PnyxDB: a Lightweight Leaderless Democratic Byzantine Fault Tolerant Replicated Datastore
Loïck Bonniot, Christoph Neumann, François Taïani

To cite this version:
Loïck Bonniot, Christoph Neumann, François Taïani. PnyxDB: a Lightweight Leaderless Democratic Byzantine Fault Tolerant Replicated Datastore. 2019. hal-02355778

HAL Id: hal-02355778
https://hal.archives-ouvertes.fr/hal-02355778
Submitted on 8 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
**PnyxDB: a Lightweight Leaderless Democratic Byzantine Fault Tolerant Replicated Datastore**

Loïck Bonniot  
InterDigital  
Univ Rennes, Inria, CNRS, IRISA  
Rennes, France  
loick.bonniot@interdigital.com

Christoph Neumann  
InterDigital  
Rennes, France  
christoph.neumann@interdigital.com

François Taïani  
Univ Rennes, Inria, CNRS, IRISA  
Rennes, France  
francois.taiani@irisa.fr

---

**Abstract**

Byzantine-Fault-Tolerant (BFT) systems are rapidly emerging as a viable technology for production-grade systems, notably in closed consortia deployments for financial and supply-chain applications. Unfortunately, most algorithms proposed so far to coordinate these systems suffer from substantial scalability issues, and lack important features to implement Internet-scale governance mechanisms.

In this paper, we observe that many application workloads offer little concurrency, and propose PnyxDB, an eventually-consistent Byzantine Fault Tolerant replicated datastore that exhibits both high scalability and low latency. Our approach is based on conditional endorsements, that allow nodes to specify the set of transactions that must not be committed for the endorsement to be valid. In addition to its high scalability, PnyxDB supports application-level voting, i.e. individual nodes are able to endorse or reject a transaction according to application-defined policies without compromising consistency. We provide a comparison against BFT-SMArt and Tendermint, two competitors with different design aims, and show that our implementation speeds up commit latencies by a factor of 11, remaining below 5 seconds in a worldwide geodistributed deployment of 180 nodes.

---

**1 Introduction**

Byzantine-Fault-Tolerant (BFT) systems have attracted a large body of works over the last two decades [10, 11, 16, 23, 34, 41, 47, 48], and have now moved into the public spotlight following the dramatic rise of blockchain platforms [26, 55]. These systems typically rely on powerful BFT replication protocols to ensure consistency between their replicas, and withstand arbitrary failures and potential malicious behavior. Unfortunately, traditional BFT replication protocols struggle to scale beyond a few tens of replicas [21], while the proof-of-work technique used by many blockchain-based systems suffers from large computing and storage overheads.

Recent attempts to overcome these scalability barriers have explored leaderless designs [1, 19, 42, 45, 54, 58, 62], alternatives to proof-of-work such as proof-of-stake [32], or assumed access to a trusted third party providing strong coordination and ordering guarantees [4]. All these strategies are however fraught with limitations: existing leaderless protocols rely either on clients for consistency checks [1] (increasing computing overhead) or on the availability of strong coordination mechanisms, such as a trusted peer-sampling service [62] or atomic broadcast primitives [4, 19, 45, 58]; proof-of-stake links a node’s influence to its stake in the system, a problematic dependency for many use cases; and trusted third parties considerably limit the applicability of such solutions to well-controlled environments.

Compounding these limitations, all above approaches are ill-equipped to support *in-system governance mechanisms*, a growing requirement for applications involving independent organizations [33]. More specifically, although most of these solutions rely on internal voting or quorum mechanisms, these mechanisms are not exposed to applications as first-class primitives. As a result, individual nodes cannot implement application-defined policies to endorse or reject transactions without additional effort, costs, and complexity. This is problematic, as such application-level voting capabilities are key to a number of emerging decentralized BFT applications involving independent participants who need to balance conflicting goals and shared interests [26, 56]. Examples of such governance concerns include basic membership management with access control, resource allocation and sharing, crowdsourced scheduling, policy administration and knowledge distribution. In all these examples, different parties are likely to pursue different agendas, prompting the need for participants to be able to influence the distributed decision making process according to their own application-defined policies and beliefs [15, 33, 46].

To address these challenges, we advocate in this paper a radically different line of attack: we borrow a popular strategy from non-Byzantine distributed datastores [35, 63, 66, 68], and tackle scalability by weakening the consistency guarantees, while maintaining Byzantine Fault Tolerance. We illustrate this design with PnyxDB, a *Byzantine-Fault-Tolerant Replicated Datastore for closed consortia*. PnyxDB is eventually consistent in that clients might perceive conflicting views of the datastore for short periods of time. PnyxDB also provides a unique application-level voting mechanism that allow participating nodes to support or reject proposed transactions according to application-defined policies.

Our proposal leverages the long-observed fact that many workloads exhibit a large proportion of commutative and
independent operations \[44, 54\]: these operations can be executed out of order without compromising the eventual convergence of all correct nodes. We exploit these commutative operations through a modified Byzantine Quorum protocol \[47\] that ensures the safety and agreement of our system. More specifically, we introduce conditional endorsements within quorums as a mean to flag and handle conflicts by allowing each node to specify the set of transactions that must not be committed for the endorsement to be valid.

In this paper, we make the following contributions:

1. We present PnyxDB, a scalable low-latency BFT replicated datastore that supports democratic voting for participants.
2. We propose a novel conflict resolution protocol that is resilient to Byzantine faults. This protocol lies at the heart of PnyxDB, and leverages commutative and independent operations to ensure safety in the face of Byzantine behavior, while delivering scalability and low-latency.
3. We implemented PnyxDB and published its source code \[61\]. We evaluate our implementation against two well-known systems, BFT-SMART \[10, 11\] and Tendermint \[13\], two competitors representing alternative trade-offs in the design space. We demonstrate that our system is able to reduce commit latencies by at least an order of magnitude under realistic Internet conditions, while maintaining steady commit throughput. We also show that PnyxDB is able to scale to up to 180 replicas on a worldwide geodistributed AWS deployment, with an average latency of a few seconds.

The remainder of this paper is structured as follows. Sections 2 and 3 define our model and specify our replication protocol, alongside with properties proofs. Section 4 presents the technical choices made to implement PnyxDB. Section 5 evaluates our PnyxDB implementation. We present related work in section 6, followed by a discussion of limitations in section 7. Section 8 concludes this paper.

## 2 PnyxDB overview

### 2.1 System Model and Assumptions

We assume a system made of distributed machines (nodes) communicating through messages. Our system defines three types of roles that one node may implement independently:

- **Clients** can submit transactions, each consisting of a list of operations on a replicated key-value datastore;
- **Endorsers** are able to participate in Byzantine consensus quorums by validating and voting on clients’ transactions. Like existing decentralized ledgers, they store the whole datastore state in order to serve clients and make policy-based decisions;
- and **Observers** maintain a copy of the shared datastore, but are not able to validate transactions.

2.2 Intuition and Overview

Closed-membership Byzantine state machine replication typically rely on some form of Byzantine-tolerant consensus that ensures strong consistency \[10, 13, 34, 65\]. As a result, they unfortunately do not scale beyond a few tens of replicated nodes, due to the inherent cost of executing a Byzantine agreement protocol \[25, 49\]. One strategy to overcome this scalability barrier exploits a trusted computing base for coordination and ordering, such as Kafka or Raft in recent versions of Hyperledger \[4, 27\], but this approach weakens the security model of the protocol. Another strategy consists in using proof-of-work or proof-of-stake techniques from open-membership Byzantine ledgers \[3, 6, 32\]. These techniques are either costly or link a node’s influence to its stake in the system, two undesirable properties in many cases. In this paper we tackle scalability by weakening consistency guarantees—a strategy often used by large-scale datastores—while maintaining Byzantine Fault Tolerance (BFT).

Figure 1 gives an overview of PnyxDB’s interface and mechanisms. Clients submit transactions that are made of
operations on keys of the PnyxDB datastore. These operations are typically reads and writes, but PnyxDB can be extended to other shared objects with a sequential specification. These transactions are then broadcast to all endorser nodes, which vote for or against the transaction through an application-level voting callback. This callback provides in-system governance by allowing nodes to endorse transactions according to application-level policies. Transactions must be supported by a configurable lower threshold of a majority of correct nodes to proceed.

The properties of PnyxDB result from the novel combination of two key ingredients: leaderless quorums for scalability, and conditional endorsements for eventual consistency.

2.2.1 Leaderless quorums

PnyxDB does not use any coordinator, rotating or elected, in contrast to many existing BFT replication solutions [10, 13, 16, 34]. This choice removes a recurring performance bottleneck in the process, trading off weaker consistency guarantees for higher scalability. Transactions only need to be endorsed by a Byzantine quorum of endorsers (more than \(\frac{n+f}{2}\)) to be permanently committed to the system’s state. If two transactions commute (i.e., they contain no conflicting operations), their respective quorums can be built independently, and the transactions applied out of order, thus ensuring PnyxDB’s eventual consistency. This strategy is directly inspired from Conflict-Free Replicated Datatypes (CRDTs) [60, 63, 66] and leverages the fact that many operations in distributed datastores either commute or are independent. When this is the case, these transactions may be executed out of order on different nodes without breaking local consistency [43, 60], while allowing every correct node to eventually converge to the same global datastore state. A typical example is the popular Unspent Transaction Outputs model (UTXO) used in cryptocurrencies [28, 55] that avoids concurrency by writing to a variable only once: within this model, conflicts only occur when Byzantine nodes try to re-use an expired variable. (This problem is well-known as the "double-spending" attack.)

2.2.2 Conditional endorsements

Leaderless quorums work well for commutative transactions, but might lead to deadlocks in case of conflicts, for instance when modifying the same key with non-commutative operations. We overcome this problem with a second core mechanism: conditional endorsements. When an endorser broadcasts an endorsement, it also publishes a (possibly empty) list of transactions that must not be committed for the endorsement to be valid. (These conflicting transactions are the conditions of the endorsement.) Given a pair of conflicting transactions, all correct nodes will use the same heuristics (based on time-stamps) to decide which one to promote over the other, ensuring a consistent conflict resolution. Without additional mechanisms, conditional endorsements may however lead to an ever-growing acyclic dependency graph between transactions. We avoid this outcome by periodically triggering garbage collections (or checkpoints) using a binary Byzantine Veto Procedure (§ 5.7).

As a result of leaderless quorums and conditional endorsements, transactions proceed through the life cycle presented in Figure 2. First, a client broadcasts a transaction to endorsers. If it agrees with the transaction’s operations, an endorser node can acknowledge the transaction by broad-
casting its endorsement. If a threshold of valid endorsements is received within a transaction deadline (as defined in §3), that transaction may enter the Applicable state. A transaction in that state has enough valid endorsements, but the node is not certain that those endorsements will remain valid - because of possible future conflicts. The Applied state is an artifact introduced by the speculative execution of a transaction, when this mode is activated: in this temporary state, the system cannot yet commit a transaction but it may execute the operations on the datastore state. This optional optimization is useful to reduce global latency if the estimated probability of commit is very high. Transactions can finally transition to final states Committed—once the node is sure that the endorsement will always stay valid—or Dropped, as we will detail in the following sections.

3 The protocol
The used variables and notations are summarized in Table 1.

3.1 Transaction applicability and endorsement validity
The notion of applicable transactions (Figure 2) plays a key role in the eventual consistency of PnyxDB, and is recursively defined in terms of valid endorsements. More precisely:

- A transaction \( t \) is Applicable at node \( p_i \) if and only if there exists at least \( \omega \) valid endorsements for \( t \) at node \( p_i \), where \( \omega \) is a Byzantine quorum threshold, chosen to be larger than \( \lceil \frac{\alpha}{2} \rceil \).
- An endorsement \( e = \langle id, q, \Delta \rangle \) of a transaction \( t = \langle id, d, c \Delta, \Delta \rangle \) is Valid at node \( p_i \) if and only if every transaction \( c \) in the condition set \( c \Delta \) of \( e \) has an earlier deadline than \( t \) and is not Applicable. A transaction deadline is set by its issuer and constrained by system-wide policies to avoid excessively-large deadlines.

The interplay between these two notions drives how a transaction proceeds through the state diagram of Figure 2, and is illustrated on the scenario shown in Figure 3. In this example, Nodes \( p_1 \) and \( p_2 \) propose two conflicting transactions \( q \) and \( r \) (Figure 3a). \( q \) is at first only endorsed by \( p_1 \) and \( p_2 \). \( e_{r,1} \) denotes the endorsement of transaction \( x \) by node \( p_i \). When transaction \( r \) is broadcast, \( p_1 \) and \( p_2 \) detect a potential conflict with \( q \). which they have already endorsed, and issue conditional endorsements for \( r \). \( p_4 \) has not endorsed \( q \): it can endorse \( r \) unconditionally.

The resulting condition graph on every node at this point is shown in Figure 3b. Endorsement conditions are represented by dashed lines: for instance, \( e_{r,1} \) is valid if \( q \) is not Applicable. In Figure 3b, \( q \) has only received 2 endorsements, and is therefore not applicable under a quorum threshold of \( \omega = 3 \). \( r \) has received 3 endorsements (from \( p_1, p_2, p_4 \)), all of which are valid: \( e_{r,4} \) because its condition set is empty, \( e_{r,1} \) and \( e_{r,2} \) because \( q \) is not applicable. Transaction \( r \) is therefore Applicable, and may be speculatively executed but cannot be Committed yet as \( q \) has not been Dropped.

When a third endorsement \( e_{q,3} \) for \( q \) is finally received from \( p_3 \), the condition graph of each node changes to that of Figure 3c. At this point, the minimum number of valid endorsements is now reached for \( q \), making two endorsements for \( r \) invalid. \( q \) is now Applicable while \( r \) is no longer so.

3.2 Algorithm
The detail of PnyxDB’s workings is presented in Algorithms 1, 2, 3 and 4. Our design is reactive: endorsers and observers react to the Transaction and Endorsement messages they receive from the network. For simplicity, we do not include authentication and invariant checks. (In the following, ‘line x.y’ refers to line \( y \) of Algorithm x.)

A client starts a set of operations by broadcasting a Transaction \( \langle id, d, R, \Delta \rangle \) to nodes, with a configurable deadline \( d \) and a set of operations \( \Delta \). On receiving this Transaction

**Algorithm 1** Message callbacks at node \( p_i \)

1: **upon reception of** Transaction \( \langle id, id, R, \Delta \rangle \)
2: \( t \leftarrow \text{Transaction}(id, id, R, \Delta) \)
3: **done** \( \leftarrow \bot \)
4: \( \Rightarrow \text{Continue until no active conflicting transaction present} \)
5: **while done** \( = \bot \) **do**
6: \( \text{if CanEndorse}(t) \) **then**
7: \( C \leftarrow \{ e : e \in T_i \land \Gamma(c, \Delta, \delta) \} \)
8: \( \text{if } C = \emptyset \) **then**
9: \( \Rightarrow \text{No conflicting transaction} \)
10: \( \text{Endorse}(t, C) \)
11: \( \text{done} \leftarrow \top \)
12: **else**
13: \( \Rightarrow \text{Otherwise, not done, going back to start of while loop} \)
14: \( \text{if } \text{all } C \in C \land c \Delta \leq \text{now}() \) **then**
15: \( \Rightarrow \text{Expired conflicting transactions} \)
16: \( \text{Endorse}(t, C) \)
17: \( \text{done} \leftarrow \top \)
18: **else**
19: \( \Rightarrow \text{Unable to endorse} \)
20: \( \text{done} \leftarrow \bot \)
21: **upon reception of** Endorsement \( \langle id, j, C \rangle \)
22: \( E_i.id \leftarrow E_i.id \cup \{ \text{Endorsement}(id, j, C) \} \)
23: \( \forall t \in T_i : \text{CheckState}(t) \)

**Algorithm 2** Endorsement checks at node \( p_i \)

1: **function CanEndorse**\( t \)
2: \( \text{if } t.d \leq \text{now}() \) **then**
3: \( \Rightarrow \text{Return abort} \) \( \Rightarrow \text{Timeout} \)
4: \( \text{if } \text{State} \neq \text{not compatible with } t.R \) **then**
5: \( \Rightarrow \text{Return abort} \) \( \Rightarrow \text{Consistency} \)
6: \( \text{State} \leftarrow t.(\Delta, \text{State}) \)
7: \( \text{if } \text{State} \neq \text{does not comply to Policy} \) **then**
8: \( \Rightarrow \text{Return abort} \) \( \Rightarrow \text{Policy} \)
9: \( \text{Return OK} \)
10: **function Endorse**\( t, C \)
11: \( \text{Broadcast(Endorsement}(t.id, i, C) \}) \)
12: \( T_i \leftarrow T_i \cup t.id \)
Algorithm 3 Predicates at node $p_i$

1: function Applicable(id)
2: \quad \begin{align*}
\quad & E_{i, id}^r \leftarrow \{ e : e \in E_{i, id} \land \text{Valid}(e) \} \\
\quad & \text{return } \left\lceil \frac{|E_{i, id}^r|}{\omega} \right\rceil \geq 1
\end{align*}

3: function Valid(e)
4: \quad \text{return } \forall c \in e.C, \neg \text{Applicable}(c, id)

(receiving endorser ($T_i$ set). CHECKSTATE() ensures that the state of the datastore $State_i$ is consistent with the Applicable state of transactions (lines 4.4 and 4.7). It also triggers the COMMIT operation on transactions $q$ when there are a sufficient number of unconditional endorsements on $t$ (line 4.9). Finally, the procedure can decide to trigger checkpoints when conditions are blocking newer transactions (represented by the Old trigger, tested at line 4.15).

More specifically, once a node has received a predefined quorum $\omega$ of valid and distinct endorsements for a given transaction $t$ (implemented by the Applicable() and Valid() functions in Algorithm 3, invoked at line 4.2), CHECKSTATE() applies $tD$ if the node is configured to execute applicable transactions speculatively (line 4.7). Coming back to Figure 2, it means that the transaction moves either to the Applicable state (if the node is not speculative) or the Applied state otherwise.

We must ensure that $\omega > \left\lceil \frac{n + f}{n} \right\rceil$ to tackle Byzantine endorsements. Higher $\omega$ values allow to build stricter transaction acceptance rules, requiring up to unanimous agreement ($\omega = n$), but this comes at the cost of availability depending on Byzantine nodes to endorse transactions. (The minimum number of nodes to allow both availability and safety is $n \geq 3f + 1$ [47].)

The CHECKSTATE() function is also used to verify the validity of previously-valid endorsements because of endorsement conditions (line 3.5), potentially triggering transaction rollback(s) (line 2.4, as illustrated in Figure 3). A transaction can move back and forth from its initial Pending state to the Applicable state. It is important to note that those states
Algorithm 4 State checking at node $p_i$

1: function CheckState($t$)
2: if $\neg$Applicable($t$) then
3: if $\neg$Applied($t$) then
4: $State_i \leftarrow$ Rollback($State_i, t$)
5: else
6: if $\neg$Applied($t$) and $\neg$Speculative then
7: $State_i \leftarrow$ Apply($State_i, t$) $\triangleright$ Speculative execution
8: $\triangleright$ Endorsments that will always stay valid
9: $\Sigma_{t \in I} = \{ e \in E_{t : t}: e.C = \emptyset \}$
10: if $|\Sigma_{t \in I}| \geq \omega$ then
11: if $\neg$Applied($t$) then $State_i \leftarrow$ Apply($State_i, t$)
12: $State_i \leftarrow$ Commit($State_i, t$)
13: $T_i \leftarrow T_i \setminus \{ t \}$ $\triangleright$ Conditions that could be dropped
14: $\triangleright$ Example of checkpoint trigger for configurable delay $\delta$
15: $\hat{T} \leftarrow \{ e \in E_{t \in I, t}: e \in e.C : \text{OLD}(e) \}$
16: if $|\hat{T}| \geq 1$ then
17: $\triangleright$ StartCheckpoint($T$)
18: return $\neg$Applicable($t$) $\land$ $t.d < (\text{now}(t) - \delta)$

Algorithm 5 Checkpoint at node $p_i$

1: function StartCheckpoint($T$)
2: $\triangleright$ Broadcast(Checkpoint($T$))
3: upon reception of Checkpoint($T$)
4: $c \leftarrow \begin{cases} 0 & \text{if } \exists t \in T : \text{Applicable}(t) \land \text{Committed}(t) \\ 1 & \text{otherwise} \end{cases}$
5: decision $\leftarrow$ BVP($T, c$)
6: if decision $= 1$ then $\triangleright$ Cleanup
7: $T_i \leftarrow T_i \setminus T$ $\triangleright$ Drop transactions
8: $\forall t \in T : E_{t, i} = \emptyset$ $\triangleright$ Forget endorsements of dropped transactions
9: $\forall t \in T_i, e \in E_{t, i} : e.C = e.C \setminus T$ $\triangleright$ Forget conditions
10: $\forall t \in T_i \cup T : \text{Checkpoint}(T)$

When a node decides to start a checkpoint, it triggers a BVP instance with a Checkpoint proposal (line 5.5), a set of transactions representing a cut of their graph of conditions. Each proposal aims at removing old transactions that block newer transactions from being committed. Informally, a proposal might be as simple as "transaction $t$ will never be applicable, drop it". During the procedure, correct nodes are expected to propose 0 ("Veto") if and only if they hold evidence that the checkpoint proposal is wrong (line 5.4). (Such nodes must submit this evidence in the form of signed endorsements.) Two checkpoint results are possible per invocation: (1) If the final decision is 1, correct nodes can prune their local graph of conditions according to the confirmed proposal (lines 5.7-5.9); (2) otherwise, some correct nodes have reasons for blocking the checkpoint proposal. After having added the evidence(s) to their graph of conditions, correct nodes can discard this checkpoint instance.

In our example from figure 3b, if the BVP decision on the proposal "drop $q$" is 1, then every node can confidently drop $q$ and remove $q$’s condition on the endorsements $e_{r,i}$, thus effectively committing $r$. On the contrary, if the BVP decision is 0, correct nodes can expect an evidence going against the proposal: for instance, node $p_3$ can broadcast $e_{q,3}$ again. This allows nodes to progress, finally triggering the commit of $q$ and the drop of $r$ for every node. We discuss and evaluate the overhead of this checkpoint procedure in § 5.7.

### 3.3 Checkpointing

In many cases, we expect that a node can conclude from received endorsements that the Applicable predicate has reached a final state (true or false) by analyzing the transaction’s graph of conditions. When complex dependencies arise between endorsements and transactions, some transactions might however interlock. As an example in Figure 3, nodes cannot know whether $r$ must be committed before receiving $e_{q,3}$. To cope with this issue and ensure both liveness and consistency, we use a simple checkpoint sub-protocol (Algorithm 5) to prune the condition graph and unblock transactions. This sub-protocol builds upon an underlying Byzantine Veto Procedure (BVP) in which each node $p_i$ proposes a choice $c_i \in \{0, 1\}$ and decides a final value $d_i$. BVP is a Byzantine-tolerant version of the Non-Blocking Atomic Commitment (NBAC) protocol [5], and is expected to satisfy the following properties with eventually-synchronous communications: 1) Termination: every correct node eventually decides on a value; 2) Agreement: no two correct nodes decide on different values; and 3) Validity: if a correct node decides 1, then all correct nodes proposed 1 (equivalently, if any correct node proposes 0, then a correct node decides 0). We return to the implementation details of BVP in section 4.

### 3.4 Eventual consistency: proofs

We first show that every correct node eventually obtains the same set of applicable transactions. We then show that transactions entering the final committed and dropped states will stay in these states for every correct node.

**Lemma 3.1 (Acyclic conditions).** Let $q, r$ be two transactions with $q.d \leq r.d$. There cannot be any ENDORSEMENT($q.id, i, C$) broadcasted by a correct node $p_i$ with $r \in C$.

**Proof sketch.** In our algorithm, the only case where conditional endorsements are broadcasted is at line 1.15. For this line to be executed, CANENDORSE($q$) must have returned OK. Hence, per line 2.2, we must have $q.d >$ now(). Given line 1.13, every element $r$ of $C$ must fulfill $r.d \leq$ now(). If we
assume that local operations execute instantaneously, the value of \( p_i \)'s local clock now shall be the same in the two constraints. We have \( \forall r \in C, r.d < q.d \).

We can use the result of this lemma to filter incoming endorsements at each node, and detect Byzantine behavior. In the following, we suppose that every malformed endorsement has correctly been filtered by correct nodes.

**Proposition 3.2** (Termination). Assuming the BVP protocol terminates, every proposed functions and callbacks terminate.

*Proof sketch.* This is trivial for functions CANENDORSE and ENDORSE in Algorithm 2. When receiving a TRANSACTION message at line 1.1, a node may execute the while loop (lines 1.5-1.20) several times if conflicts are detected. A node is, however, guaranteed to exit the while loop when CANENDORSE returns false because the transaction's deadline is over (timeout clause). Upon reception of the messages ENDORSE (line 1.21), and CHECKPOINT (line 4.3), and for the function CHECKSTATE (line 4.1), the termination is conditioned by the termination of the Binary Veto Procedure (BVP) and the APPLICABLE predicate.

BVP terminates by assumption. The case of APPLICABLE is somewhat more involved, as the functions APPLICABLE and Valid recursively call each other. Every APPLICABLE call of transaction \( r \) will trigger a finite number of Valid calls on endorsements \( e \in E_{i,r,\text{id}} \) received for \( r \) (line 3.2). Each Valid call will in turn call a finite number of APPLICABLE call for every \( c \in c.C \) (line 3.5). Because of Lemma 3.1, we know that for two transactions \( q, r \) with \( q.d \leq r.d \), there can be no ENDORSEMENT \( q, \text{id}, i, C \) with \( r \in C \) in any \( E_{j,q,\text{id}} \) set.

This property implies that the recursion between the two predicates will call APPLICABLE with transactions ordered by decreasing deadline \( q.d \), thus eliminating any loop.

**Lemma 3.3** (Local safety). Let \( p_i \) be a correct node and \( q, r \) two transactions with \( q \neq r \):

\[
\Gamma(q, \Delta, r, \Delta) \Rightarrow \neg \left( \text{APPLICABLE}(q) \land \text{APPLICABLE}(r) \right)
\]

*Proof sketch.* Without loss of generality, we suppose \( q.d \leq r.d \) and that APPLICABLE(q) and APPLICABLE(t) hold at correct node \( p_i \). Let us note \( C_0 \) a condition set such that \( q \notin C_0 \). Given Lemma 3.1 and instruction 3.5, the only endorsements for \( q \) and \( r \) that are valid for \( p_i \) are \( E_{i,q,\text{id}} = \{ \text{ENDORSEMENT}(q, j; C_0) \} \) and \( E_{i,r,\text{id}} = \{ \text{ENDORSEMENT}(r, j; C_0) \} \), for any endorser \( p_j \). According to line 3.3, we must have \( \omega \leq |E_{i,q,\text{id}}| \).

Since correct nodes cannot send both kind of endorsements per algorithm 1, we also must have \( \omega \leq |E_{i,q,\text{id}}| \leq n - |E_{i,q,\text{id}}| + f \leq n - \omega + f \), or more simply \( \omega \leq \frac{n - f}{2} \). This leads to a contradiction with \( \left\lceil \frac{n - f}{2} \right\rceil + 1 \leq \omega \).

**Lemma 3.4** (Eventual consistency). After enough time, for two correct nodes \( (i, j) \), if APPLICABLE\((q)\) holds at node \( p_i \), then APPLICABLE\((q)\) must hold at node \( p_j \).

*Proof sketch.* Given the reliable broadcast and the eventual synchrony assumptions, every correct node will receive the same set of endorsements. Since we have \( E_{i,q,\text{id}} = E_{j,q,\text{id}} \), the value of APPLICABLE\((q)\) must be the same for \( p_i \) and \( p_j \) nodes before checkpointing. During every checkpoint, endorsement sets and conditions may be modified by lines 5.8 and 5.9. Giving the agreement property of the BVP, every correct node will prune their graph of conditions according to the confirmed proposal, or no node will do.

**Proposition 3.5** (Durability). The proposed algorithm ensures that if a transaction \( t \) is COMMITTED (resp. DROPPED) at a correct node \( p_i \), it will stay in this state and will eventually be COMMITTED (resp. DROPPED) for every other correct node.

*Proof sketch.* The COMMITTED state is triggered in the CHECKSTATE routine when a transaction obtains a number of unconditional endorsements \( k \geq \omega \) (line 4.9), either after receiving a new endorsement or after a successful checkpoint. With the same arguments than Lemma 3.4, we know that in the first case every correct node will COMMIT \( t \). The second case is covered by the underlying BVP during a checkpoint: after a successful checkpoint (\( \text{decision} = 1 \)), every correct node prunes its conditions graph in the same way, thus eventually triggering COMMIT operations on every correct node (resp. DROP, line 5.7). Per line 4.4, operating a ROLLBACK on a transaction \( t \) is only possible if APPLICABLE\((t)\) does not hold anymore. Since we know that at least \( \omega \) unconditional endorsements for \( t \) have been received at this point, the only way that the predicate would not hold is due to a successful checkpoint on \( t \in \bar{T} \), with the "endorsement-forgetting" operation depicted at line 5.8. However, given line 5.4, if at least one correct node has COMMITTED \( t \), no further checkpoint on \( t \in \bar{T} \) can return a decision of 1. No DROPPED transaction could become APPLICABLE again in correct nodes due to the pruning of endorsements after a successful checkpoint.

It follows from Lemmas 3.3 and 3.4 that no two conflicting operations can be committed in different orders by two correct nodes, thus ensuring eventual consistency with Proposition 3.5. Another core feature of our algorithm is the ability for correct nodes to reject any transaction without giving their reasons, as underlined in line 2.8. We formally define this property as the system’s fairness.

**Proposition 3.6** (Fairness). If no majority of correct nodes \( k > \left\lceil \frac{n - f}{2} \right\rceil + f \) endorsed a transaction \( t \), APPLICABLE\((t)\) will never hold at any correct node.

*Proof sketch.* Let \( \epsilon \) be the number of endorsements for \( t \). We must have \( \omega \leq \epsilon \leq \left\lceil \frac{n - f}{2} \right\rceil + f \) for \( q \) to be APPLICABLE, which is impossible due to our definition of \( \omega \).
4 Implementation

We have implemented PnyxDB in Go [61, 67]. In this section, we describe our implementation of node authentication using a web of trust, along with practical solutions for the assumed algorithm primitives, namely the Reliable Broadcast and the Byzantine Veto Procedure (BVP).

Our technical choices were driven by the common size of consortia and state of the art cryptography techniques. Hence, we decided to target a scale of several hundreds to thousands of nodes per network, excluding clients.

4.1 Web of trust and policy files

Nodes need to be authorized to participate in a closed consortium. The Hyperledger Fabric consortium blockchain proposes a centralized approach, where a single authority gives cryptographic certificates to network members [4]. However, a corrupted authority may introduce a large number of malicious nodes in the network, potentially breaking the assumption on the maximum number of faulty nodes $f$.

Our implementation relies instead on a web of trust and policy files, inspired from PGP [2]. The web of trust is used to link nodes’ identities with their public key, providing a sound authentication mechanism. Our implementation supports several cryptographic authentication schemes, and uses the recognized fast ed25519 procedure by default [9].

Nodes need to know the identities of endorsers, along with useful metadata such as authorized operations and default network parameters. We use a universal policy file for this, and we expect nodes to agree on the content of this policy file: this is similar to the distribution of a common genesis file required by a number of existing BFT systems [4, 10, 13]. Classic PnyxDB transactions could be used to update the universal policy in a consistent and democratic way, for instance as done in the Tezos Blockchain [33].

4.2 Reliable broadcast and recovery

A Byzantine-resilient reliable broadcast is required in PnyxDB to ensure that correct nodes will eventually receive every transaction and endorsement, possibly out-of-order. Such an algorithm was proposed by Bracha [12], but it has a message complexity of $O(n^2)$, which makes it impractical for our targeted scale. Based on current public and consortium blockchains implementations [7], we propose a probabilistic gossip algorithm as our reliable broadcast primitive, where each node communicates only with a small number of neighbors to lower the total message complexity. Such algorithms are known to disseminate information with a logarithmic number of messages and are used in popular BFT public and consortium blockchain networks. We selected Gossip-Sub [38] from the libp2p project as our gossip broadcast algorithm. Libp2p is a popular set of libraries for peer-to-peer communication, that targets gossip networks of 10000 nodes with practical Byzantine Fault Tolerance. It comes with standard mechanisms for inter-node communication and authentication that fulfill our specifications, and supports our default ed25519 authentication scheme.

Using a gossip algorithm as our broadcast primitive inherently introduces uncertainty in the reliability of the broadcast [35]. We propose to complement this probabilistic broadcast with retransmissions and state transfers: with very low probability, some nodes may not receive a given message. In that case, they may later ask for a retransmission of a transaction or endorsements related to a transaction. After long failures (such as power outage or network partition), some nodes may have missed a large number of messages and become out-of-sync with the remainder of the network. At this point, retransmitting every message becomes prohibitively expensive: that’s why each node is able to synchronize its complete state from its neighbors. We rely on the web of trust (§ 4.1) to retrieve the state from neighbors that are sufficiently trusted by the out-of-sync node. (In our implementation, a configurable quorum of identical values must be received before re-synchronizing one node’s state.)

4.3 Binary Veto Procedure

The main issue with our endorsement scheme is that Byzantine nodes can arbitrarily delay their endorsements. To cope with that limitation in a practical way, we propose a BVP implementation in Algorithm 6, based on periodic health probes of the gossip mechanism in our eventually synchronous network.

Definition 4.1. The maximum gossip broadcast latency, denoted $\tau$, is the maximum possible delay from a message broadcast to its delivery by every correct node.

We make the following two assumptions: every correct node $p_i$ is able to estimate ($A$) $\hat{\tau}$ such as $\hat{\tau} \geq \tau$ and ($B$) $\delta_{i,j}$ the relative clock deviation for any endorser $p_j$. In practice, it is possible to obtain these two values from passive round-trip measurements in the gossip network. (We note that under asynchrony, $\tau = \delta_{i,j} = \infty$.) With that additional knowledge, each correct node can estimate locally the earliest possible sending time of a message, and discard the messages published after a specific deadline (line 6.6). This simple approach is sound during periods of synchrony, but may introduce significant delays due to use of a deadline. As BVP is not the main contribution of this paper, we leave the optimization of this primitive to future work.

Proposition 4.2. Algorithm 6 satisfies the properties of BVP.

Proof sketch. Termination is trivial in eventually synchronous networks (line 6.4). Per assumptions $A$ and $B$ every correct node will compute the same value for ‘deadline’ at line 6.5. By line 6.6, no endorsement for $i \in T$ sent after this shared deadline can be accepted. Thanks to assumption $A$, endorsements sent before ‘deadline’ are delivered before (deadline + $\hat{\tau}$) > (deadline + $\tau$), leading to the same set of endorsements for $\hat{T}$ being received for every correct node after
Algorithm 6 Byzantine Veto Procedure (BVP) at node $p_i$

1: function BVP($T$, $c_j$)  
2:    if $c_j = 0$ then $\triangleright p_i$ is vetoing the decision to drop $\bar{T}$  
3:        return 0  
4:    wait until $\bar{T} < \infty \land \forall j, \delta_i j < \infty$ $\triangleright$ Wait for synchrony  
5:    deadline = $\max(t, d, t \in \bar{T}) \div \max(\delta_i j)$  
6:    Stop delivering endorsements for $t \in \bar{T}$ sent after deadline  
7:    wait until either  
8:        $\exists t \in \bar{T} \colon \text{APPlicable}(t)$ then return 0  
9:        now() > deadline + $\bar{T}$ then return 1

(deadline + $\bar{T}$). This implies the Agreement property given the decisions of lines 6.8 and 6.9. A correct node proposes a veto if and only if at least one transaction in $\bar{T}$ is Applicable: Validity follows from the properties of Applicable. □

5 Evaluation

5.1 Experimental setup

We tested our implementation of PnyxDB in two different environments: an emulation setup, and a global network using Amazon Web Services (AWS). The emulation was performed on a server able to sustain several hundreds of nodes with the Mininet [51] network emulation tool (48 threads of Intel(R) Xeon(R) Gold 6136 CPU at 3.00GHz with 188GB of RAM). We used Mininet [51] to isolate nodes from each other and to simulate real network latencies. We drew latency values from an exponential distribution law with an average of 20 ms per link. Every node’s clock was shifted by a random amount in the $[-5, 5]$ seconds interval between the reference time to simulate a relatively small asynchrony between network participants. For the BVP algorithm, we chose the conservative value $\bar{T} = 10$ seconds: this leads to a practical checkpoint timeout of 20 second. Each experiment was run 40 times, taking the average as the final result.

5.2 Baseline

To compare our work with available solutions, we executed the same experiments with BFT-SMArt v1.2 server [10] and a Tendermint v0.32.5 voting application [13]. BFT-SMArt has been recognized as an efficient Java library for the BFT problem, and is being added in a number of applications, including Hyperledger Fabric [4, 65]. Tendermint is a BFT Consensus mechanism based on a permissioned blockchain with a leader-based algorithm; its implementation relies on a gossip broadcast primitive, like PnyxDB. Both implementations allow custom application logic to be executed during consensus; this empowered us to emulate a voting behavior within these two existing solutions. The two systems are leader-based, but their consensus choices are quite different: while BFT-SMArt rely on a single leader as long as it reports no issue to avoid costly view changes, Tendermint leaders are selected in a round-robin fashion with each leader batching transactions into blocks. BFT-SMArt is based on a fully connected mesh topology whereas Tendermint nodes communicate via gossip. The two baselines offer a different trade-off than our proposal, targeting stronger consistency guarantees but with no native democratic capabilities. (For fair comparison, we configured Tendermint with the “skip timeout commit” option to optimize its commit latency.)

5.3 Network size ($n$)

This first experiment measures the latency from a single transaction submission to its commit by every node. We set the required number of endorsers to $\omega = \left\lceil \frac{1}{3} n \right\rceil + 1$ and increased $n$ from 10 to 320. For completeness, we note that setting $\omega = n$ (unanimous agreement) had the effect of slightly increasing the latency, since nodes had to wait for more votes before committing any transaction. As denoted by the $\otimes$ symbols, we were unable to complete some large network experiments for BFT-SMArt ($n \geq 80$) and Tendermint ($n \geq 320$) in our testbed, due to extremely high CPU and network load. Figure 4 shows that PnyxDB outperforms existing implementations for small and large networks by an order of magnitude.

5.4 Number of clients

To measure the effect of client scaling, we configured a various number of clients to submit transactions at an average rate of 2 transactions per second, as controlled by a Poisson point process. The transactions were generated using the Yahoo! Cloud Serving Benchmark (YCSB) [18], a well recognized non-relational datastore testing tool that allowed us to vary the ratio of conflicting transactions, and hence the contention level on PnyxDB. We customized the benchmark workload to create only update transactions to a set of 100 keys, from which updated keys were selected using a uniform distribution. This relatively low level of contention reflects a number of real workloads, but we present some results for higher contention rates in § 5.5. Additionally, each tested network required a quorum of $\omega = 7$ endorsers among
We increased the level of transaction contention by rising \( n \) with realistic network latencies. PnyxDB performed well for 10 clients for those experiments, leading to an average of 20 records selected by YCSB. PnyxDB can scale with the number of transactions per second (tx/s). We also tested PnyxDB with a probability of conflicts with high contention levels. We kept note that the artificially-added clock shift tend to increase the conflicting transactions for PnyxDB in our setup. We also implemented speculative execution (§ 3) to compare its performance against classic commit latencies as presented in the previous subsections. First, let us recall that once a transaction is committed, it is dropped if a significant number of nodes \((n - \omega + 1)\) are unable to endorse the transaction before its deadline. As expected, the contention level has a direct impact on the ratio of dropped transactions. For low levels of contention (up to 3%), almost no transaction is being dropped thanks to conditional endorsements. For higher levels of contention (up to excessively high levels), the drop ratio stays well below the worst case scenario.

### 5.5 Effect of contention

We increased the level of transaction contention by rising a “hotspot” hit probability (Figure 6), one option provided by YCSB. This parameter has an immediate effect on the probability that (at least) two transactions ask to update the same datastore record around the same time, thus becoming conflicting transactions for PnyxDB in our setup. (We also note that the artificially-added clock shift tend to increase the probability of conflicts with high contention levels.) We kept 10 clients for those experiments, leading to an average of 20 transactions per second (tx/s). We also tested PnyxDB with a slower rate of 10 transactions per second for comparison. The worst case scenario would be that every transaction hitting the hotspot is eventually dropped: in PnyxDB, a transaction is dropped if a significant number of nodes \((n - \omega + 1)\) are unable to endorse the transaction before its deadline. As expected, the contention level has a direct impact on the ratio of dropped transactions. For low levels of contention (up to 3%), almost no transaction is being dropped thanks to conditional endorsements. For higher levels of contention (up to excessively high levels), the drop ratio stays well below the worst case scenario.

### 5.6 Speculative execution

We implemented speculative execution (§ 3) to compare its benefits against classic commit latencies as presented in the previous subsections. First, let us recall that once a transaction is applied by speculative execution, there is no guarantee that it will eventually become committed. However, we can expect that the probability of rolling back to a non-applicable (or dropped) state stays very low (Figure 8).

Figure 7 shows the average latency observed for weak and strong guarantees with an increasing level of concurrency (resp. applied and committed states). Speculative execution benefits are clearly visible for any level of contention, improving latency by up to 50%, a boost that could clearly benefit applications that only require weak guarantees. This improvement came with no additional cost, since at most 0.03% of speculative executions had to rollback. In the classic configuration, the commit latencies can be severely affected by pending checkpoints, leading to less-predictable performance. As shown in Figure 7, the commit latency is clearly more foreseeable in the speculative execution mode, staying stable despite increasing contention. This observation confirms the positive impact of speculative execution.
Table 2. Summary of comparison with emulated network latencies and 10 clients for a total of 1000 transactions. This is the average over 40 experiments with 10 nodes tolerating up to 3 Byzantine faults ($n = 10$, $\omega = 7$).

<table>
<thead>
<tr>
<th></th>
<th>Average latency (ms)</th>
<th>$95^{th}$ perc. latency (ms)</th>
<th>Throughput (tx/s)</th>
<th>Drop rate</th>
<th>Disk usage per node (MB)</th>
<th>Transfer duration (s)</th>
<th>Exp. duration (s)</th>
<th>Bandwidth per node average / max (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFT-SMAr [10]</td>
<td>89</td>
<td>170</td>
<td>5.68</td>
<td>0%</td>
<td>-</td>
<td>36</td>
<td>230</td>
<td>0.16 / 0.21 MB/s</td>
</tr>
<tr>
<td>Tendermint [13]</td>
<td>1.7</td>
<td>3.9</td>
<td>17.0</td>
<td>9.3%</td>
<td>26 MB</td>
<td>26 MB</td>
<td>65</td>
<td>0.40 / 1.40 MB/s</td>
</tr>
<tr>
<td>PnyxDB (ours)</td>
<td>0.15 $\div 11$</td>
<td>0.16 $\div 24$</td>
<td>18.6 $\div 9.4$</td>
<td>2.3%</td>
<td>1.4 MB</td>
<td>20 MB</td>
<td>71</td>
<td>0.28 / 1.27 MB/s</td>
</tr>
</tbody>
</table>

Figure 7. Difference between classic and speculative operation execution latencies. There is a significant gain with speculative execution for high levels of contention.

Figure 8. Top: experimental ratio between the number of rollbacks and the number of transaction applications. Bottom: number of checkpoints executed across 1000 transaction submissions.

Figure 9. Worldwide AWS deployment: commit latency with increasing number of endorser nodes. $\omega$ was set to 70% of endorsers.

5.7 Impact of checkpoints for conflict resolution

Algorithms 4 and 5 suggest that checkpoints must be triggered immediately when a single transaction could be dropped by a node. This is inefficient, since the proposed checkpoint procedure is costly in terms of bandwidth. Thus, we added a pooling mechanism to limit the number of checkpoints: by aggregating transactions before proposing checkpoints, each node optimizes its bandwidth while slightly increasing the commit latencies of conflicting transactions. In our evaluation, at most 10 checkpoints were triggered (Figure 8). Given that checkpoints happen mainly in times of high levels of contention, we can conclude that their number is still practical, thanks to our pooling optimization.

The longest path measured in the graphs of conditions was of length 34, requiring less than 1 millisecond to be processed. This observation indicates that our conditional endorsement scheme is scalable and practical in terms of complexity. We note that non-conflicting transactions were not affected and continue to be committed even when the hotspot probability is high. This is not the case for the other baselines, where transactions are totally ordered by successive leaders.

5.8 Large scale experiments

To assert PnyxDB’s scalability in a global setting, we used Amazon Web Services (AWS) t2.micro EC2 instances to deploy nodes uniformly in 6 AWS regions (Table 3). During our experiments, we observed a steady maximum inter-region round-trip time of 315 ms between São Paulo and Sydney regions. Even under these conditions, PnyxDB managed to commit transactions with an average latency lower than 2 seconds for networks up to 180 nodes (Figure 9).

Table 3. Worldwide AWS deployment: inter-region round-trip time (May 6, 2019). Nodes were evenly sharded between regions.

<table>
<thead>
<tr>
<th>(in ms)</th>
<th>Virginia</th>
<th>São Paulo</th>
<th>Paris</th>
<th>Frankfurt</th>
<th>Sydney</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>61</td>
<td>194</td>
<td>138</td>
<td>147</td>
<td>149</td>
</tr>
<tr>
<td>Virginia</td>
<td>147</td>
<td>79</td>
<td>88</td>
<td>315</td>
<td>292</td>
</tr>
<tr>
<td>São Paulo</td>
<td>223</td>
<td>226</td>
<td>10</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>Paris</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frankfurt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 Related work

There exists a large number of consensus proposals for blockchain-like applications [7, 30]. The consensus problem is tackled very differently in public permissionless systems compared to permissioned consortium systems. For public systems, there have been many efforts towards Proof-Of-
Stake consensus: a small subset of participants are pseudorandomly selected to lead the consensus for a specific round, based on their account balance (i.e. stake in the network). Recent proposals rely on a multiparty coin-flipping protocol for leader selection [40], or propose a probabilistic method using verifiable random functions [32]. These proposals are said to be easily vulnerable to Sybil attacks [22] since anyone can participate. Proof-Of-Work schemes are still considered to be the safest, and main public networks still use it extensively. More efficient algorithms have been proposed to provide more fairness to small devices [59], less communication rounds [3] or more useful computations [6].

In this paper we focus on permissioned systems where participants are known in advance. This path allows for more flexibility in the choice of the threat model and the trust assumptions. RSCoin [20] relies on a trusted central bank and on distributed sets of authorities for improved scalability. Similarly, the Corda platform [36] puts trust in small notary clusters running consensus algorithms like BFT-SMArt [10]. In its current version, Hyperledger Fabric [4, 37] also requires that a centralized ordering service is trusted by every party. These solutions, while giving good scalability promises, rely on central points of trust, that if manipulated by a malicious actor would break the entire system. The common assumption is that such entities would be legally accountable through audits: to remove that assumption, Sousa et. al. [65] proposed to replace the current Kafka ordering service by BFT-SMArt in Hyperledger Fabric. PnyxDB leverages a web of trust to ensure good scalability and node recovery, while avoiding central points and elected leaders. We believe this makes our proposal more robust to corruption and malicious manipulation.

Other consortium systems have also been proposed [13, 19, 29, 62]. Randomization techniques have been used to solve asynchronous BFT consensus [50, 52, 53], among which BEAT [24] that suggests to rely on recent cryptographic primitives. Such systems usually require that correct nodes present deterministic execution for consistency [14, 69]. By comparison, PnyxDB relies on a consensus algorithm specifically designed for non-deterministic democratic decisions, and exploits operations commutativity in a similar way than [36, 57, 60, 64]. In similar AWS deployments, BEAT reports a latency of around 500 ms for 6 nodes and more than 1 minute for 92 nodes, while PnyxDB proposes 130 ms and less than 1 second, respectively [24].

Speculative execution has been proposed in BFT consensus to reduce latencies [23, 31, 41, 45, 64]. We considered this optimization to speed-up our state synchronization algorithm while achieving extremely low rates of rollbacks. Finally, note that some vote schemes have been proposed [8, 39], but they apply only for non-Byzantine fault models.

7 Discussion

PnyxDB has been designed to work with state-of-the-art networking techniques. However, some elements can affect its liveness.

Invalid deadline A client may submit a transaction with a very low or high deadline relative to the absolute time. The first case is handled by the endorsement conditions mechanism, but nodes may block in the second case. Bounds on deadlines would be a simple countermeasure to filter incoming transactions and avoid endorsing transactions with out-of-bounds deadlines [17].

Conflicting transaction flooding A rogue client could send many simultaneous conflicting transactions, such that it will be hard to reach a single quorum agreement within the transactions deadline. This attack will not break the safety, but the system may drop transactions, with a large number of checkpoints being handled. A solution would be to isolate the responsible node and rate-limit it.

Query drops As underlined in our evaluation, query drops are expected during contention. This behavior is very common in classic and BFT replicated databases [45, 58], and each client could easily propose several transactions until one is finally committed. It is however clear that PnyxDB has been designed mainly for commutative workloads, as commonly seen in modern distributed applications.

Checkpoint with asynchrony Our BVP implementation waits for good network conditions before allowing dropping transactions. This is a safety requirement, given that Byzantine nodes could delay their endorsements indefinitely under asynchrony. An interesting property is that non-conflicting transactions are always allowed to proceed, independently of pending checkpoints.

Other optimizations We did not test batching of transactions to increase throughput [24, 41, 62, 64, 65]. We focused in this work on latency optimization, hence we believe that transaction batching is an orthogonal optimization that may further increase PnyxDB throughput.

8 Conclusion

In this paper, we presented PnyxDB, a scalable eventually-consistent BFT replicated datastore. At its core lies a scalable low-latency conflict resolution protocol, based on conditional endorsements. PnyxDB supports nodes having different beliefs and policy agendas, allowing to build new kinds of democratic applications with first-class support for non-conflicting transactions. Compared to popular BFT implementations, we demonstrated that our system is able to reduce commit latencies by an order of magnitude under realistic conditions, while ensuring steady commit throughput. In particular, our experimental evaluation shows that PnyxDB is capable of scaling to up to hundreds of replicas on a geodistributed cloud deployment.
References


blockchain protocol. In CRYPTO. 357–388. DOI: http://dx.doi.org/10.1007/978-3-319-63688-7_12


[54] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In SOSP. DOI: http://dx.doi.org/10.1145/2517349.2517350


