all bidders and the sequencer are honest.

Auctions using bidset. Building on a bidset protocol, it is trivial to construct single-shot first price and second price open auctions as follows: 1. Bidders place their open bids b by calling *submitBid(b)*; 2. Consumers determine the winner by calling *readResult()* to obtain B and outputting either the first or second highest bid. We conjecture that single-shot sealed bid auction protocols such as those of [3, 5, 8, 11, 12, 23] 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 execute extra steps to determine the winner. However, a formal analysis of sealed-bid auction protocols based on bidset is left as future work.

7 Discussion

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 replica has a particular role in pod (as compared to leaders, block proposers or miners in similar protocols), pod achieves censorship resistance by default, without any extra mechanisms or additional cost. Furthermore, replica misbehavior, such as voting in incompatible ways or censoring confirmed transactions, is accountable.

Regarding applications, we have presented an efficient and censorship-resistant auction mechanism, 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. Similar to auctions, pod can enable censorship-resistant voting applications – pod guarantees that no single party or authority can censor or delay a valid vote.

Moreover, payments can be realized on top of pod. We leave the complete specification as future work, but outline here two ways in which this can be achieved. The first is by making the replicas stateful, in which case pod can directly support a protocol similar to FastPay [4]. The second option is to implement the payment logic on the client side, hence leaving pod stateless. This can be achieved using the past-perfection property of pod: the sender of a payment writes the payment transaction to pod; the recipient waits until the transaction becomes confirmed and its confirmed round becomes past-perfect; the recipient can then verify whether the sender has created a conflicting transaction before it. Compared to the solution of FastPay, the second approach has the advantage that clients do not need to maintain sequence numbers.

We remark that pod differs from standard notions of consensus because it does not offer an agreement property, neither to replicas nor to clients. A client reading the pod obtains a past-perfect round r_{perf}, and it is guaranteed to have received all transactions that obtained a confirmed round r_{conf} such that $\mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\text{perf}}$. It is also guaranteed to have received all transactions

that can potentially obtain an $\mathsf{r}_{\text{conf}} \leq \mathsf{r}_{\text{perf}}$ 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 will be transaction written by a malicious client).

References

1. M. K. Aguilera and S. Toueg. A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999.
2. K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, and A. Sonnino. Mysticeti: Low-latency DAG consensus with fast commit path. *CoRR*, abs/2310.14821, 2023.
3. S. Bag, F. Hao, S. F. Shahandashti, and I. G. Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15:2042–2052, 2020.
4. M. Baudet, G. Danezis, and A. Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *AFT*, pages 163–177. ACM, 2020.
5. T. Chitra, M. V. X. Ferreira, and K. Kulkarni. Credible, Optimal Auctions via Public Broadcast. In R. Böhme and L. 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.
6. D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. 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.
7. G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In *EuroSys*, pages 34–50. ACM, 2022.
8. B. David, L. Gentile, and M. Pourpouneh. FAST: Fair auctions via secret transactions. In G. Ateniese and D. Venturi, editors, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 727–747. Springer, Cham, June 2022.
9. I. Doidge, R. Ramesh, N. Shrestha, and J. Tobkin. Moonshot: Optimizing chain-based rotating leader BFT via optimistic proposals. *CoRR*, abs/2401.01791, 2024.
10. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.
11. H. S. Galal and A. M. Youssef. Trustee: Full privacy preserving vickrey auction on top of ethereum. In A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, and M. Sala, editors, *Financial Cryptography and Data Security*, pages 190–207, Cham, 2020. Springer International Publishing.
12. C. Ganesh, S. Gupta, B. Kanukurthi, and G. Shankar. Secure vickrey auctions with rational parties. Cryptology ePrint Archive, Paper 2024/1011, 2024. To appear at CCS 2024.
13. J. A. Garay, J. Katz, C. Koo, and R. Ostrovsky. Round complexity of authenticated broadcast with a dishonest majority. In *FOCS*, pages 658–668. IEEE Computer Society, 2007.
14. P. Gazi, L. Ren, and A. Russell. Practical settlement bounds for longest-chain consensus. In H. Handschuh and A. 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.
15. R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. 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.
16. S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.
17. R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi. The consensus number of a cryptocurrency. *Distributed Comput.*, 35(1):1–15, 2022.
18. D. Malkhi and K. Nayak. Extended abstract: Hotstuff-2: Optimal two-phase responsive BFT. *IACR Cryptol. ePrint Arch.*, page 397, 2023.
19. J. Neu, E. N. Tas, and D. Tse. The availability-accountability dilemma and its resolution via accountability gadgets. In I. Eyal and J. A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 541–559. Springer, Cham, May 2022.
20. B. Simons. An overview of clock synchronization. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, pages 84–96, New York, NY, 1990. Springer New York.
21. J. Sliwinski and R. Wattenhofer. ABC: asynchronous blockchain without consensus. *CoRR*, abs/1909.10926, 2019.

22. A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. Bullshark: The partially synchronous version. *CoRR*, abs/2209.05633, 2022.
23. N. Tyagi, A. Arun, C. Freitag, R. Wahby, J. Bonneau, and D. Mazières. Riggs: Decentralized sealed-bid auctions. In W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, editors, *ACM CCS 2023*, pages 1227–1241. ACM Press, Nov. 2023.
24. A. Tzinas, S. Sridhar, and D. Zindros. On-chain timestamps are accurate. Cryptology ePrint Archive, Report 2023/1648, 2023.
25. J. Widder. Booting clock synchronization in partially synchronous systems. In F. E. Fich, editor, *Distributed Computing*, pages 121–135, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
26. M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019.

A Definition of pod monotonicity

The property of pod monotonicity requires that, as time advances, $\mathsf{r}_{\min}$ does not decrease, $\mathsf{r}_{\max}$ does not increase and confirmed transactions remain confirmed.

Definition 14 (pod monotonicity). *A pod protocol satisfies* pod monotonicity*, and is called a* monotone pod*, if it is a secure pod, as per Definition 10, and the following properties hold for any rounds $r_1, r_2 > r_1$ and for any honest client c:*

Past-perfection monotonicity: *It holds that $D^{\mathsf{c}}_{r_2}.r_{perf} \geq D^{\mathsf{c}}_{r_1}.r_{perf}$.*
Transaction monotonicity: *If transaction tx appears in $D^{\mathsf{c}}_{r_1}.\mathsf{T}$, then tx appears in $D^{\mathsf{c}}_{r_2}.\mathsf{T}$.*
Confirmation-bounds monotonicity: *For every tx that appears in $D^{\mathsf{c}}_{r_1}.\mathsf{T}$ with $r_{min}, r_{max}, r_{conf}$ and appears in $D^{\mathsf{c}}_{r_2}.\mathsf{T}$ with $r'_{min}, r'_{max}, r'_{conf}$, it holds that $r'_{min} \geq r_{min}$, $r'_{max} \leq r_{max}$.*

We now observe that a monotone pod protocol can be obtained from any secure pod protocol with stateful clients, and that monotonicity implies certain specific properties that may be useful for applications. In particular, our pod-core protocol naturally satisfies this property.

Remark 7. Every secure pod can be transformed into a monotone pod if parties are stateful. Let r_1 be the last round when an honest client c read the pod obtaining view $D^{\mathsf{c}}_{\mathsf{r}_1}$, which is stored as state until c reads the pod again. At any round $\mathsf{r}_2 > \mathsf{r}_1$, if c reads the pod and obtains $D^{\mathsf{c}}_{\mathsf{r}_2}$, c can define a view $\overline{D^{\mathsf{c}}_{\mathsf{r}_2}}$ satisfying the properties of pod monotonicity:

1. If tx appears in $D^{\mathsf{c}}_{\mathsf{r}_1}$ with $\mathsf{tx}.\mathsf{r}_{\min}, \mathsf{tx}.\mathsf{r}_{\max}, \mathsf{tx}.\mathsf{r}_{\mathrm{conf}}, \mathsf{tx}.C_{\mathrm{tx}}$, then tx appears in $\overline{D^{\mathsf{c}}_{\mathsf{r}_2}}$ with $\mathsf{tx}.\overline{\mathsf{r}_{\min}} = \mathsf{r}_{\min}, \mathsf{tx}.\overline{\mathsf{r}_{\max}} = \mathsf{r}_{\max}, \mathsf{tx}.\overline{\mathsf{r}_{\mathrm{conf}}} = \mathsf{r}_{\mathrm{conf}}, \mathsf{tx}.\overline{C_{\mathrm{tx}}} = C_{\mathrm{tx}}$.
2. If tx appears in $D^{\mathsf{c}}_{\mathsf{r}_2}$ with $\mathsf{tx}.\mathsf{r}'_{\min}, \mathsf{tx}.\mathsf{r}'_{\max}, \mathsf{tx}.\mathsf{r}'_{\mathrm{conf}}, \mathsf{tx}.C'_{\mathrm{tx}}$ and does not appear in $D^{\mathsf{c}}_{\mathsf{r}_1}$, then tx appears in $\overline{D^{\mathsf{c}}_{\mathsf{r}_2}}$ with $\mathsf{tx}.\overline{\mathsf{r}_{\min}} = \mathsf{r}'_{\min}, \mathsf{tx}.\overline{\mathsf{r}_{\max}} = \mathsf{r}'_{\max}, \mathsf{tx}.\overline{\mathsf{r}_{\mathrm{conf}}} = \mathsf{r}'_{\mathrm{conf}}, \mathsf{tx}.\overline{C_{\mathrm{tx}}} = C'_{\mathrm{tx}}$.
3. For every tx that appears in $D^{\mathsf{c}}_{\mathsf{r}_1}.\mathsf{T}$ and in $D^{\mathsf{c}}_{\mathsf{r}_2}.\mathsf{T}$ such that $\mathsf{r}'_{\min} \geq \mathsf{r}_{\min}$, $\mathsf{r}'_{\max} \leq \mathsf{r}_{\max}$, $\mathsf{r}'_{\mathrm{conf}} \geq \mathsf{r}_{\mathrm{conf}}$, update $\mathsf{tx}.\overline{\mathsf{r}_{\min}} = \mathsf{r}'_{\min}, \mathsf{tx}.\overline{\mathsf{r}_{\max}} = \mathsf{r}'_{\max}, \mathsf{tx}.\overline{\mathsf{r}_{\mathrm{conf}}} = \mathsf{r}'_{\mathrm{conf}}, \mathsf{tx}.\overline{C_{\mathrm{tx}}} = C'_{\mathrm{tx}}$.
4. If $D^{\mathsf{c}}_{\mathsf{r}_2}.\mathsf{r}_{\mathrm{perf}} > D^{\mathsf{c}}_{\mathsf{r}_1}.\mathsf{r}_{\mathrm{perf}}$, then $\overline{D^{\mathsf{c}}_{\mathsf{r}_2}}.\mathsf{r}_{\mathrm{perf}} = D^{\mathsf{c}}_{\mathsf{r}_2}.\mathsf{r}_{\mathrm{perf}}$. Otherwise, $\overline{D^{\mathsf{c}}_{\mathsf{r}_2}}.\mathsf{r}_{\mathrm{perf}} = D^{\mathsf{c}}_{\mathsf{r}_1}.\mathsf{r}_{\mathrm{perf}}$.

In the remarks below, we observe that pod monotonicity implies a number of useful properties about the monotonicity of past perfection and the values $\mathsf{r}_{\min}, \mathsf{r}_{\max}, \mathsf{r}_{\mathrm{conf}}$ associated to a transaction in the pod.

Remark 8 (Confirmation monotonicity). Properties 2 and 3 of pod monotonicity imply that for any honest client c and rounds $\mathsf{r}_1, \mathsf{r}_2 > \mathsf{r}_1$, if $\mathsf{tx} \in D^{\mathsf{c}}_{\mathsf{r}_1}$ and $\mathsf{tx}.\mathsf{r}_{\mathrm{conf}} \neq \bot$, then $\mathsf{tx} \in D^{\mathsf{c}}_{\mathsf{r}_2}$ and $\mathsf{tx}.\mathsf{r}_{\mathrm{conf}} \neq \bot$.

Remark 9. Observe that the *confirmation monotonicity* property in Remark 8 is a specific version of a more general *common subset* property, which would demand the condition for any two honest clients $\mathsf{c}_1, \mathsf{c}_2$.

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

In order to prove Theorem 1 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, β denotes the number of Byzantine replicas, γ denotes the number of omission-faulty replicas in an execution, and α denotes the number of replicas required to confirm a transaction.

Lemma 1 (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}, ∞ 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 β times the 0 value in the beginning of the list and return the median of the first α values, hence they return the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$. Function *maxPossibleTs*() appends β times the ∞ value at the end of the list and returns the median of the last α 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 2 ($\mathsf{r}_{\mathbf{perf}}$ bounded by honest timestamp). *Assuming $n \geq 5\beta + 3\gamma + 1$ (equiv., $\alpha \geq 4\beta + 2\gamma + 1$), for a valid 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 $\mathsf{mrt} \leq D.\mathsf{r}_{perf}$.*

Proof. Since $valid(D, C) = \mathsf{true}$, the past-perfect round $D.\mathsf{r}_{\mathrm{perf}}$ is the value returned by *computePastPerfectRound*() of Algorithm 3. From Lemma 1 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 1, which we prove through a series of lemmas.

Theorem 1 (pod-core security)**.** *Assume that the network is partially synchronous with actual network delay δ, that β is the number of Byzantine replicas, γ 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 4, and the* identify() *function described in Algorithm 8, is a responsive secure pod (Definition 10) with Confirmation within $u = 2\delta$, Past-perfection within $w = \delta$ and β-accountable safety (Definition 2), except with negligible probability.*

Proof. The proof follows from Lemmas 3–7, presented and proven in the remainder of this section. □

Lemma 3 (Confirmation within u). *For the conditions stated in Theorem 1, Protocol 1 satisfies the* confirmation within *u property (Definition 10) for $u = 2\delta$.*

Proof. Assume an honest client c calls *write*(tx) at round r. It sends a message $\langle \mathit{WRITE}\ \mathsf{tx} \rangle$ to all replicas at round r (line 20). An honest replica receives this by round $\mathsf{r} + \delta$ and sends a $\langle \mathit{VOTE} \rangle$ message back to all connected clients (line 21). An honest client c' receives the vote by round $\mathsf{r} + 2\delta$. As are at least α honest (not Byzantine and not omission-faulty) replicas, c' receives at least α such votes, hence the condition in line 7 is satisfied and c' observes tx as confirmed. □

Lemma 4 (Past-perfection within w). *For the conditions stated in Theorem 1, Protocol 1 satisfies the* past-perfection within *w property (Definition 10) for $w = \delta$.*

Proof. Assume an honest client c at round r has view D^c_{r}. From Lemma 2, there exists some honest replica R_j, such that the most-recent timestamp $\mathsf{mrt}[R_j]$ that R_j has sent to 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 27), which arrives within δ rounds, and an honest client updates $\mathsf{mrt}[R_j]$ when it receives the heartbeat or vote message. Hence, 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 5 (Past-perfection safety). *For the conditions stated in Theorem 1, Protocol 1 satisfies the* past-perfection *safety property (Definition 10), 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 tx such that $(\mathsf{tx}, \mathsf{r}^1_{\min}, \mathsf{r}^1_{\max}, \mathsf{r}^1_{\text{conf}}) \notin D_1.\mathsf{T}$ and $(\mathsf{tx}, \mathsf{r}^2_{\min}, \mathsf{r}^2_{\max}, \mathsf{r}^2_{\text{conf}}) \in D_2.\mathsf{T}$ 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_{pp} contains a vote with timestamp $\mathsf{mrt}_i \geq D_1.\mathsf{r}_{\text{perf}}$. From Lemma 1 ($\mathsf{r}_{\text{perf}}$ is computed as the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of sorted 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 mrt_i (line 34 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 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 tx with timestamp 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 β 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 tx, and tx must appear in $D_1.\mathsf{T}$, a contradiction. □

Lemma 6 (Confirmation bounds). *For the conditions stated in Theorem 1, Protocol 1 satisfies the* confirmation bounds *safety property (Definition 10), except with negligible probability.*

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 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 1, C^1_{tx} can include at most $\lfloor \alpha/2 \rfloor - \beta$ votes with a timestamp for tx smaller than $\mathsf{r}^1_{\min}$. Allowing up to β replicas to equivocate, the adversary can obtain at most $\lfloor \alpha/2 \rfloor$ votes on 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 tx, the adversary must include in C^2_{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 1, C^1_{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_{tx} is at most $\alpha - \lfloor \alpha/2 \rfloor - 1$ (since β are malicious). If α 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 α 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. □

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
5:             continue
6:         end if
7:         for ⟨VOTE (tx_2, ts_2, sn_2, σ_2, R_2)⟩ ∈ T do
8:             if not Verify(pk_2, (tx_2, ts_2, sn_2), σ_2) then
9:                 continue
10:            end if
11:            if R_1 = R_2 and sn_1 = sn_2 and (tx_1 ≠ tx_2 or ts_1 ≠ ts_2) then
12:                R̃ ← R̃ ∪ {R_1}
13:            end if
14:        end for
15:    end for
16: end function
```

Lemma 7 (β-Accountable safety). *For the conditions stated in Theorem 1, Protocol 1 satisfies accountable safety (Definition 2) with resilience β, 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 10) is violated, there exists a partial transcript T, such that identify() on input T returns at least β replicas.
Proof: We resume the proof of Lemma 5. 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 $\mathbb{C}^1_{\mathsf{tx}}$ 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 $\mathbb{C}^2_{\mathsf{tx}}$ contains the following k'_i votes from R_i (possibly more, but we care for the votes up to transaction 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 tx_{i,k_i} in (1) cannot be tx, as $D_1.\mathsf{T}$ does not contain 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 identify(T) returns $\mathcal{R}_1 \cap \mathcal{R}_2$.

2. If the confirmation-bounds property (Definition 10) is violated, there exists a partial transcript T, such that Algorithm 8 on input T returns at least β replicas.
Proof: As in the proof of Lemma 6, 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 tx such that $(\mathsf{tx}, \mathsf{r}^1_{\min}, \mathsf{r}^1_{\max}, \mathsf{r}^1_{\text{conf}}) \in D_1.\mathsf{T}$, $(\mathsf{tx}, \mathsf{r}^2_{\min}, \mathsf{r}^2_{\max}, \mathsf{r}^2_{\text{conf}}) \in D_2.\mathsf{T}$, and $\mathsf{r}^1_{\min} > \mathsf{r}^2_{\text{conf}} \vee \mathsf{r}^1_{\max} < \mathsf{r}^2_{\text{conf}}$ Let $C_1 = (C^1_{\text{pp}}, \mathbb{C}^1_{\mathsf{tx}})$ and $C_2 = (C^2_{\text{pp}}, \mathbb{C}^2_{\mathsf{tx}})$, and $C^1_{\mathsf{tx}} = \mathbb{C}^1_{\mathsf{tx}}[\mathsf{tx}]$ and $C^2_{\mathsf{tx}} = \mathbb{C}^2_{\mathsf{tx}}[\mathsf{tx}]$.

Let's take the case $\mathsf{r}^1_{\min} > \mathsf{r}^2_{\text{conf}}$ first. From Lemma 1 (timestamps contains at least $n - \lfloor \alpha/2 \rfloor + \beta$ timestamps ts such that $\mathsf{ts} \geq \mathsf{r}_{\min}$), there is a set $\mathcal{R}_1$ with at least $n - \lfloor \alpha/2 \rfloor + \beta$ replicas R_i, from each of which $\mathbb{C}^1_{\mathsf{tx}}$ 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 $\mathsf{ts}_{i,m_i} \geq \mathsf{r}_{\min}$ and either $\mathsf{tx}_{i,m_i} = \mathsf{tx}$ (i.e., a vote from R_i on tx is included in C^1_{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 tx is not included in C^1_{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 $\mathsf{r}^2_{\text{conf}} < \mathsf{r}^1_{\min}$, certificate C^2_{tx} must contain timestamps smaller than $\mathsf{r}_{\min}$ from at least $\lfloor \alpha/2 \rfloor + 1$ replicas. Call this set $\mathcal{R}_2$. From each of these replicas, certificates $\mathbb{C}^2_{\mathsf{tx}}$ 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 tx, for which $\mathsf{ts}'_{i,m'_i} < \mathsf{r}_{\min}$.

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

3. The $\mathsf{identify}()$ 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 11 on Algorithm 8). An honest replica always increments nextsn after each vote it inserts to its log (line 23 on Algorithm 1), hence, the adversary can only construct such verifying votes by forging a signature under the public key of an honest replica, which happens with negligible probability. □

C Proofs for additional pod properties

In this section we prove the θ-timeliness property for pod, as stated in Appendix A.

Theorem 3 (θ-timeliness for honest transactions). *For the conditions stated in Theorem 1, Protocol 1 satisfies* θ-timeliness for honest transactions *(Definition 11), for $\theta = \delta$, except with negligible probability.*

Proof. Assume an honest client c calls *write*(tx) at round r. It sends a message $\langle \mathit{WRITE}\ \mathsf{tx} \rangle$ to all replicas at round r (line 20). An honest replica receives this by round $\mathsf{r} + \delta$ and assigns its current round, which lies in the interval $(\mathsf{r}, \mathsf{r} + \delta]$, as the timestamp (line 18).

1. Regarding r_{conf}, when a client calls *read*() (after the point in time when tx is confirmed, which happens after u rounds from the property of *confirmation within* u), it receives votes on tx from at least α replicas. All honest replicas have sent timestamps for tx in the interval $(r, \mathsf{r} + \delta]$. Since r_{conf} is computed as the median of α timestamps and $\alpha \geq 4\beta + 2\gamma + 1$,[7] we get $\lfloor \alpha/2 \rfloor > 2\beta + \gamma$, hence r_{conf} will be a timestamp returned by an honest (not Byzantine and not omitting messages) replica, or it will lie between timestamps returned by honest replicas. Hence, $\mathsf{r}_{\text{conf}} \in (r, \mathsf{r} + \delta]$.

[7] For this argument on r_{conf}, $\alpha \geq 2\beta + 2\gamma + 1$ would also be enough. The condition $\alpha \geq 4\beta + 2\gamma + 1$ is necessary in order for $\mathsf{r}_{\min}$ and $\mathsf{r}_{\max}$ of a confirmed transaction to be timestamps returned by honest replicas.

2. Regarding $\mathsf{r}_{\max}$, from Lemma 1 ($\mathsf{r}_{\max}$ is the timestamp at index $n - \alpha + \lfloor \alpha/2 \rfloor + \beta$ of timestamps), there areat least $\alpha - \lfloor \alpha/2 \rfloor - \beta + 1 > \lfloor \alpha/2 \rfloor - \beta$ timestamps in timestamps that bound $\mathsf{r}_{\max}$ from above. Since $\alpha \geq 4\beta + 2\gamma + 1$, we get that $\lfloor \alpha/2 \rfloor > 2\beta + \gamma$, hence $\lfloor \alpha/2 \rfloor - \beta > \beta + \gamma$, hence at least one of those timestamps that bound $\mathsf{r}_{\max}$ is returned by an honest replica, hence $\mathsf{r}_{\max} \in (r, \mathsf{r} + \delta]$.
3. Similarly, from Lemma 1 ($\mathsf{r}_{\min}$ is the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of timestamps), there are at least $\lfloor \alpha/2 \rfloor - \beta + 1$ timestamps in timestamps that bound $\mathsf{r}_{\min}$ from below, and, since $\alpha \geq 4\beta + 2\gamma + 1$, we get $\lfloor \alpha/2 \rfloor - \beta + 1 > \beta + \gamma$. Hence, $\mathsf{r}_{\min}$ is a timestamp returned by an honest replica, hence $\mathsf{r}_{\min} \in (r, \mathsf{r} + \delta]$ and $\mathsf{r}_{\max} - \mathsf{r}_{\min} < \theta$.

The proofs hold except with negligible probability, as the adversary can forge a signature under the public key of an honest replica with a negligible probability. □

D Security of bidset-core

In this section, we recall and prove Theorem 2.

Theorem 2 (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* δ-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. In Lemmas 8–11. The function for identifying a malicious sequencer is shown in Algorithm 9. □

Lemma 8 (Termination within W). *Under the assumptions of Theorem 2, Protocol 2 satisfies* termination within $W = t_0 + 3\Delta + \delta$.

Proof. The *result()* event is generated by an honest consumer when its exits the loop of lines 2–9 in Algorithm 7. At the latest, this happens when round $t_0 + 3\Delta$ becomes past-perfect (line 6 in Algorithm 7), which, from the *past-perfection within* δ property of pod, happens at round at most $t_0 + 3\Delta + \delta$, hence $W = t_0 + 3\Delta + \delta$. We remark that a sequencer (Algorithm 6) also terminates, because from the *past-perfection within* δ property of pod, the condition of line 3 becomes true by round $t_0 + \Delta + \delta$. □

Lemma 9 (Censorship resistance). *Under the assumptions of Theorem 2, Protocol 2 satisfies the* censorship resistance *property.*

Proof. Assume the sequencer is honest, and an honest bidder calls *submitBid(b)* at time t_0. We will show that $b \in B$. First, the pod view D^a_{r} of the sequencer a on the round r when it constructs B satisfies $D^a_{\mathsf{r}}.\mathsf{r}_{\text{perf}} > t_2$. Second, from the *confirmation within u* property of pod, the transaction containing b becomes confirmed, and from the *θ-timeliness* property of pod, it gets a confirmation round $\mathsf{r}_{\text{conf}} \leq t_0 + \theta$. For $\theta = \delta$, and since $\delta \leq \Delta$, we get that $\mathsf{r}_{\text{conf}} \leq t_2$. Hence, from the past-perfection safety property of pod we get that $b \in D^a_{\mathsf{r}}$, and, since the sequencer is honest, $b \in B$. □

Lemma 10 (Consistency). *Under the assumptions of Theorem 2, Protocol 2 satisfies the* consistency *property.*

Proof. Assume the sequencer is honest, and two honest consumers generate events *result*$(B_1, \cdot)$ and *result*$(B_2, \cdot)$. The condition in line 3 of Algorithm 6 becomes true in the view of sequencer by round $t_0 + \Delta + \delta$ (from the *past-perfection within* $w = \delta$ property of pod-core), hence the sequencer writes transaction $\langle \mathit{BIDS}\ (B, C_{\text{bid}}, \sigma) \rangle$ to pod by round $t_0 + \Delta + \delta$. This transaction gets assigned a confirmed round $\mathsf{r}_{\text{conf}} \leq t_0 + \Delta + 2\delta$ (from the *δ-timeliness* property of pod) and, by assumption of a synchronous network, $\mathsf{r}_{\text{conf}} \leq t_0 + 3\Delta$. The condition in line 4 of Algorithm 7 requires that a round $\mathsf{r}' > t_0 + 3\Delta$ becomes past perfect. As $\mathsf{r}' > \mathsf{r}_{\text{conf}}$, and by *past-perfection safety* of pod, the consumer observes the transaction as confirmed before r' becomes past-perfect, hence the condition in line 4 becomes true before the condition in line 6 and an honest consumer outputs *result*(B, C_{bid}). □

Lemma 11 (Accountable safety). *Under the assumptions of Theorem 2, and assuming that pod is an instance of* pod*-core, Protocol 2 achieves* accountable safety*, using the* identifySequencer() *function (Algorithm 9) to identify a malicious sequencer.*

Proof. Following Section 2.3, we show an identifySequencer(T) function (Algorithm 9), that, on input a partial transcript T outputs true when safety is violated due to misbehavior of the sequencer (i.e., it identifies the sequencer as malicious), and false if the sequencer is honest. We prove the theorem in three parts.

1. For violations of censorship-resistance:
Assume an honest bidder calls *submitBid(b)* at time t_0 and the network is synchronous. The transaction tx containing b becomes confirmed, and any honest party can observe $(\mathsf{tx}, \mathsf{r}_{\text{conf}}, \cdot, \cdot)$ and the corresponding transaction certificate C_{tx} in their view of the pod, as returned by pod-core. Assume b is censored, i.e., an event *result(B, C_{bid})* is output by an honest consumer, such that $b \notin B$. Let σ be the signature of the sequencer in the corresponding $\langle BIDS\ (B, C_{\text{bid}}, \sigma)\rangle$ message written on *pod*. We will show how the sequencer can be made accountable, using $(C_{\text{tx}}, B, C_{\text{bid}}, \sigma)$ as evidence T. In order for $(C_{\text{tx}}, B, C_{\text{bid}}, \sigma)$ to be valid evidence, the following must hold:

Requirement 1: The signature σ must be a valid signature, produced by the sequencer on message (B, C_{bid}), as per line 7 of the sequencer code (checked on line 3 of Algorithm 9 – we remind that notation '**require** P' returns false if P evaluates to false).

Requirement 2: C_{tx} must contain at least α votes (checked on line 18 of Algorithm 9), on the same transaction tx^* (checked on line 23), signed by a *pod* replica (checked on line 24).

If any of these requirements are not met, T does not constitute valid evidence and the function exits. Otherwise, let $\mathsf{r}^*_{\text{conf}}$ be the median of all votes in C_{tx}. The function makes the following checks, and if any of them fails, then the sequencer is accountable.

Check 1: Verify whether the votes that the sequencer has included in C_{bid} are valid, obtained from the replicas that run *pod* (lines 5-11). If this is not the case, the sequencer has misbehaved.

Check 2: Compute the r_{perf} from the timestamps found in the votes in C_{bid} (lines 12-17). This r_{perf} must be larger than $t_0 + \Delta$, as per line 3 of Algorithm 6.

Check 3: If $\mathsf{r}^*_{\text{conf}} \leq t_0 + \Delta$ but tx^* is not in the bag, the sequencer has misbehaved.

2. For violations of consistency:
The consistency property can be violated if the sequencer writes two transactions $\langle BIDS\ (B_1, \cdot, \cdot)\rangle$ and $\langle BIDS\ (B_2, \cdot, \cdot)\rangle$ to *pod*, such that $B_1 \neq B_2$, in which case B_1 and B_2 identify the sequencer. As this is a simpler case, we do not show it in Algorithm 9.

3. A honest sequencer cannot be framed:
Finally, we show that an honest sequencer cannot be framed. If the sequencer has followed Algorithm 6, then C_{bid} will contain valid votes, hence *Check 1* will pass. Moreover, an honest sequencer waits until the past-perfect round returned by the pod is larger than $t_0 + \Delta$, hence *Check 2* will pass. Regarding *Check 3*, observe that for identifySequencer() to compute $\mathsf{r}^*_{\text{conf}} \leq t_0 + \Delta$, C_{tx} must contain at least $\lfloor \alpha/2 \rfloor + 1$ votes on tx^* with a timestamp smaller or equal than $t_0 + \Delta$, and at least $\lfloor \alpha/2 \rfloor + 1 - \beta$ of them must be from honest replicas. Call this set $\mathcal{R}'$. The honest sequencer, in order to output a past-perfect round greater than $t_0 + \Delta$, must have received timestamps greater than $t_0 + \Delta$ from at least $n - \lfloor a/2 \rfloor + \beta$ replicas (from Lemma 1). By counting arguments, at least one of these timestamps must be from one of the honest replicas in $\mathcal{R}'$, and, since honest replicas do not omit transactions, that replica will have sent a vote on tx^* to the sequencer. Hence, the honest sequencer will include tx^* in B. □

Algorithm 9 The identifySequencer() function for Protocol 2, instantiated with an instance of pod-core (Protocol 1) as *pod*, run by a set of replicas $\mathcal{R} = \{R_1, \cdots, R_n\}$ with public keys $\{\mathsf{pk}1, \cdots, \mathsf{pk}_n\}$, and using the *median*() operation defined by Protocol 1. It identifies a malicious sequencer, whose public key is pk_a, by returning true.

```
1: function identify(T)
2:     (C_tx, B, C_bid, σ) ← T
3:     require Verify(pk_a, (B, C_bid), σ)
4:     timestamps ← [ ]
5:     for vote ∈ C_bid do
6:         (tx, ts, sn, σ, R_j) ← vote
7:         if Verify(pk_j, (tx, ts, sn), σ = 0) then
8:             return true
9:         end if
10:        timestamps ← timestamps ‖ ts
11:    end for
12:    sort timestamps in increasing order
13:    timestamps ← [0, β times ⋯, 0] ‖ timestamps
14:    r_perf ← median(timestamps [:α ])
15:    if r_perf ≤ t_0 + Δ then
16:        return true
17:    end if

18:    require |C_tx| ≥ α
19:    tx* ← C_tx[0].tx
20:    timestamps ← [ ]
21:    for vote ∈ C_tx do
22:        (tx, ts, sn, σ, R_j) ← vote
23:        require tx = tx*
24:        require Verify(pk_j, (tx, ts, sn), σ)
25:        timestamps ← timestamps ‖ ts
26:    end for
27:    r*_conf ← median(timestamps)
28:    if r*_conf ≤ t_0 + Δ and tx* ∉ B then
29:        return true
30:    end if
31: end function
```