

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

Orestis Alpos¹, 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 one round trip time, *i.e.* requiring only one round for writing a new transaction and one round for reading it. To accomplish this, we first eliminate inter-replica communication. Instead, clients send transactions directly to all replicas, which independently process transactions and append them to local logs. Replicas assign a timestamp and a sequence number to each transaction in their logs, allowing clients to extract valuable metadata about the transactions and the system state. Later on, clients retrieve these logs and extract transactions (and associated metadata) from them.

Necessarily, this construction achieves weaker properties than a total-order broadcast protocol, due to existing lower bounds. Our work models the primitive of **pod** and defines its security properties. We then show **pod-core**, a protocol that satisfies properties such as transaction confirmation within 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-roundtrip 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.

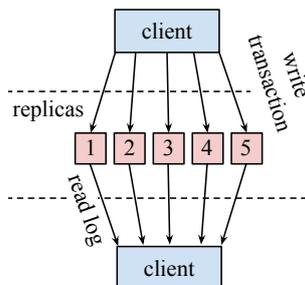


Fig. 1: **pod-core**'s simple architecture. A writing client (top) sends a transaction to all replicas (middle). Each replica appends it to its own log and transmits it to the reading client (bottom).

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 latency, with each confirmed transaction assigned a *confirmation timestamp*; we say that the transaction becomes *confirmed* at the time indicated by the confirmation timestamp.
- Censorship resistance when facing up to β Byzantine or 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 any safety violations, *i.e.*, identifying replicas that cause such violations.

In particular, our Protocol **pod-core** presented in Section 4 realizes the notion of **pod** with the properties above under two adversarial models:

- (Security against Byzantine faults) β Byzantine replicas (and no omission-faulty replicas) out of a total of $n \geq 5\beta + 1$ replicas.
- (Security against omission faults) γ Byzantine replicas (and no Byzantine replicas) out of a total of $n \geq 3\beta + 1$ replicas.

Our Protocol **pod-core** requires no expensive cryptographic primitives or setup beyond digital signatures and a PKI registering replicas' public keys. We showcase Protocol **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 $r_{\min} \geq 0$ (minimum round), $r_{\max} \leq \infty$ (maximum round) and r_{conf} (confirmed round). It may be that initially $r_{\text{conf}} = \perp$ but later becomes $r_{\min} \leq r_{\text{conf}} \leq r_{\max}$, when a transaction is *confirmed*. While each party reads different values $r_{\min}, r_{\max}, r_{\text{conf}}$ 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 $D = (\mathbb{T}, r_{\text{perf}}, C_{\text{pp}})$, where \mathbb{T} is set of transactions with their associated timestamps, r_{perf} is a *past-perfect* round and C_{pp} is auxiliary data. The *past-perfection* safety property guarantees that \mathbb{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.

For each transaction `tx` in \mathbb{T} , the reader obtains associated timestamps $r_{\min}, r_{\max}, r_{\text{conf}}$ and auxiliary data C_{tx} , which may evolve. 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 $r_{\text{conf}} \neq \perp$. Moreover, `pod` guarantees *confirmation bounds*, ensuring that the values r_{\min} and r_{\max} for `tx`, read by an honest party, determine the range of the r_{conf} value for `tx` that can ever be read by another honest party, i.e., $r_{\min} \leq r_{\text{conf}} \leq r_{\max}$.

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` via Protocol `pod-core` only requires clients to send `tx` to the replicas, who each assign a timestamp `ts` (their current time) and a *sequence number* to `tx` and return a signature on $(\text{tx}, \text{ts}, \text{sn})$. When reading the `pod`, the client simply requests each replica's log of transactions, validates the responses, and determines r_{\min} and 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.

We define Protocol `pod-core` such that clients may specify their own trust assumptions, expecting up to β Byzantine *and* at the same time up to γ additional omission-faulty replicas. However, although we prove our results for executions with only Byzantine and only omission faults, we conjecture that Protocol `pod-core` realizes the `pod` notion for a continuum between these two cases. We leave the security analysis of our protocol under a continuum of mixed Byzantine and omission faults to the full version of this work.

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] may also be easily realized over `pod`.

1.3 Related work

Reducing latency. Many previous works have lowered the latency of ordering transactions. Hot-Stuff [25] 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 it 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 [10] 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 work, Tusk [8] achieves an 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, 7, 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 $L_1 \parallel L_2$.

2.1 Execution Model

Parties. We consider n replicas $R = \{R_1, \dots, R_n\}$ 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. In this work we use a combination of two adversarial models, the *Byzantine* and the *omission* models. In this case, the adversary has access to the internal state and secret keys of all corrupted parties. In the *Byzantine* model, corrupted replicas are malicious and may deviate arbitrarily from the protocol. We denote by $\beta \in [0, n]$ the number of Byzantine replicas in an execution. 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. 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.

Modeling time. Time proceeds in discrete *rounds* (akin to timestamps). The environment is constrained to work in a prespecified, polynomial number of rounds. The environment begins by spawning all honest parties as well as the adversary, and initializing the current round to 0. Upon spawning the honest parties, the environment must call the *init()* function of each party in a round robin fashion. It then sets the *network inbox* of each honest party to the empty sequence. It then proceeds to iterate over all rounds, one by one, incrementing the round variable by one during each iteration, starting with round 1 in the first iteration. For each round it iterates over, it invokes all honest parties one by one in a round robin fashion. In each such honest party invocation, the environment looks at the honest party's *network inbox* and delivers any messages that are present by invoking the respective *upon* functionalities specified by the honest protocol, parametrized by the message and the sender, once per message in the inbox. During the invocation of such *upon* functionalities, the honest party may send a message to another party by invoking the *send()* function (with the message and the recipient as arguments), which places the message on the *network outbox* of the sending honest party. During the *upon* invocation, the honest party may also invoke the *round()* function, which returns the current round number. The *round()* function works by asking the environment for the current round, and the environment is compelled to answer truthfully. This means that *the clocks of honest parties are synchronized*. At the end of the round, after all honest parties have been invoked, the environment cleans the *network inboxes* of all honest parties in preparation for the next round. Subsequently, the adversary is invoked (i.e., she is a rushing adversary), by handing her all the network outboxes of all honest parties. The adversary decides the network inboxes of all honest parties for the next round. The adversary can include, in these network inboxes newly prepared for the next round, messages that were present in network outboxes of honest parties in previous rounds. The adversary can also reorder messages (potentially providing a different reordering for each recipient), add new messages of her own (potentially different for each recipient), as well as choose not to include messages in the network inboxes, subject to the constraints in the next paragraph.

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*. The environment is constrained to deliver all honest network messages within δ . Namely, for each message m sent by an honest party at round r , the environment observes and remembers the inboxes (decided by the adversary) of all honest parties during rounds $r+1, \dots, r+\delta$. If none of these inboxes contains m , the environment appends m to the network inbox of the receiving party prior to invoking it at round $r+\delta$, thereby forcing the inclusion of this message. In the *synchronous* model, Δ is *known* to all parties (namely, the honest code is parametrized by Δ). In the *partially synchronous* model, Δ is *unknown* but still finite, i.e., all messages are eventually delivered. We now define responsive protocols, which is the case of our main protocol *pod-core*.

Definition 1 (Responsive Protocol (Informal)). *A protocol is responsive if it does not rely on knowledge of Δ and its liveness guarantees depend only on the actual network delay δ .*

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:

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

2.2 Accountable safety

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

Definition 2 (Transcript and partial transcript). We define a 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 3 (β -Accountable safety). A protocol satisfies accountable safety with resilience β if its interface contains a function $\text{identify}(T) \rightarrow \tilde{R}$, 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 $\text{identify}(T) \rightarrow \tilde{R}$ and $|\tilde{R}| > \beta$.

No-framing: For any partial transcript T produced during an execution of the protocol, the output of $\text{identify}(T)$ 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 $\text{identify}()$, 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 replicas maintain transactions with an evolving partial order, which can be *read* and *written* by clients. We first define basic data structures and the interface of a *pod*. We then introduce the basic definition of a secure *pod* and its security properties, as well as some additional properties that it may satisfy, which we will later use for our applications. In particular, we define the notions of *timeliness* and *monotonicity* for a secure *pod*.

Definition 4 (Transaction and associated values). A transaction $\text{tx} \in \{0, 1\}^*$ is associated with values $r_{\min}, r_{\max}, r_{\text{conf}}$, and C_{tx} , which change during the execution of a *pod* protocol. We call $r_{\min} \in \mathbb{N}$ the minimum round, $r_{\max} \in \mathbb{N} \cup \{\infty\}$ the maximum round, $r_{\text{conf}} \in \mathbb{N} \cup \{\perp\}$ the confirmed round, and $C_{\text{tx}} \in \{0, 1\}^*$ contains some auxiliary data. We denote by $r_{\max} = \infty$ an unbounded maximum round and by $r_{\text{conf}} = \perp$ an undefined confirmed round. For transaction tx , denote its associated values by $\text{tx}.r_{\min}, \text{tx}.r_{\max}, \text{tx}.r_{\text{conf}}$, and $\text{tx}.C_{\text{tx}}$.

In *pod*-core, C_{tx} will contain digital signatures used to achieve accountability.

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

Definition 6 (Transaction set). A transaction set T is a set of tuples $(\text{tx}, r_{\min}, r_{\max}, r_{\text{conf}}, C_{\text{tx}})$. We say that a transaction tx appears in T when $\exists (\text{tx}', r_{\min}, r_{\max}, r_{\text{conf}}, C_{\text{tx}}) \in T$ such that $\text{tx}' = \text{tx}$. Conversely, we say that a transaction tx does not appear in T when this condition is not satisfied.

Definition 7 (Pod data structure). A *pod* data structure D is a triple $(T, r_{\text{perf}}, C_{\text{pp}})$, where T is a transaction set, r_{perf} is a round number, called the past-perfect round, and $C_{\text{pp}} \in \{0, 1\}^*$ contains auxiliary data.

In *pod*-core, C_{pp} will also contain signatures used to achieve accountability. Our construction will allow for deriving C_{pp} from all certificates C_{tx} in T , but we define C_{pp} explicitly for clarity and generality. Moreover, r_{perf} will imply a completeness property on T , defined by the past-perfection safety property of *pod*. We remark that transactions in T may be confirmed on unconfirmed.

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

- $\text{write}(\text{tx})$: It writes a transaction tx to the pod.
- $\text{read}() \rightarrow D$: It outputs a pod data structure $D = (T, r_{\text{perf}}, C_{\text{pp}})$.

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

Definition 9 (View of the pod). We call view of the pod the output of $\text{read}()$, where $\text{read}()$ is invoked by client c and the output is produced at round r , and denote it as D_r^c . We remark that r denotes the round when $\text{read}()$ outputs, as the client may have invoked it at an earlier round. We denote the components of a view as $D_r^c = (D_r^c.T, D_r^c.r_{\text{perf}}, D_r^c.C_{\text{pp}})$. We write $\text{tx} \in D_r^c$ if tx appears in $D_r^c.T$.

Definition 10 (Secure pod). A protocol is a secure pod if it implements the pod interface of Definition 8 and satisfies the following properties.

- (Liveness) Confirmation within u :** Transactions become confirmed after at most u rounds. That is, if an honest client c writes a transaction tx at round r , then for any honest client c' (including $c = c'$) it holds that $\text{tx} \in D_{r+u}^{c'}$ and $\text{tx}.r_{\text{conf}} \neq \perp$.
- (Liveness) Past-perfection within w :** Rounds become past-perfect after at most w rounds. That is, for any honest client c and round $r \geq w$, it holds that $D_r^c.r_{\text{perf}} \geq r - w$.
- (Safety) Past-perfection:** Let c be an honest client with view D_r^c at round r . Then, for any transaction $\text{tx} \in \{0, 1\}^*$ that becomes confirmed (in some view $D_{r'}^{c'}$ of an honest client c' at round $r' \geq 0$) with confirmed round r_{conf} , it holds that, if $r_{\text{conf}} < D_r^c.r_{\text{perf}}$, then $\text{tx} \in D_r^c$.
- (Safety) Confirmation bounds:** For any honest clients c_1, c_2 , if c_1 observes tx with r_{\min} and r_{\max} and c_2 observes tx as confirmed with confirmation round r'_{conf} , then $r_{\min} \leq r'_{\text{conf}} \leq r_{\max}$.

The confirmation bounds property gives $r_{\min} \leq r'_{\text{conf}} \leq r_{\max}$, for r'_{conf} computed by honest client c_2 and r_{\min}, r_{\max} computed by honest client c_1 , but it does not guarantee anything about the values of r_{\min} and r_{\max} (for example, it could trivially be $r_{\min} = 0$ and $r_{\max} = \infty$). 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 c_1, c_2 , if c_1 writes transaction tx in round r , then c_2 computes $r_{\min}, r_{\max}, r_{\text{conf}}$ for tx such that:

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

Moreover, a pod protocol allows the values $r_{\min}, r_{\max}, r_{\text{conf}}$ 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 3 and 3.

4 Protocol pod-core

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

Definition 12 (Vote). A vote is a tuple $(\text{tx}, \text{ts}, \text{sn}, \sigma, R)$, 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 $m = (\text{tx}, \text{ts}, \text{sn})$ 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.

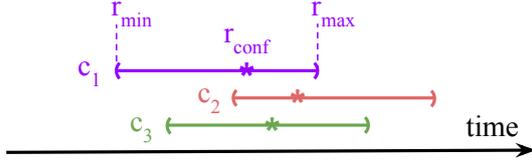


Fig. 2: The same transaction in the view of three different pod clients. Each client assigns it a minimum round r_{\min} and a maximum round r_{\max} . If it gets confirmed, the confirmation round r_{conf} will be between these two values. The r_{conf} that each client locally computes respects the bounds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

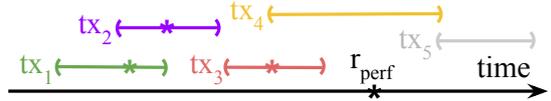


Fig. 3: A possible view of a single pod client. Transactions $\text{tx}_1, \text{tx}_2, \text{tx}_3$ are confirmed, tx_4 is not yet confirmed. A client also derives a past-perfect round r_{perf} . No transaction other than $\text{tx}_1, \text{tx}_2, \text{tx}_3, \text{tx}_4$ may obtain $r_{\text{conf}} \leq r_{\text{perf}}$. There may exist tx_5 for which the client has not received votes, but tx_5 cannot obtain $r_{\text{conf}} \leq r_{\text{perf}}$.

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, votes, and associated values) across all invocations of $\text{write}()$ and $\text{read}()$.

Pseudocode notation. For a timestamp ts , notation $\text{ts.getVoteMsg}()$ denotes the vote message from some replica through which a client obtained timestamp ts . We abstract away the logic of how $\text{getVoteMsg}()$ is implemented. Notation $x : a \in A \rightarrow b \in B$ 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 $x.\text{keys}()$ and $x.\text{values}()$ 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, 3 and 4 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 $\alpha = n - \beta - \gamma$.*

Remark 5. Looking forward, we only analyze the cases where either β or γ are non-zero, *i.e.* we consider only the cases where all corrupted replicas are assumed to be either Byzantine or omission-faulty but not the cases where there are mixed Byzantine and omission-fault corruptions. However, we conjecture that Protocol **pod-core** is also a secure **pod** under a mixed adversarial model with a continuum of both Byzantine and omission-fault corruptions. A security analysis of Protocol **pod-core** in this mixed adversarial model will appear in the full version.

Replica code. The state of a replica (lines 1–4 of Algorithm 1) contains, among others, the transaction log replicaLog , which is implemented as a sequence of votes $(\text{tx}, \text{ts}, \text{sn}, \sigma, R_i)$ 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. A functionality $\text{round}()$ allows the replica to determine the current round number. When the replica receives a $\langle \text{CONNECT} \rangle$ message from a client c , it appends c to its set of connected clients and sends to c all entries in replicaLog (lines 8–13).

When it receives $\langle \text{WRITE tx} \rangle$, 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 $(\text{tx}, \text{ts}, \text{sn})$ (line 19). 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 $(\text{tx}, \text{ts}, \text{sn}, \sigma)$ to replicaLog , and sends it via a $\langle \text{VOTE}(\text{tx}, \text{ts}, \text{sn}, \sigma, R_i) \rangle$ message to all connected clients (line 22).

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

```

1:  $\mathcal{C}$  ▷ The set of all connected clients
2: nextsn ▷ The next sequence number to assign to votes
3: replicaLog ▷ The transaction log of the replica
4: round() ▷ A function that returns the wall time of the replica

5: upon event init() do
6:   nextsn  $\leftarrow 0$ ;  $\mathcal{C} \leftarrow \emptyset$ ; replicaLog  $\leftarrow []$ 
7: end upon

8: upon receive  $\langle CONNECT \rangle$  from client  $c$  do
9:    $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$ 
10:  for  $(tx, ts, sn, \sigma) \in \text{replicaLog}$  do
11:    Send  $\langle VOTE(tx, ts, sn, \sigma, R_i) \rangle$  to  $c$ 
12:  end for
13: end upon

14: upon receive  $\langle WRITE tx \rangle$  from a client do
15:  if replicaLog[tx]  $\neq \perp$  then return ▷ Ignore duplicate transactions
16:  doVote(tx)
17: end upon

18: function DOVOTE(tx)
19:  ts  $\leftarrow \text{round}()$ ; sn  $\leftarrow \text{nextsn}$ ;  $\sigma \leftarrow \text{Sign}(sk, (tx, ts, sn))$ 
20:  replicaLog  $\leftarrow \text{replicaLog} \parallel (tx, ts, sn, \sigma)$ 
21:  for  $c \in \mathcal{C}$  do
22:    Send  $\langle VOTE(tx, ts, sn, \sigma, R_i) \rangle$  to  $c$ 
23:  end for
24:  nextsn  $\leftarrow \text{nextsn} + 1$ 
25: end function

26: upon end round do ▷ Executed at the end of each round
27:  doVote(HEARTBEAT)
28: end upon

```

Client initialization. The state of a client is shown in Algorithm 2 in lines 2–6. The state contains mrt , $nextsn$, $tsps$, and D . Variable $tsps$ is a map from transactions tx to a map from replicas R to timestamps ts . The state gets initialized in lines 7–14. 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.

Receiving votes. A client maintains a connection to each replica and receives votes through $\langle VOTE(tx, ts, sn, \sigma, R_j) \rangle$ messages (lines 15–24). When a vote is received from replica R_j , the client first verifies the signature σ under R_j 's public key (line 16). 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 17). 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 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 19). Second, the replica must have not previously sent a different timestamp for tx (line 22), except if tx is a HEARTBEAT (as checked on line 21). If both checks pass, the client updates $mrt[j]$ (line 20) and $tsps[tx][R_j]$ (line 23) with ts . 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 $\langle VOTE(tx, ts, sn, \sigma, R_j) \rangle$ and the vote it received when it updated $mrt[R_j]$ to prove that R_j has misbehaved. In the second case, it can use $\langle VOTE(tx, ts, sn, \sigma, R_j) \rangle$ and the previous vote it has received for tx . The `identify()` function we show in Algorithm 8 can detect such misbe-

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

```
1: State:
2:  $\mathcal{R} = \{R_1, \dots, R_n\}; \{\mathbf{pk}_1, \dots, \mathbf{pk}_n\}$  ▷ All replicas and their public keys
3:  $\text{mrt} : R \rightarrow \text{ts}$  ▷ The most recent timestamp returned by each replica
4:  $\text{nextsn} : R \rightarrow \text{sn}$  ▷ The next sequence number expected by each replica
5:  $\text{tsps} : \text{tx} \rightarrow (R \rightarrow \text{ts})$  ▷ Timestamp received for each tx from each replica
6:  $D = (\mathbb{T}, r_{\text{perf}}, C_{\text{pp}})$  ▷ The pod observed by the client so far

7: upon event  $\text{init}(R_1, \dots, R_n, k_1, \dots, k_n)$  do
8:    $\mathcal{R} \leftarrow \{R_1, \dots, R_n\}; (\mathbf{pk}_1, \dots, \mathbf{pk}_n) \leftarrow (k_1, \dots, k_n)$ 
9:    $\text{tsps} \leftarrow \emptyset; D = (\emptyset, 0, [])$ 
10:  for  $R_j \in \mathcal{R}$  do
11:     $\text{mrt}[R_j] \leftarrow 0; \text{nextsn}[R_j] = -1$ 
12:    Send  $\langle \text{CONNECT} \rangle$  to  $R_j$ 
13:  end for
14: end upon

15: upon receive  $\langle \text{VOTE}(\text{tx}, \text{ts}, \text{sn}, \sigma, R_j) \rangle$  do ▷ Received a vote from replica  $R_j$ 
16:  if not  $\text{Verify}(\mathbf{pk}_j, (\text{tx}, \text{ts}, \text{sn}), \sigma)$  then return ▷ Invalid vote
17:  if  $\text{sn} \neq \text{nextsn}[R_j]$  then return ▷ Vote cannot be processed yet
18:   $\text{nextsn}[R_j] \leftarrow \text{nextsn}[R_j] + 1$ 
19:  if  $\text{ts} < \text{mrt}[R[j]]$  then return ▷  $R_j$  sent old timestamp
20:   $\text{mrt}[R[j]] \leftarrow \text{ts}$ 
21:  if  $\text{tx} = \text{HEARTBEAT}$  then return ▷ Do not update  $\text{tsps}$  for HEARTBEAT
22:  if  $\text{tsps}[\text{tx}][R_j] \neq \perp$  and  $\text{tsps}[\text{tx}][R_j] \neq \text{ts}$  then return ▷ Duplicate timestamp
23:   $\text{tsps}[\text{tx}][R_j] \leftarrow \text{ts}$ 
24: end upon
```

havior. However, in this paper we formalize accountability conditioned on safety being violated (Definition 3), hence we do not further explore this.

Writing to and reading from pod. Protocol pod-core implements functions $\text{write}(\text{tx})$ and $\text{read}()$ (Definition 8) functions as shown in Algorithm 3. In order to write a transaction tx , a client sends $\langle \text{WRITE tx} \rangle$ to each replica (lines 1–5). Since the construction is stateful and streaming, the client state contains at all times the latest view the client has of the pod, $\text{read}()$ operates on the local state (lines 6–28). It returns a pod D (line 27) with a transaction set \mathbb{T} , containing the transactions the client has received so far (line 8) and their associated values, a past-perfect round r_{perf} , and auxiliary data C_{pp} . Note that $\text{tsps.keys}()$ on line 8 returns all entries in tsps .

The client uses the $\text{minPossibleTs}()$ and $\text{maxPossibleTs}()$ functions to compute r_{min} (line 9) and r_{max} (line 10). They respectively return the minimum and maximum round number that *any client* can ever confirm tx with. For r_{perf} (line 22) the client uses $\text{minPossibleTsForNewTx}()$, which returns the earliest round that *any client* can ever assign to a transaction *not yet seen* by the client. C_{pp} contains the vote on the most recent timestamp $\text{mrt}[R_j]$ received from each R_j .

A transaction becomes confirmed (i.e., is assigned a confirmation round $r_{\text{conf}} \neq \perp$) when the client receives α votes for tx from different replicas (line 12). Before it becomes confirmed, tx has $r_{\text{conf}} = \perp$ and $C_{\text{tx}} = []$ (line 11). When confirmed, r_{conf} is the median of all received timestamps (line 18), and the auxiliary data C_{tx} contains all the received votes on tx (line 16).

Computing the associated values and the past-perfect round. Function $\text{minPossibleTs}()$ in Algorithm 4 fills a missing vote from replica R_j with $\text{mrt}[R_j]$ (line 7), the minimum timestamp that can ever be accepted from R_j (smaller values will not pass the check in line 19 of Algorithm 3). Moreover, it prepends β times the 0 value (line 11), 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 $\beta > 0$. Similarly, function $\text{maxPossibleTs}()$ in Algorithm 4 fills a missing vote with ∞ (line 20) and appends β times the ∞ value (line 24), the worst-case timestamp that Byzantine replicas may send to other clients. Finally, $\text{minPossibleTsForNewTx}()$ is similar to $\text{minPossibleTs}()$ but considers timestamps mrt the minimum possible for future transactions.

Algorithm 3 Protocol pod-core: Code for a client, part 2

```
1: function WRITE(tx)
2:   for  $R_j \in \mathcal{R}$  do
3:     Send  $\langle \text{WRITE tx} \rangle$  to  $R_j$ 
4:   end for
5: end function

6: function READ()
7:    $\mathsf{T} \leftarrow \emptyset$ ;  $C_{\text{pp}} \leftarrow []$ 
8:   for  $\text{tx} \in \text{tsps.keys}()$  do ▷ Loop over all received transactions
9:      $r_{\text{min}} \leftarrow \text{minPossibleTs}(\text{tx})$ 
10:     $r_{\text{max}} \leftarrow \text{maxPossibleTs}(\text{tx})$ 
11:     $r_{\text{conf}} \leftarrow \perp$ ;  $C_{\text{tx}} \leftarrow []$ ;  $\text{timestamps} = []$ 
12:    if  $|\text{tsps}[\text{tx}].\text{keys}()| \geq \alpha$  then
13:      for  $R_j \in \text{tsps}[\text{tx}].\text{keys}()$  do
14:         $\text{ts} \leftarrow \text{tsps}[\text{tx}][R_j]$ 
15:         $\text{timestamps} \leftarrow \text{timestamps} \parallel \text{ts}$ 
16:         $C_{\text{tx}} \leftarrow C_{\text{tx}} \parallel \text{ts.getVoteMsg}()$ 
17:      end for
18:       $r_{\text{conf}} \leftarrow \text{median}(\text{timestamps})$ 
19:    end if
20:     $\mathsf{T} \leftarrow \mathsf{T} \cup \{(\text{tx}, r_{\text{min}}, r_{\text{max}}, r_{\text{conf}}, C_{\text{tx}})\}$ 
21:  end for
22:   $r_{\text{perf}} \leftarrow \text{minPossibleTsForNewTx}()$ 
23:  for  $R_j \in \mathcal{R}$  do
24:     $C_{\text{pp}} \leftarrow C_{\text{pp}} \parallel \text{mrt}[R_j].\text{getVoteMsg}()$ 
25:  end for
26:   $D \leftarrow (\mathsf{T}, r_{\text{perf}}, C_{\text{pp}})$ 
27:  return  $D$ 
28: end function
```

Heartbeat messages. Clients update their most-recent timestamp $\text{mrt}[R_j]$ every time they receive a vote from replica R_j (line 20 in Algorithm 2). However, in case R_j has not written any transaction in round r (simply because no client called $\text{write}()$ in round r), R_j will advance its round number, but clients will not advance $\text{mrt}[R_j]$. We solve this by having replicas send a vote on a dummy HEARTBEAT transaction the end of each round (lines 26–28). An obvious practical implementation 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 15 in Algorithm 2), except that we do not check for duplicate HEARTBEAT votes and do not update any associated values for the heartbeat transaction (see line 21 in Algorithm 2).

Theorem 1 (pod-core security under Byzantine faults). *Assuming that the network is asynchronous with actual network delay δ (and unknown delay upper bound), β is the number of Byzantine replicas, $\gamma = 0$ is the number of omission-faulty replicas and $n \geq 5\beta + 1$ is the total number of replicas. Protocol pod-core (Protocol 1) instantiated with a EUF-CMA secure signature scheme is a responsive secure pod (Definition 10) with Confirmation within $u = 2\delta$, Past-perfection within $w = \delta$ and β -accountable safety (Definition 3) with the identify() function described in Algorithm 8, except with negligible probability.*

Proof. Shown in Appendix B. □

Theorem 2 (pod-core security under Omission faults). *Assuming that the network is asynchronous with actual network delay δ (and unknown delay upper bound), $\beta = 0$ is the number of Byzantine replicas, γ is the number of omission-faulty replicas and $n \geq 3\gamma + 1$ is the total number of replicas. Protocol pod-core (Protocol 1) instantiated with a EUF-CMA secure signature scheme is a responsive secure pod (Definition 10) with Confirmation within $u = 2\delta$ and Past-perfection within $w = \delta$, except with negligible probability.*

Proof. Shown in Appendix C. □

Algorithm 4 Protocol `pod-core`: Client code, part 3 (computing associated 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 MINPOSSIBLETs(tx)
2:   timestamps  $\leftarrow$  []
3:   for  $R_j \in \mathcal{R}$  do
4:     if  $\text{tsps}[\text{tx}][R_j] \neq \perp$  then
5:       timestamps  $\leftarrow$  timestamps  $\parallel$  [ $\text{tsps}[\text{tx}][R_j]$ ]
6:     else
7:       timestamps  $\leftarrow$  timestamps  $\parallel$  [ $\text{mrt}[R_j]$ ]
8:     end if
9:   end for
10:  sort timestamps in increasing order
11:  timestamps  $\leftarrow$   $[0, \overset{\beta \text{ times}}{\dots}, 0]$   $\parallel$  timestamps  $\triangleright$  omitted altogether if  $\beta = 0$ 
12:  return median(timestamps[:  $\alpha$ ])
13: end function

14: function MAXPOSSIBLETs(tx)
15:   timestamps  $\leftarrow$  []
16:   for  $R_j \in \mathcal{R}$  do
17:     if  $\text{tsps}[\text{tx}][R_j] \neq \perp$  then
18:       timestamps  $\leftarrow$  timestamps  $\parallel$  [ $\text{tsps}[\text{tx}][R_j]$ ]
19:     else
20:       timestamps  $\leftarrow$  timestamps  $\parallel$  [ $\infty$ ]
21:     end if
22:   end for
23:  sort timestamps in increasing order
24:  timestamps  $\leftarrow$  timestamps  $\parallel$  [ $\infty, \overset{\beta \text{ times}}{\dots}, \infty$ ]  $\triangleright$  omitted altogether if  $\beta = 0$ 
25:  return median(timestamps[- $\alpha$  :])
26: end function

27: function MINPOSSIBLETsFORNEWTx()
28:   timestamps  $\leftarrow$  mrt
29:   sort timestamps in increasing order
30:   timestamps  $\leftarrow$   $[0, \overset{\beta \text{ times}}{\dots}, 0]$   $\parallel$  timestamps  $\triangleright$  omitted altogether if  $\beta = 0$ 
31:   return median(timestamps[:  $\alpha$ ])
32: end function

33: function MEDIAN(Y)
34:   return  $Y[\lfloor |Y|/2 \rfloor]$ 
35: end function

```

Remark 6. Notice that Theorem 2 does not capture β -accountable safety (Definition 3) because there are no safety violations when all corrupted replicas are only omission-faulty.

5 Evaluation

To validate our theoretical results regarding optimal latency in Protocol `pod-core`, we implement⁵ prototype replicas and clients in Rust 1.85. We benchmark this prototype by measuring the end-to-end confirmation latency of a transaction from the moment it is written by client until it is confirmed by another client in a different continent, both interacting with replicas distributed around the world. We present the results in Figure 4.

The experiment involves 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 connected 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.

⁵ Our prototype implementation is available at <https://github.com/commonprefix/pod-experiments>

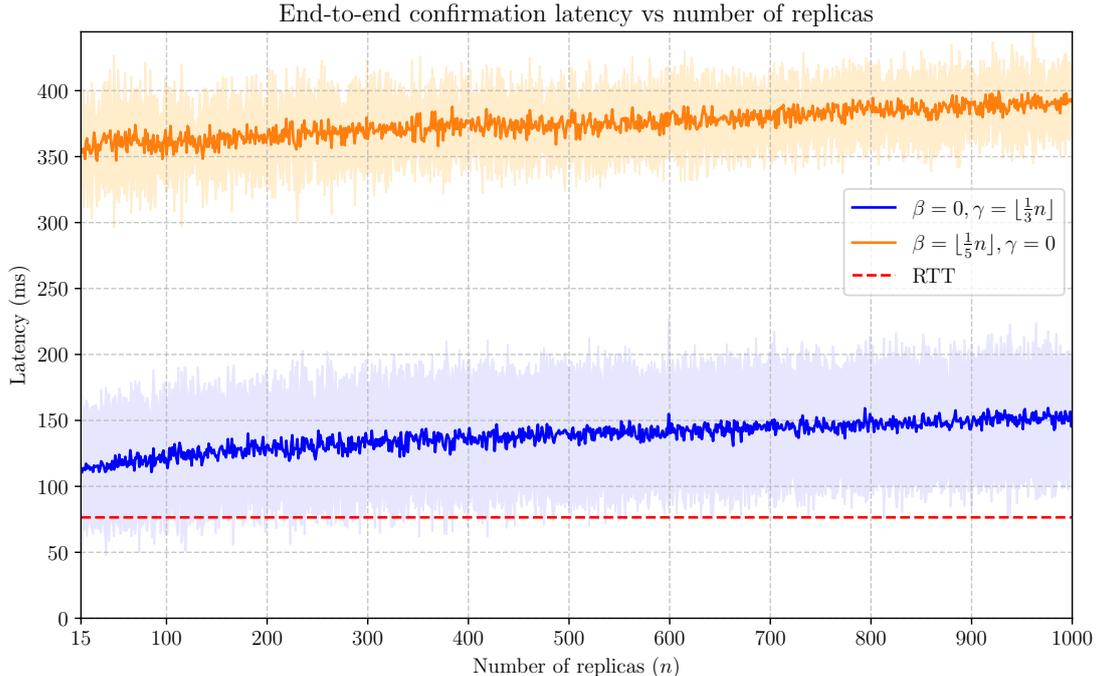


Fig. 4: End-to-end confirmation latency from a writing client to a reading client as a transaction traverses across $n = 15, \dots, 1000$ replicas, for two reading clients: (1) a client that expects up to $\gamma = \lfloor \frac{1}{3}n \rfloor$ omission faults (blue line, below), and (2) a client that 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 signature⁶ on secp256k1 curve, appends it to its local log, and forwards the signed tuple to the reading client.

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 ($n = 15, \dots, 1000$). The latency is computed as the difference between the timestamp recorded by the reading client upon receiving sufficiently many votes from different replicas to confirm the transaction and the initial timestamp recorded by the writing client. We repeat each experiment five times and report the mean latency and a 95% confidence interval.

The deployment process begins by launching replicas using a round-robin strategy 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. After collecting all replica IP addresses and public keys, 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 and public keys).

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

⁶ <https://crates.io/crates/secp256k1>

the $1 - \alpha$ slowest replicas do not slow down confirmation. Even with 1000 replicas the mean confirmation latency is 138ms for the omission fault experiment and 375ms for the Byzantine experiment. Which approximates the physical network RTT 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, which guarantees accountable censorship resistance and consistency among honest parties. We first define `bidset` and then construct it using an underlying `pod`.

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, Cbid)`: 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, Cbid)` by round $t_0 + W$.*

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

(Safety) Weak consistency: *If two honest consumers generate `result(B1, ·)` and `result(B2, ·)` 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 uses a `pod` instance `pod`.

```

1: function SUBMITBID( $b$ )
2:   pod.write( $b$ )
3: end function

```

Algorithm 6 `bidset-core`: Code for the sequencer. It uses a `pod` instance `pod`, and sk_a denotes the secret key of the sequencer.

```

1: function READBIDS()
2:   ( $T, r_{\text{perf}}, C_{\text{pp}}$ )  $\leftarrow$  pod.read()
3:   while  $r_{\text{perf}} \leq t_0 + \Delta$  do
4:     ( $T, r_{\text{perf}}, C_{\text{pp}}$ )  $\leftarrow$  pod.read()
5:   end while
6:    $B \leftarrow \{\text{tx} \mid (\text{tx}, \cdot, \cdot, \cdot, \cdot) \in T\}$ ;  $C_{\text{bid}} \leftarrow C_{\text{pp}}$ 
7:    $\sigma \leftarrow \text{Sign}(sk_a, (B, C_{\text{bid}}))$ 
8:    $\text{tx} \leftarrow \langle \text{BIDS}(B, C_{\text{bid}}, \sigma) \rangle$ 
9:   pod.write( $\text{tx}$ )
10: end function

```

A bidder submits a bid by writing it on the `pod` (Algorithm 5) at round t_0 . The sequencer (Algorithm 6) waits until the `pod` returns a past-perfect round larger than $t_0 + \Delta$ (line 3). The

Algorithm 7 *bidset-core*: Code for a consumer. It uses a *pod* instance *pod*.

```

1: function READRESULT()
2:   loop
3:      $(\mathbb{T}, r_{\text{perf}}, C_{\text{pp}}) \leftarrow \text{pod.read}()$ 
4:     if  $\exists (\text{tx}, \cdot, \cdot, r_{\text{conf}}, \cdot) \in \mathbb{T} : \text{tx} = \langle \text{BIDS}(B, C_{\text{bid}}, \sigma) \rangle$  and  $r_{\text{conf}} \leq t_0 + 3\Delta$  then
5:       output event result( $B, C_{\text{bid}}$ )
6:     else if  $r_{\text{perf}} > t_0 + 3\Delta$  then
7:       output event result( $\emptyset, C_{\text{pp}}$ )
8:     end if
9:   end loop
10: end function

```

δ -*timeliness* property of the *pod*, given that $\delta \leq \Delta$, ensures that the bids of honest parties will have a confirmation round at most $t_0 + \Delta$, and the *accountable past-perfection safety* guarantees that a malicious sequencer cannot exclude them from the bid-set B in 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 $\langle \text{BIDS}(B, C_{\text{bid}}, \sigma) \rangle$ on *pod*. As an intuition on how *bidset-core* achieves its properties, observe that 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 $\langle \text{BIDS}(B, C_{\text{bid}}, \sigma) \rangle$ 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 $r_{\text{conf}} \leq t_0 + \Delta + 2\delta$ (from the δ -*timeliness* property).

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 $\langle \text{BIDS}(B, C_{\text{bid}}, \sigma) \rangle$ appears in \mathbb{T} , for which $r_{\text{conf}} \leq t_0 + 3\Delta$ (line 4), in which case it outputs the bid-set B in the *result()* event. Second, a round higher than $t_0 + 3\Delta$ becomes past-perfect in *pod* (line 6), in which case it outputs $B = \emptyset$.

Observe that, 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 $W = 3\Delta + \delta$* . If the network is synchronous and the sequencer honest, the condition in line 4 becomes true at round at most $t_0 + \Delta + 3\delta$ for an honest consumer, hence it outputs B as the bid-set, hence *bidset-core* satisfies the *consistency* property.

Remark 7 (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.

Theorem 3 (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 8. 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, 6, 9, 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. `Pod` eliminates communication among replicas. Instead, clients read the system state by performing lightweight computation on logs retrieved from the replicas. As no validator has a particular role in `pod` (as compared to leaders, block proposers, miners, etc. in similar protocols), `pod` achieves censorship resistance by default, without any extra mechanisms or additional cost. Furthermore, validator misbehavior, such as voting in incompatible ways or censoring confirmed transactions, is accountable.

As an application, we present an efficient and censorship-resistant auction mechanism, which leverages `pod` as a bulletin board. We saw how the accountability, offered by `pod`, is also inherited by applications built on it – the auctioneer cannot censor confirmed bids without being detected.

We remark that `pod` differs from standard notions of consensus because it does not offer an agreement property, neither to validators nor to clients. A client reading the `pod` obtains a past-perfect round r_{perf} , and it is guaranteed to have received all transactions that obtained a confirmed round r_{conf} such that $r_{\text{conf}} \leq r_{\text{perf}}$. It is also guaranteed to have received all transactions that can potentially obtain an $r_{\text{conf}} \leq r_{\text{perf}}$ 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. R. A. Bazzi and S. T. Piergiovanni. The fractional spending problem: Executing payment transactions in parallel with less than $f + 1$ validations. In R. Gelles, D. Olivetti, and P. Kuznetsov, editors, *43rd ACM PODC*, pages 295–305. ACM, June 2024.
6. 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.
7. D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygiak. 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.
8. 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.
9. B. David, L. Gentile, and M. Pourpouneh. FAST: Fair auctions via secret transactions. In G. Ateniese and D. Venturi, editors, *ACNS 22 International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 727–747. Springer, Cham, June 2022.
10. I. Doidge, R. Ramesh, N. Shrestha, and J. Tobkin. Moonshot: Optimizing chain-based rotating leader BFT via optimistic proposals. *CoRR*, abs/2401.01791, 2024.
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. D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright. Probabilistic quorum systems. *Inf. Comput.*, 170(2):184–206, 2001.
20. 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.
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. 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, r_{\min} does not decrease, r_{\max} does not increase and confirmed transactions remain confirmed.

Definition 14 (pod monotonicity). *A pod protocol satisfies pod monotonicity, and 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 $D_{r_2}^c.r_{\text{perf}} \geq D_{r_1}^c.r_{\text{perf}}$.*

Transaction monotonicity: *If transaction tx appears in $D_{r_1}^c.T$, then tx appears in $D_{r_2}^c.T$.*

Confirmation-bounds monotonicity: *For every tx that appears in $D_{r_1}^c.T$ with $r_{\min}, r_{\max}, r_{\text{conf}}$ and appears in $D_{r_2}^c.T$ with $r'_{\min}, r'_{\max}, r'_{\text{conf}}$, 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 9. 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 $D_{r_1}^c$, which is stored as state until c reads the pod again. At any round $r_2 > r_1$, if c reads the pod and obtains $D_{r_2}^c$, c can define a view $\overline{D}_{r_2}^c$ satisfying the properties of pod monotonicity:

1. If tx appears in $D_{r_1}^c$ with $\text{tx}.r_{\min}, \text{tx}.r_{\max}, \text{tx}.r_{\text{conf}}, \text{tx}.C_{\text{tx}}$, then tx appears in $\overline{D}_{r_2}^c$ with $\text{tx}.\overline{r}_{\min} = r_{\min}, \text{tx}.\overline{r}_{\max} = r_{\max}, \text{tx}.\overline{r}_{\text{conf}} = r_{\text{conf}}, \text{tx}.\overline{C}_{\text{tx}} = C_{\text{tx}}$.
2. If tx appears in $D_{r_2}^c$ with $\text{tx}.r'_{\min}, \text{tx}.r'_{\max}, \text{tx}.r'_{\text{conf}}, \text{tx}.C'_{\text{tx}}$ and does not appear in $D_{r_1}^c$, then tx appears in $\overline{D}_{r_2}^c$ with $\text{tx}.\overline{r}_{\min} = r'_{\min}, \text{tx}.\overline{r}_{\max} = r'_{\max}, \text{tx}.\overline{r}_{\text{conf}} = r'_{\text{conf}}, \text{tx}.\overline{C}_{\text{tx}} = C'_{\text{tx}}$.
3. For every tx that appears in $D_{r_1}^c.T$ and in $D_{r_2}^c.T$ such that $r'_{\min} \geq r_{\min}, r'_{\max} \leq r_{\max}, r'_{\text{conf}} \geq r_{\text{conf}}$, update $\text{tx}.\overline{r}_{\min} = r'_{\min}, \text{tx}.\overline{r}_{\max} = r'_{\max}, \text{tx}.\overline{r}_{\text{conf}} = r'_{\text{conf}}, \text{tx}.\overline{C}_{\text{tx}} = C'_{\text{tx}}$.
4. If $D_{r_2}^c.r_{\text{perf}} > D_{r_1}^c.r_{\text{perf}}$, then $\overline{D}_{r_2}^c.r_{\text{perf}} = D_{r_2}^c.r_{\text{perf}}$. Otherwise, $\overline{D}_{r_2}^c.r_{\text{perf}} = D_{r_1}^c.r_{\text{perf}}$.

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

Remark 10 (Confirmation monotonicity). Properties 2 and 3 of pod monotonicity imply that for any honest client c and rounds $r_1, r_2 > r_1$, if $\text{tx} \in D_{r_1}^c$ and $\text{tx}.r_{\text{conf}} \neq \perp$, then $\text{tx} \in D_{r_2}^c$ and $\text{tx}.r_{\text{conf}} \neq \perp$.

Remark 11 (Strong confirmation-bounds monotonicity). Properties 2 and 3 of pod monotonicity imply that if an honest client computes r_{\min} and r_{\max} for tx in round r_1 , then the same client will later obtain r'_{\min} and r'_{\max} for tx in round $r_2 > r_1$ such that $r'_{\min} \geq r_{\min}$ and $r'_{\max} \leq r_{\max}$.

Remark 12. Observe that the *confirmation monotonicity* property in Remark 10 is a specific version of a more general *common subset* property, which would demand the condition for any two honest clients c_1, c_2 .

B Security of Protocol pod-core under Byzantine 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. *Regarding Algorithm 4, we have the following. Assume we keep all received timestamps for a transaction tx (n values in total) in a structure `timestamps`, filling in a special value (0 or ∞) for missing votes, sorted in increasing order. Assume `mrt` is also sorted in increasing order of timestamps.*

1. `minPossibleTs()` returns the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of `timestamps`.
2. `maxPossibleTs()` returns the timestamp at index $n - \alpha + \lfloor \alpha/2 \rfloor + \beta$ of `timestamps`.
3. `minPossibleTsForNewTx()` returns the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of `mrt`.

Proof. Functions `minPossibleTs()` and `minPossibleTsForNewTx()` 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 $n - \alpha + \beta + \lfloor \alpha/2 \rfloor$. \square

Lemma 2. *Assuming $n \geq 5\beta + 1$ and $\gamma = 0$, if a client outputs r_{perf} as the past-perfect round, then there exists some honest replica R_j , such that the most-recent timestamp (`mrt`) from R_j received by the client satisfies $\text{mrt}[R_j] \leq r_{\text{perf}}$.*

Proof. From Lemma 1 we have that r_{perf} is the timestamp at index $(\lfloor \alpha/2 \rfloor - \beta)$ of sorted `mrt`. The condition $\alpha \geq 4\beta + 1$ implies that $\beta < \lfloor \alpha/2 \rfloor - \beta$, i.e., there are less than $\lfloor \alpha/2 \rfloor - \beta$ malicious parties, hence one of the indexes between 0 and $\lfloor \alpha/2 \rfloor - \beta$ (inclusive) will contain the timestamp sent to the client by an honest replica. \square

We now recall Theorem 1, which we prove through a series of lemmas.

Theorem 1 (pod-core security under Byzantine faults). *Assuming that the network is asynchronous with actual network delay δ (and unknown delay upper bound), β is the number of Byzantine replicas, $\gamma = 0$ is the number of omission-faulty replicas and $n \geq 5\beta + 1$ is the total number of replicas. Protocol `pod-core` (Protocol 1) instantiated with a `EUFCMA` secure signature scheme is a responsive secure `pod` (Definition 10) with Confirmation within $u = 2\delta$, Past-perfection within $w = \delta$ and β -accountable safety (Definition 3) with the `identify()` function described in Algorithm 8, except with negligible probability.*

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

Lemma 3 (Confirmation within u). *Protocol 1, assuming $n \geq 5\beta + 1$ and $\gamma = 0$, satisfies the confirmation within u property for $u = 2\delta$.*

Proof. Assume an honest client c calls `write(tx)` at round r . It sends a message $\langle \text{WRITE tx} \rangle$ to all replicas at round r (line 3). An honest replica receives this by round $r + \delta$ and sends a $\langle \text{VOTE} \rangle$ message back to all connected clients (line 22). An honest client c' receives the vote by round $r + 2\delta$. As there are at least α honest replicas, c' receives at least α such votes, hence the condition in line 12 is satisfied and c' observes tx as confirmed. \square

Lemma 4 (Past-perfection within w). *Protocol 1, assuming $n \geq 5\beta + 1$ and $\gamma = 0$, satisfies the past-perfection within w property for $w = \delta$.*

Proof. Assume an honest client c at round r has view D_r^c . From Lemma 2, there exists some honest replica R_j , such that the most-recent timestamp $\text{mrt}[R_j]$ that R_j has sent to c satisfies $D_r^c.r_{\text{perf}} \geq \text{mrt}[R_j]$. 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 $\text{mrt}[R_j]$ when it receives the heartbeat or vote message. Hence, c will have $\text{mrt}[R_j] \geq r - \delta$. All together, $D_r^c.r_{\text{perf}} \geq r - \delta$. \square

Lemma 5 (Past-perfection safety). *Protocol 1, assuming $n \geq 5\beta + 1$ and $\gamma = 0$, satisfies the past-perfection safety property.*

Proof. Assume an honest client c at round $r > 0$ reads the pod and obtains $(\mathbb{T}, r_{\text{perf}}, C_{\text{pp}})$. Let \mathcal{R}_1 be the set of replicas R_i for which c stores at round r an $\text{mrt}_i \geq r_{\text{perf}}$. From Lemma 1 (r_{perf} is computed as the timestamp at index $\lfloor \alpha/2 \rfloor - \beta$ of sorted mrt) there exist at least $n - \lfloor \alpha/2 \rfloor + \beta$ such replicas, hence $|\mathcal{R}_1| \geq n - \lfloor \alpha/2 \rfloor + \beta$. For each $R_i \in \mathcal{R}_1$, client c has received the whole log of R_i with timestamps up to mrt_i (line 17 of Algorithm 2 does not allow gaps in the sequence number of the received votes). That is, for each $R_i \in \mathcal{R}_1$ client c has received votes

$$(\text{tx}_{i,1}, \text{ts}_{i,1}, 1, \sigma_{i,1}, R_i), (\text{tx}_{i,2}, \text{ts}_{i,2}, 2, \sigma_{i,2}, R_i), \dots, (\text{tx}_{i,m_i}, \text{ts}_{i,m_i}, m_i, \sigma_{i,m_i}, R_i), \quad (1)$$

where m_i is the smallest sequence number for which $\text{ts}_{i,m_i} \geq r_{\text{perf}}$.

Assume another client c' at some round $r' > 0$ outputs a pod $(\mathbb{T}', \cdot, \cdot)$, such that $(\text{tx}, \cdot, \cdot, r_{\text{conf}}, C_{\text{tx}}) \in \mathbb{T}'$ and $r_{\text{conf}} < r_{\text{perf}}$ (where r_{perf} is the past-perfect round observed by c). For c' to output a confirmation round $r_{\text{conf}} < r_{\text{perf}}$, it must have received timestamps ts_i on tx , such that $\text{ts}_i < r_{\text{perf}}$, from at least $\lfloor \alpha/2 \rfloor + 1$ (because r_{conf} is computed as the median of at least α votes). Let \mathcal{R}_2 be the set of these replicas, with $|\mathcal{R}_2| \geq \lfloor \alpha/2 \rfloor + 1$. For each $R_i \in \mathcal{R}_2$, client c' has received a vote

$$(\text{tx}, \text{ts}_i, \text{sn}_i, \sigma_i, R_i), \quad (2)$$

such that $\text{ts}'_i < r_{\text{perf}}$.

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 eq. (2) must be one of the votes in eq. (1) since $\text{ts}_i < r_{\text{perf}}$ and $\text{ts}_{j,m_i} \geq r_{\text{perf}}$. Hence, client c has received at least one vote on tx , so $\text{tx} \in \mathbb{T}$, as demanded by the past-perfection property. \square

Lemma 6 (Confirmation bounds). *Protocol 1, assuming $n \geq 5\beta + 1$ and $\gamma = 0$, satisfies the confirmation bounds safety property.*

Proof. Assume client c_1 computes r_{min} and client c_2 computes $r_{\text{conf}} < r_{\text{min}}$ for tx . From Lemma 1, the number of replicas that have sent to c_1 a timestamp for tx smaller than r_{min} can be at most $\lfloor \alpha/2 \rfloor - \beta$. Allowing up to β replicas to equivocate, there can be at most $\lfloor \alpha/2 \rfloor$ replicas that send c_2 a timestamp smaller than r_{min} . However, in order to assign $r_{\text{conf}} < r_{\text{min}}$ to tx , client c_2 must have received timestamps smaller than r_{min} from at least $\lfloor \alpha/2 \rfloor + 1$ replicas, a contradiction. For r_{max} , assume c_1 computes r_{max} and c_2 computes $r_{\text{conf}} > r_{\text{max}}$. Using Lemma 1, the number of replicas that have sent to c_1 a timestamp larger than r_{max} can be at most $\alpha - \lfloor \alpha/2 \rfloor - \beta - 1$, hence the number of honest replicas that will send a timestamp larger than r_{max} to c_2 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 c_2 would need at least $\lfloor \alpha/2 \rfloor + 1$ votes larger than r_{max} , and if α is even, then $\alpha - \lfloor \alpha/2 \rfloor - 1 = \lfloor \alpha/2 \rfloor - 1$, while c_2 would need at least $\lfloor \alpha/2 \rfloor$ votes larger than r_{max} in order to compute a median larger than r_{max} (we remind that algorithm 4 returns as median the value at position $\lfloor \alpha/2 \rfloor$). \square

Lemma 7 (β -Accountable safety). *Protocol 1, assuming $n \geq 5\beta + 1$ and $\gamma = 0$, satisfies accountable safety with resilience β .*

Proof. We show that `identify()` (Algorithm 8) satisfies the *correctness* and *no-framing* properties required by Definition 3, in three steps.

1. If the past-perfection safety property is violated, there exists a partial transcript T , such that `identify()` on input T returns at least β adversarial replicas.

Proof: Assume an honest client c at round $r > 0$ reads the pod and obtains $(\mathbb{T}, r_{\text{perf}}, C_{\text{pp}})$, such that $\text{tx} \notin \mathbb{T}$, for some $\text{tx} \in \{0,1\}^*$, and another client c' outputs a pod $(\mathbb{T}', \cdot, \cdot)$, such that $(\text{tx}, \cdot, \cdot, r_{\text{conf}}, C_{\text{tx}}) \in \mathbb{T}'$ and $r_{\text{conf}} < r_{\text{perf}}$, thereby violating the past-perfection property for tx . We resume the proof of Lemma 5. There, we constructed the sets $\mathcal{R}_1, \mathcal{R}_2$, such that $\mathcal{R}_1 \cap \mathcal{R}_2$ contains at least $\beta + 1$. For each $R_i \in \mathcal{R}_1 \cap \mathcal{R}_2$, client c has received the replica log shown in

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

```

1: function identify( $T$ )
2:    $\tilde{R} \leftarrow \emptyset$ 
3:   for  $\langle VOTE(\mathbf{tx}_1, \mathbf{ts}_1, \mathbf{sn}_1, \sigma_1, R_1) \rangle \in T$  do
4:     if not Verify( $\rho k_1, (\mathbf{tx}_1, \mathbf{ts}_1, \mathbf{sn}_1), \sigma_1$ ) then
5:       continue
6:     end if
7:     for  $\langle VOTE(\mathbf{tx}_2, \mathbf{ts}_2, \mathbf{sn}_2, \sigma_2, R_2) \rangle \in T$  do
8:       if not Verify( $\rho k_2, (\mathbf{tx}_2, \mathbf{ts}_2, \mathbf{sn}_2), \sigma_2$ ) then
9:         continue
10:      end if
11:      if  $R_1 = R_2$  and  $\mathbf{sn}_1 = \mathbf{sn}_2$  and  $(\mathbf{tx}_1 \neq \mathbf{tx}_2$  or  $\mathbf{ts}_1 \neq \mathbf{ts}_2)$  then
12:         $\tilde{R} \leftarrow \tilde{R} \cup \{R_1\}$ 
13:      end if
14:    end for
15:  end for
16: end function

```

(1), containing all votes with timestamp up to $\mathbf{ts}_{i,m_i} \geq r_{\text{perf}}$. Client c' has received from R_i the following m'_i votes (possibly more, but we care for the votes up to transaction \mathbf{tx})

$$(\mathbf{tx}'_{i,1}, \mathbf{ts}'_{i,1}, 1, \sigma'_{i,1}, R_i), (\mathbf{tx}'_{i,2}, \mathbf{ts}'_{i,2}, 2, \sigma'_{i,2}, R_i), \dots, (\mathbf{tx}'_{i,m'_i}, \mathbf{ts}'_{i,m'_i}, m'_i, \sigma'_{i,m'_i}, R_i), \quad (3)$$

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

Summarizing, we have shown for at least $\beta + 1$ replicas $R_i \in \mathcal{R}_1 \cap \mathcal{R}_2$, clients c and c' hold votes $(\mathbf{tx}_1, \mathbf{ts}_1, \mathbf{sn}_1, \sigma_1, R_i)$ and $(\mathbf{tx}_2, \mathbf{ts}_2, \mathbf{sn}_2, \sigma_2, R_i)$, such that $\mathbf{sn}_1 = \mathbf{sn}_2$ but $\mathbf{tx}_1 \neq \mathbf{tx}_2$ or $\mathbf{ts}_1 \neq \mathbf{ts}_2$. 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 is violated, there exists a partial transcript T , such that Algorithm 8 on input T returns at least β adversarial replicas.

Proof: Assume a client c_1 outputs some r_{min} and client c_2 outputs $r_{\text{conf}} < r_{\text{min}}$ for \mathbf{tx} . Let's look at the timestamps in timestamps when c_1 computes r_{min} (function *minPossibleTs()* in Algorithm 4.) From Lemma 1, timestamps contains at least $n - \lfloor \alpha/2 \rfloor + \beta$ timestamps \mathbf{ts} , such that $\mathbf{ts} \geq r_{\text{min}}$. Hence, there is a set \mathcal{R}_1 with at least $n - \lfloor \alpha/2 \rfloor + \beta$ replicas R_i , from each of which c_1 has received votes

$$(\mathbf{tx}_{i,1}, \mathbf{ts}_{i,1}, 1, \sigma_{i,1}, R_i), (\mathbf{tx}_{i,2}, \mathbf{ts}_{i,2}, 2, \sigma_{i,2}, R_i), \dots, (\mathbf{tx}_{i,m_i}, \mathbf{ts}_{i,m_i}, m_i, \sigma_{i,m_i}, R_i), \quad (4)$$

up to some sequence number m_i , such that $\mathbf{ts}_{i,m_i} \geq r_{\text{min}}$ and either $\mathbf{tx}_{i,m_i} = \mathbf{tx}$ (i.e., R_i has sent to c_1 a vote on \mathbf{tx} , and we only consider the votes up to this one), or $\mathbf{tx}_{i,j} \neq \mathbf{tx}, \forall j \leq m_i$ (i.e., R_i has not sent to c_1 a vote on \mathbf{tx} , in which case timestamps contains the timestamp R_i has sent on \mathbf{tx}_{i,m_i}).

Now, for c_2 to output $r_{\text{conf}} < r_{\text{min}}$, it must have received timestamps smaller than r_{min} from at least $\lfloor \alpha/2 \rfloor + 1$ replicas. Call this set \mathcal{R}_2 . From each of these replicas, c_2 has received votes

$$(\mathbf{tx}'_{i,1}, \mathbf{ts}'_{i,1}, 1, \sigma'_{i,1}, R_i), (\mathbf{tx}'_{i,2}, \mathbf{ts}'_{i,2}, 2, \sigma'_{i,2}, R_i), \dots, (\mathbf{tx}, \mathbf{ts}'_{i,m'_i}, m'_i, \sigma'_{i,m'_i}, R_i), \quad (5)$$

showing only votes up to \mathbf{tx} , for which $\mathbf{ts}'_{i,m'_i} < r_{\text{min}}$.

By counting arguments there are at least $\beta + 1$ replicas in $\mathcal{R}_1 \cap \mathcal{R}_2$. For each one, we make the following argument. Since $\text{ts}_{i,m_i} \geq r_{\min}$ and $\text{ts}'_{i,m'_i} < r_{\min}$, we get $\text{ts}'_{i,m'_i} < \text{ts}_{i,m_i}$, 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., $\text{tx}_{i,m'_i} \neq \text{tx}'_{i,m'_i} = \text{tx}$. 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 $\text{identify}(T)$ returns $\mathcal{R}_1 \cap \mathcal{R}_2$.

For the case client c_1 outputs some r_{\max} and client c_2 outputs $r_{\text{conf}} \geq r_{\max}$ for tx , similar arguments apply. As explained in the proof of Lemma 6, c_2 has received at least $\lfloor \alpha/2 \rfloor$ or $\lfloor \alpha/2 \rfloor + 1$ votes on tx with timestamp larger than r_{\max} , and (from Lemma 1) at least $n - \alpha + \lfloor \alpha/2 \rfloor + \beta$ replicas have sent to c_1 a vote on tx with a timestamp smaller or equal than r_{\max} . As before, the replicas in the intersection of these two sets have sent conflicting votes for some sequence numbers.

3. The $\text{identify}()$ function never outputs honest replicas.

Proof: The function only adds a replica to \hat{R} if given as input two valid vote messages from that replica, in which it assigns the same sequence number to two different votes (line 11 on Algorithm 8). An honest replica always increments nextsn after each vote it inserts to its log (line 24 on Algorithm 1), hence, except with negligible probability, no such valid messages can be constructed for an honest replica. \square

Theorem 4 (θ -timeliness for honest transactions). *Protocol 1, assuming $n \geq 5\beta + 1$ and $\gamma = 0$, satisfies θ -timeliness for honest transactions, as per Definition 11, for $\theta = \delta$.*

Proof. Assume an honest client c calls $\text{write}(\text{tx})$ at round r . It sends a message $\langle \text{WRITE tx} \rangle$ to all replicas at round r (line 3). An honest replica receives this by round $r + \delta$ and assigns its current round $r' \in (r, r + \delta]$ as the timestamp (line 19).

1. Regarding r_{conf} , when a client calls $\text{read}()$, and since by assumption tx is confirmed, it will have received votes on tx from at least α replicas. All honest replicas have sent timestamps for tx in the interval $(r, r + \delta]$. Since r_{conf} is computed as the median of α timestamps and $\alpha \geq 4\beta + 1$,⁷ it will be a timestamp returned by an honest replica, or it will lie between timestamps returned by honest replicas. Hence, $r_{\text{conf}} \in (r, r + \delta]$.
2. Regarding r_{\max} , from Lemma 1, at least $\lfloor \alpha/2 \rfloor - 2\beta$ replicas have sent to c a timestamp greater or equal than r_{\max} and, since $\alpha \geq 4\beta + 1$, at least one of those is a timestamp returned by an honest replica, hence $r_{\max} \in (r, r + \delta]$.
3. Similarly, from Lemma 1 we get that r_{\min} is a timestamp returned by an honest replica, hence $r_{\min} \in (r, r + \delta]$ and $r_{\max} - r_{\min} < \theta$.

\square

C Security of Protocol pod-core under omission faults

Let us assume now an execution where $\gamma = 0$, i.e., no Byzantine replicas exist, and $\gamma > 0$, i.e., there are omission-faulty replicas. In this section we present lemmas and proofs that are analogous to those in Appendix B, accounting for this change.

Lemma 8. *Regarding Algorithm 4, we have the following, assuming $\beta = 0$. Assume we keep all received timestamps for a transaction tx (n values in total) in a structure timestamps , filling in a special value (0 or ∞) for missing votes, sorted in increasing order. Assume mrt is also sorted in increasing order of timestamps.*

1. $\text{minPossibleTs}()$ returns the timestamp at index $\lfloor \alpha/2 \rfloor$ of timestamps.
2. $\text{maxPossibleTs}()$ returns the timestamp at index $n - \alpha + \lfloor \alpha/2 \rfloor$ of timestamps.

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

3. $\text{minPossibleTsForNewTx}()$ returns the timestamp at index $\lfloor \alpha/2 \rfloor$ of mrt .

Proof. Follows by observation of Algorithm 4. \square

Lemma 9. *Assuming $\beta = 0$ and $\alpha \geq 2\gamma + 1$, if a client outputs r_{perf} as the past-perfect round, then there exists some honest (not crashed and not omitting messages) replica R_j , such that the most-recent timestamp (mrt) from R_j received by the client satisfies $\text{mrt}[R_j] \leq r_{\text{perf}}$.*

Proof. From Lemma 8 we have that r_{perf} is the timestamp at index $\lfloor \alpha/2 \rfloor$ of sorted mrt . The condition $\alpha \geq 2\gamma + 1$ implies that $\gamma \leq \lfloor \alpha/2 \rfloor$, hence one of the indexes between 0 and $\lfloor \alpha/2 \rfloor$ (inclusive) will contain the timestamp sent to the client by an honest replica. \square

We now recall Theorem 2, which we prove through a series of lemmas.

Theorem 2 (pod-core security under omission faults). *Assuming that the network is asynchronous with actual network delay δ (and unknown delay upper bound), $\beta = 0$ is the number of Byzantine replicas, γ is the number of omission-faulty replicas and $n \geq 3\gamma + 1$ is the total number of replicas. Protocol **pod-core** (Protocol 1) instantiated with an EUF-CMA secure signature scheme is a responsive secure pod (Definition 10) with Confirmation within $u = 2\delta$ and Past-perfection within $w = \delta$, except with negligible probability.*

Proof. The proof follows from Lemmas 10–13, presented and proven in the remainder of this section. \square

Lemma 10 (Confirmation within u). *Protocol 1, assuming $\beta = 0$ and $n \geq 3\gamma + 1$, satisfies the confirmation within u property for $u = 2\delta$.*

Proof. Assume an honest client c calls $\text{write}(\text{tx})$ at round r . It sends a message $\langle \text{WRITE tx} \rangle$ to all replicas at round r (line 3). An honest replica receives this by round $r + \delta$ and sends a $\langle \text{VOTE} \rangle$ message back to all connected clients (line 22). An honest client c' receives the vote by round $r + 2\delta$. As there are at least α honest replicas, c' receives at least α such votes, hence the condition in line 12 is satisfied and c' observes tx as confirmed. \square

Lemma 11 (Past-perfection within w). *Protocol 1, assuming $\beta = 0$ and $n \geq 3\gamma + 1$, satisfies the past-perfection within w property for $w = \delta$.*

Proof. The proof is the same as the proof of Lemma 4, except that it uses Lemma 9 instead of Lemma 2 for the argument that there exists some honest replica R_j for which c outputs $D_r^c.r_{\text{perf}} \geq \text{mrt}[R_j]$. \square

Lemma 12 (Past-perfection safety). *Protocol 1, assuming $\beta = 0$ and $n \geq 3\gamma + 1$, satisfies the past-perfection safety property.*

Proof. As in the proof of Lemma 5, we construct set \mathcal{R}_1 . Now we have that $|\mathcal{R}_1| \geq n - \lfloor \alpha/2 \rfloor$, because from Lemma 8 r_{perf} is computed as the timestamp at index $\lfloor \alpha/2 \rfloor$ of sorted mrt . We construct \mathcal{R}_2 in the same way, for which we get $|\mathcal{R}_2| \geq \lfloor \alpha/2 \rfloor + 1$. From the cardinality of \mathcal{R}_1 and \mathcal{R}_2 , at least one replica must be in both sets, except with negligible probability. The result follows from the same argument: for that replica, the vote in eq. (2) must be one of the votes in eq. (1). \square

Lemma 13 (Confirmation bounds). *Protocol 1, assuming $\beta = 0$ and $n \geq 3\gamma + 1$, satisfies the confirmation bounds safety property.*

Proof. Assume client c_1 computes r_{min} and client c_2 computes $r_{\text{conf}} < r_{\text{min}}$ for tx . From Lemma 8, the number of replicas that have sent to c_1 a timestamp for tx smaller than r_{min} can be at most $\lfloor \alpha/2 \rfloor$, hence there can be at most $\lfloor \alpha/2 \rfloor$ replicas that send c_2 a timestamp smaller than r_{min} (no replicas are Byzantine). However, in order to assign $r_{\text{conf}} < r_{\text{min}}$ to tx , client c_2 must have received timestamps smaller than r_{min} from at least $\lfloor \alpha/2 \rfloor + 1$ replicas, a contradiction. Similarly for r_{max} , assume c_1 computes r_{max} and c_2 computes $r_{\text{conf}} > r_{\text{max}}$. Using Lemma 8, the number of replicas that have sent to c_1 a timestamp larger than r_{max} can be at most $\alpha - \lfloor \alpha/2 \rfloor - 1$, hence

at most that many replicas will send a timestamp larger than r_{\max} to c_2 . If α is odd, this upper bound becomes $\alpha - \lfloor \alpha/2 \rfloor - 1 = \lfloor \alpha/2 \rfloor$, while c_2 would need at least $\lfloor \alpha/2 \rfloor + 1$ votes larger than r_{\max} , and if α is even, then $\alpha - \lfloor \alpha/2 \rfloor - 1 = \lfloor \alpha/2 \rfloor - 1$, while c_2 would need at least $\lfloor \alpha/2 \rfloor$ votes larger than r_{\max} in order to compute a median larger than r_{\max} (we remind that algorithm 4 returns as median the value at position $\lfloor \alpha/2 \rfloor$). \square

Theorem 5 (θ -timeliness for honest transactions). *Protocol 1, assuming $\beta = 0$ and $n \geq 3\gamma + 1$, satisfies θ -timeliness for honest transactions, as per Definition 11, for $\theta = \delta$.*

Proof. Assume an honest client c calls $\text{write}(tx)$ at round r . It sends a message $\langle \text{WRITE } tx \rangle$ to all replicas at round r (line 3). An honest replica receives this by round $r + \delta$ and assigns its current round $r' \in (r, r + \delta]$ as the timestamp (line 19). Since only γ replicas may omit sending votes for tx , at least α replicas will send this timestamp to all clients. It follows that $r_{\text{conf}} \in (r, r + \delta], r_{\text{min}} \in (r, r + \delta], r_{\text{max}} \in (r, r + \delta]$. \square

D Security of bidset-core

In this section, we recall and prove Theorem 3.

Theorem 3 (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 14–17. The function for identifying a malicious sequencer is shown in Algorithm 9. \square

Lemma 14 (Termination within W). *Under the assumptions of Theorem 3, Protocol 2 satisfies termination within $W = t_0 + 3\Delta + \delta$.*

Proof. The `result()` event is generated by an honest consumer when it 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 $t_0 + \Delta + \delta$. \square

Lemma 15 (Censorship resistance). *Under the assumptions of Theorem 3, 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_r^a of the sequencer a on the round r when it constructs B satisfies $D_r^a.r_{\text{perf}} > t_2$. 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 $r_{\text{conf}} \leq t_0 + \theta$. For $\theta = \delta$, and since $\delta \leq \Delta$, we get that $r_{\text{conf}} \leq t_2$. Hence, from the past-perfection safety property of **pod** we get that $b \in D_r^a$, and, since the sequencer is honest, $b \in B$. \square

Lemma 16 (Consistency). *Under the assumptions of Theorem 3, 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 $\langle \text{BIDS } (B, C_{\text{bid}}, \sigma) \rangle$ to the pod by round $t_0 + \Delta + \delta$. This transaction gets assigned a confirmed round $r_{\text{conf}} \leq t_0 + \Delta + 2\delta$ (from the δ -timeliness property of **pod**) and, by assumption of a synchronous network, $r_{\text{conf}} \leq t_0 + 3\Delta$. The condition in line 4 of Algorithm 7

requires that a round $r' > t_0 + 3\Delta$ becomes past perfect. As $r' > r_{\text{conf}}$, 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 $\text{result}(B, C_{\text{bid}})$. \square

Lemma 17 (Accountable safety). *Under the assumptions of Theorem 3, 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 2.2, 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 $(\text{tx}, r_{\text{conf}}, \cdot, \cdot, C_{\text{tx}})$ in their view of the `pod`, where C_{tx} is the auxiliary data associated by tx , as returned by `pod-core`. Assume b is censored, i.e., an event $\text{result}(B, C_{\text{bid}})$ is output by an honest consumer, such that $b \notin B$. Let σ be the signature of the sequencer in the corresponding $\langle \text{BIDS}(B, C_{\text{bid}}, \sigma) \rangle$ 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 (line 3).

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

If any of these requirements are not met, T does not constitute valid evidence and the function exits. Otherwise, let r_{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 $r_{\text{conf}}^* \leq t_0 + \Delta$ 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 $\langle \text{BIDS}(B_1, \cdot, \cdot) \rangle$ and $\langle \text{BIDS}(B_2, \cdot, \cdot) \rangle$ 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 $r_{\text{conf}}^* \leq t_0 + \Delta$, 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 $n - \lfloor \alpha/2 \rfloor + \beta$ 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 . \square

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 $\mathcal{R} = \{R_1, \dots, R_n\}$ with public keys $\{\text{pk}_1, \dots, \text{pk}_n\}$, 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:   ( $C_{\text{tx}}, B, C_{\text{bid}}, \sigma$ )  $\leftarrow T$ 
3:   require Verify( $\text{pk}_a, (B, C_{\text{bid}}), \sigma$ )
4:   timestamps  $\leftarrow []$ 
5:   for vote  $\in C_{\text{bid}}$  do
6:     ( $\text{tx}, \text{ts}, \text{sn}, \sigma, R_j$ )  $\leftarrow$  vote
7:     if Verify( $\text{pk}_j, (\text{tx}, \text{ts}, \text{sn}), \sigma = 0$ ) then
8:       return true
9:     end if
10:    timestamps  $\leftarrow$  timestamps  $\parallel$  ts
11:  end for
12:  sort timestamps in increasing order
13:  timestamps  $\leftarrow [0, \overset{\beta}{\dots}, 0] \parallel$  timestamps
14:   $r_{\text{perf}} \leftarrow \text{median}(\text{timestamps}[: \alpha])$ 
15:  if  $r_{\text{perf}} \leq t_0 + \Delta$  then
16:    return true
17:  end if

18:  require  $|C_{\text{tx}}| \geq \alpha$ 
19:   $\text{tx}^* \leftarrow C_{\text{tx}}[0].\text{tx}$ 
20:  timestamps  $\leftarrow []$ 
21:  for vote  $\in C_{\text{tx}}$  do
22:    ( $\text{tx}, \text{ts}, \text{sn}, \sigma, R_j$ )  $\leftarrow$  vote
23:    require  $\text{tx} = \text{tx}^*$ 
24:    require Verify( $\text{pk}_j, (\text{tx}, \text{ts}, \text{sn}), \sigma$ )
25:    timestamps  $\leftarrow$  timestamps  $\parallel$  ts
26:  end for
27:   $r_{\text{conf}}^* \leftarrow \text{median}(\text{timestamps})$ 
28:  if  $r_{\text{conf}}^* \leq t_0 + \Delta$  and  $\text{tx}^* \notin B$  then
29:    return true
30:  end if
31: end function

```
