



**HAL**  
open science

# 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. The 39th IEEE International Symposium on Reliable Distributed Systems (SRDS '20), Sep 2020, Shanghai, China. 10.1109/SRDS51746.2020.00023 . hal-02355778v2

**HAL Id: hal-02355778**

**<https://hal.science/hal-02355778v2>**

Submitted on 24 Sep 2020

**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<sup>1,2</sup>, Christoph Neumann<sup>1</sup>, François Taïani<sup>2</sup>

<sup>1</sup>InterDigital, <sup>2</sup>Univ Rennes, Inria, CNRS, IRISA

*{firstname.surname}@interdigital.com, {firstname.surname}@irisa.fr*

## Abstract

Byzantine-Fault-Tolerant systems for closed consortia have recently attracted a growing attention notably in financial and supply-chain applications. Unfortunately, most existing solutions suffer from substantial scalability issues, and lack self-governance mechanisms. In this paper, we observe that many workloads present little concurrency, and propose PnyxDB, an eventually-consistent Byzantine Fault Tolerant replicated datastore that exhibits both high scalability and low latency. Our approach hinges on conditional endorsements that track conflicts between transactions. 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 Tolerance (BFT) has attracted much attention over the last two decades [1, 2, 3, 4, 5, 6], and has now moved into the public spotlight following the rise of blockchain platforms. BFT systems typically rely on advanced replication protocols to ensure consistency between their replicas and withstand arbitrary failures and malicious behavior. Unfortunately, traditional BFT replication protocols struggle to scale beyond a few tens of replicas [7], 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 [8, 9, 10, 11, 12], alternatives to proof-of-work such as proof-of-stake [13], or assumed access to a trusted third party providing strong ordering guarantees [14]. All these strategies are however fraught with limitations: existing leaderless protocols rely either on clients for consistency checks [8] (increasing computing overhead) or on the availability of strong coordination mechanisms, such as an atomic broadcast primitive [9, 14]; proof-of-stake requires financial incentives and could allow monopolies by linking a node's influence to its stake in the system; and trusted third parties limit the applicability of such solutions to well-controlled environments.

Compounding these limitations, all above approaches are ill-equipped to support *in-system governance*, a growing requirement for applications involving independent organizations [15]. Although most of these solutions rely on internal vote or quorums, these mechanisms are not exposed to applications. As a result, individual nodes cannot implement application-defined policies to endorse or reject transactions without additional costs and complexity. This is problematic, as application-level voting capabilities are key to decentralized BFT applications involving independent participants who need to balance conflicting goals and shared interests [16]. Examples of such governance concerns include basic membership management, access control, resource allocation, and policy administration. In all these examples, different parties are likely to pursue different agendas, and need to be able to influence the final decision according to their own application-defined policies and beliefs [15, 17, 18].

In this paper, we address these challenges using a radically different line of attack: we borrow a popular strategy from non-Byzantine distributed datastores [19, 20], 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 participants to support or reject proposed transactions according to application-defined policies.

Our proposal leverages the long-observed fact that many workloads exhibit a lot of independent operations [11, 21] that can be executed out of order without compromising the eventual convergence of all correct nodes. We exploit these independent operations through a *modified Byzantine Quorum protocol* [1] that ensures the safety and agreement of our system. 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*<sup>1</sup>, a scalable low-latency BFT replicated datastore that supports democratic voting.
2. We propose a novel conflict resolution protocol that leverages independent operations to tolerate Byzantine behavior, while delivering scalability and low-latency.
3. We evaluate PnyxDB against two open-source systems, BFT-SMART [3] and Tendermint [22], 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.

---

<sup>1</sup><https://github.com/technicolor-research/pnyxdb>

In the following, we first define our model and specifies our replication protocol (Sec. 2 and 3). Sections 4 and 5 then present and evaluate PnyxDB implementation. Related work is detailed in section 6, followed by a discussion of PnyxDB limitations in section 7. We provide some key properties’ proofs in Appendix A.

## 2 PnyxDB overview

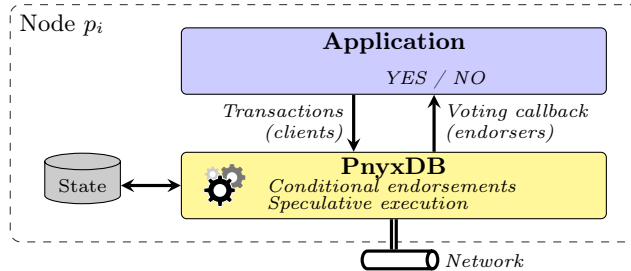
### 2.1 System Model and Assumptions

We assume a system made of distributed machines (*nodes*) communicating through messages. Our system defines two types of roles that one node may implement independently. **Clients** submit transactions, each consisting of a list of operations on a replicated key-value datastore. **Endorsers** participate in Byzantine consensus quorums by validating and voting on clients’ transactions. As in existing decentralized ledgers, they store the whole datastore state. Each system contains a known number  $n$  of endorsers, of which a maximum of  $f$  can act as *Byzantine*. Byzantine nodes are allowed to ignore the protocol specification occasionally or completely, and they can collude to create more sophisticated attacks. Non-Byzantine nodes are said to be *correct*.

We assume we have access to a *reliable BFT broadcast primitive* with the following property: *if one message is delivered to one correct node, every correct node will eventually receive that message* [23]. In our implementation, we rely on *eventually synchronous networks* as detailed in § 4.2: informally, we assume that networks alternate between periods of synchrony and asynchrony. For transaction liveness, we add the assumption that correct nodes are able to detect periods of synchrony (§ 4.3). Cryptographic signatures are used to verify nodes’ identity and authorizations. We make the standard assumptions that participants cannot break these signatures, and that they know each other beforehand. (In the parlance of distributed ledgers, our system is *permissioned*.)

### 2.2 Intuition and Overview

Closed-membership Byzantine state machine replication typically rely on some form of Byzantine-tolerant consensus that ensures strong consistency [3, 6, 22, 24]. 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, 26]. 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 [27, 14], 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 [28, 13]. Unfortunately, these techniques are either costly or link a node’s influence to its stake in the system, two properties that can be undesirable in closed-membership deployments that do not have a financial dimension. In this paper we tackle scalabil-



**Figure 1:** Overview of PnyxDB: the application submits transactions to be executed on shared state and polls the application back for transaction approval before creating conditional endorsements.

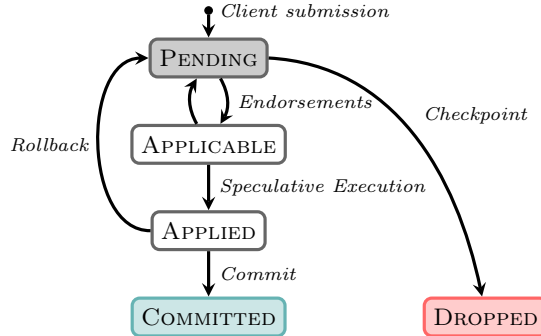
ity by weakening consistency guarantees—a strategy often used by large-scale datastores—while maintaining Byzantine Fault Tolerance.

Figure 1 provides an overview of PnyxDB’s design. 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 [2, 3, 6, 22]. This choice removes a recurring performance bottleneck, 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) [29, 20] 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 [29, 30], 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 that avoids concurrency by writing to



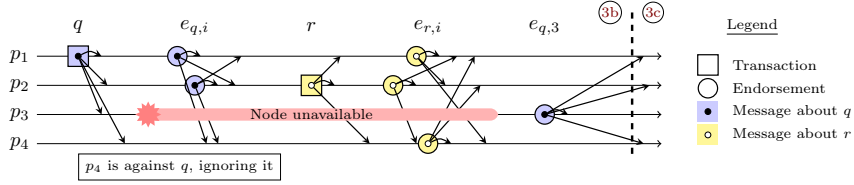
**Figure 2:** Transaction state diagram, as viewed by a node. From the PENDING state, a transaction evolves either to DROPPED or COMMITTED given received messages. DROPPED and COMMITTED are eventually consistent across all nodes. In contrast, PENDING, APPLICABLE and APPLIED are intermediate states local to each node.

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.)

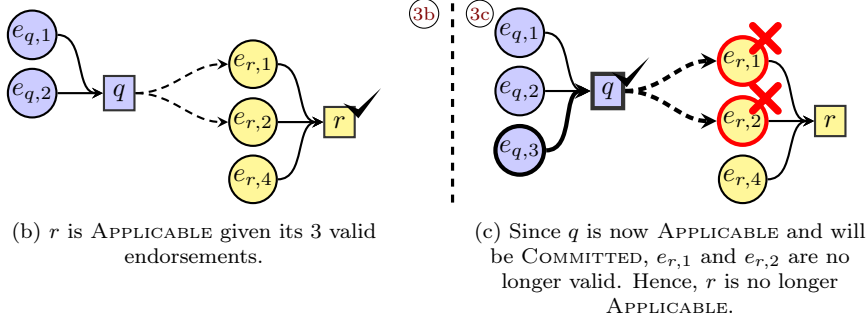
### 2.2.2 Conditional endorsements

Leaderless quorums work well for independent transactions, but might lead to deadlocks in case of conflicts, for instance when modifying the same key with conflicting 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 generated at transaction creation) 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 set of unresolved transactions. We avoid this outcome by periodically triggering garbage collections (or *checkpoints*) using a binary *Byzantine Veto Procedure* (§ 5.6).

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 broadcasting 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 these endorsements will remain valid - because of possible future conflicts. The APPLIED state is an artifact introduced by the speculative execution of a transaction: in this temporary state, the system cannot yet commit a transac-



(a) Simplified history of messages exchanged between the four nodes in our example. Node  $p_1$  first submits transaction  $q$ , that is endorsed by both  $p_1$  and  $p_2$  through  $e_{q,1}$  and  $e_{q,2}$ . Shortly after,  $p_2$  submits a conflicting transaction  $r$  that is endorsed by 3 nodes: with conditions for  $p_1$  and  $p_2$  (see 3b) and without condition for  $p_4$ . After a period of unavailability,  $p_3$  broadcasts its endorsement of  $q$ , leading to state 3c.



**Figure 3:** Example of graph of conditions for transactions  $q, r$  and their respective endorsements  $e_{q,i}$  and  $e_{r,i}$ .  $e_{r,1}$  and  $e_{r,2}$  are conditioned by  $q$ , while other endorsements are not. We set  $\omega = 3$ . (3b) shows the knowledge of correct nodes before the arrival of  $e_{q,3}$  (3c).

tion, 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

#### 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  $\lfloor \frac{n+f}{2} \rfloor$ .
- An endorsement  $e = \langle id, i, C \rangle$  of a transaction  $t = \langle id, d, R, \Delta \rangle$  with

---

**Algorithm 1** Message callbacks at node  $p_i$ 

---

```
1: upon reception of TRANSACTION( $id, d, R, \Delta$ )
2:    $t \leftarrow$  TRANSACTION( $id, d, R, \Delta$ )
3:    $done \leftarrow \perp$ 
4:   while  $done = \perp \wedge \text{CANENDORSE}(t)$  do
5:      $C \leftarrow \{c : c \in T_i \wedge \Gamma(c, \Delta, \Delta)\}$  ▷ List all conflicting transactions
6:     if  $\forall c \in C, c.d \leq \text{now}()$  then
7:       ENDORSE( $t, C$ ) ▷ All conflicting transactions are expired
8:        $done \leftarrow \top$ 
9: upon reception of ENDORSEMENT( $id, j, C$ )
10:  $E_{i,id} \leftarrow E_{i,id} \cup \{\text{ENDORSEMENT}(id, j, C)\}$ 
11:  $\forall t \in T_i : \text{CHECKSTATE}(t)$ 
```

---

( $e.id = t.id$ ) is VALID at node  $p_i$  if and only if every transaction  $c$  in the condition set  $e.C$  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  $p_2$ . ( $e_{x,i}$  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  $p_{1,2,4}$ ), 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. While  $r$  is no longer APPLICABLE,  $q$  received enough *unconditional* endorsements to be APPLICABLE and will be COMMITTED.

## 3.2 Algorithm

The detail of PnyxDB's workings is presented in Algorithms 1 to 4. Our design is reactive: endorsers 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.)



---

**Algorithm 2** Endorsement checks at node  $p_i$ 

---

```
1: function CANENDORSE( $t$ )
2:   if  $t.d \leq \text{now}()$  then
3:     return abort ▷ Timeout
4:   if  $\text{State}_i$  not compatible with  $t.R$  then
5:     return abort ▷ Consistency
6:    $\text{State}' \leftarrow t.\Delta(\text{State})$ 
7:   if  $\text{State}'$  does not comply to  $\text{Policy}_i$  then
8:     return abort ▷ Policy
9:   return OK
10: function ENDORSE( $t, C$ )
11:   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:    $E_{i,id}^+ \leftarrow \{e : e \in E_{i,id} \wedge \text{VALID}(e) \text{ with distinct } e.i\}$ 
3:   return  $|E_{i,id}^+| \geq \omega$ 
4: function VALID( $e$ )
5:   return  $\forall c \in e.C, \neg \text{APPLICABLE}(c.id)$ 
```

---

A client starts a set of operations by broadcasting a  $\text{TRANSACTION}\langle id, d, R, \Delta \rangle$  to nodes, with a configurable deadline  $d$  and a set of operations  $\Delta$ . On receiving this  $\text{TRANSACTION}$  (line 1.1), each endorser first checks whether the transaction can be endorsed ( $\text{CANENDORSE}()$ ) at line 1.4, described in Algorithm 2). In particular, endorsers must check that the transaction's deadline has not been reached with respect to their local clock (line 2.2). Endorsers can also deliberately choose **not** to endorse a transaction simply by ignoring it, for local policy reasons (line 2.8). If  $\text{CANENDORSE}()$  returns *true*, each endorser  $p_i$  then checks that it has not already endorsed conflicting transactions  $C$  (line 1.5). The (symmetric) predicate  $\Gamma$  returns whether the two transactions passed as arguments are in conflict or not. Three cases may happen:

- With no conflicting transaction,  $C = \emptyset$  and  $p_e$  can broadcast an unconditional  $\text{ENDORSEMENT}\langle id, i, \emptyset \rangle$  (line 1.7).
- If  $C$  only contains outdated transactions,  $p_e$  can broadcast a conditional  $\text{ENDORSEMENT}\langle id, i, C \rangle$ , allowing the application of the transaction given the non applicability of every outdated transactions (line 1.7).
- Otherwise,  $p_e$  must wait until conflicting deadlines are over, and restarts the while loop (line 1.4).

New endorsements are received at line 1.9, and trigger the execution of the  $\text{CHECKSTATE}()$  function (described in Algorithm 4) on all transactions already endorsed by the receiving endorser ( $T_i$  set).  $\text{CHECKSTATE}()$  ensures that the state of the datastore  $\text{State}_i$  is consistent with the  $\text{APPLICABLE}$  state of transactions (lines 4.4 and 4.7). It also triggers the  $\text{COMMIT}$  operation on transac-

tions  $q$  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).

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  $t.\Delta$  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  $\omega > \lfloor \frac{n+f}{2} \rfloor$  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 by depending on Byzantine nodes to endorse transactions. (The minimum number of nodes to allow both availability and safety is  $n \geq 3f + 1$  [1].)

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. Note that these states are *local*: each node may have a different view of APPLICABLE transactions depending on the messages it has received. However, our safety property guarantees that no transaction can both be COMMITTED at a correct node  $p_i$  and DROPPED at another correct node  $p_j$ . Conversely, if a transaction reaches one of these two final states at a correct node  $p_i$ , every other correct node will eventually set the same state for that transaction.

### 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 [31], 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.

---

**Algorithm 4** State checking at node  $p_i$ 

---

```
1: function CHECKSTATE( $t$ )
2:   if  $\neg$ APPLICABLE( $t$ ) then
3:     if APPLIED( $t$ ) then
4:        $State_i \leftarrow$  ROLLBACK( $State_i, t$ )
5:     else
6:       if  $\neg$ APPLIED( $t$ ) and  $isSpeculative_i$  then
7:          $State_i \leftarrow$  APPLY( $State_i, t$ ) ▷ Speculative execution
8:         ▷ Endorsements that will always stay valid
9:          $\Sigma_{i,t} = \{e \in E_{i,t} : e.C = \emptyset\}$ 
10:        if  $|\Sigma_{i,t}| \geq \omega$  then
11:          if  $\neg$ APPLIED( $t$ ) then  $State_i \leftarrow$  APPLY( $State_i, t$ )
12:           $State_i \leftarrow$  COMMIT( $State_i, t$ )
13:           $T_i \leftarrow T_i \setminus \{t\}$ 
14:          ▷ Conditions that could be dropped
15:           $\bar{T} \leftarrow \{c \in \cup_{e \in E_{i,t}} e.C : OLD(c)\}$ 
16:          if  $|\bar{T}| \geq 1$  then
17:            STARTCHECKPOINT( $\bar{T}$ )
18: ▷ Example of checkpoint trigger for configurable delay  $\delta$ 
19: function OLD( $t$ )
20:   return  $\neg$ APPLICABLE( $t$ )  $\wedge t.d < (now()) - \delta$ 
```

---

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

---

**Algorithm 5** Checkpoint at node  $p_i$ 

---

```
1: function STARTCHECKPOINT( $\bar{T}$ )
2:   BROADCAST(CHECKPOINT( $\bar{T}$ ))
3: upon reception of CHECKPOINT( $\bar{T}$ )
4:    $c \leftarrow \begin{cases} 0 & \text{if } \exists t \in \bar{T} : \text{APPLICABLE}(t) \vee \text{COMMITTED}(t) \\ 1 & \text{otherwise} \end{cases}$ 
5:    $decision \leftarrow$  BVP( $\bar{T}, c$ )
6:   if  $decision = 1$  then ▷ Cleanup
7:      $T_i \leftarrow T_i \setminus \bar{T}$  ▷ Drop transactions
8:      $\forall t \in \bar{T} : E_{i,t} = \emptyset$  ▷ Forget endorsements of dropped transactions
9:      $\forall t \in T_i, e \in E_{i,t} : e.C = e.C \setminus \bar{T}$  ▷ Forget conditions
10:     $\forall t \in T_i \cup \bar{T} : \text{CHECKSTATE}(t)$ 
```

---

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 § 5.6.

## 4 Implementation

We describe some key elements of our prototype: the web of trust we use for node authentication, and how we implemented the *Reliable Broadcast* and the *Byzantine Veto Procedure* (BVP) we rely on. Our technical choices are driven by our target scale of hundreds to thousands of nodes per network (a reasonable size for closed consortia), excluding clients.

### 4.1 Web of trust and policy files

Our implementation relies on a web of trust and policy files, inspired from PGP, to authenticate nodes. The web of trust links nodes’ identities to their public key, and supports several cryptographic authentication schemes, with ed25519 used by default [32]. 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* which we assume is initially known to all participants: this is similar to the distribution of a common *genesis* file required by a number of existing BFT systems [3, 14, 22].

### 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 [23], but it has a message complexity of  $O(n^2)$ , which makes it impractical for our targeted scale. Following current public and consortium blockchains implementations [33], we use a probabilistic gossip algorithm as our reliable broadcast primitive, where each node communicates only with a small number of neighbors. Such algorithms are known to disseminate information with a logarithmic number of messages. We use GossipSub from the libp2p project as our gossip broadcast algorithm.

Using a gossip algorithm as our broadcast primitive inherently introduces uncertainty in the reliability of the broadcast [34]. 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 Byzantine Veto Procedure

The main limitation with our endorsement scheme is that Byzantine nodes can arbitrarily delay their endorsements. To cope with that 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 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  $p_j$ . It is possible to obtain these two values from active or passive round-trip measurements in the gossip network. With that additional knowledge, each correct node can estimate locally the *earliest possible sending time of a message* and discard messages published after a specific deadline (line 6.6). In our implementation, nodes broadcast periodic heartbeats and we set conservative floor values for  $\hat{\tau}$  and  $\delta_{i,j}$  as an additional safety. Long failures (as described in § 4.2) are indistinguishable from network asynchrony, leading to missing heartbeats and  $\hat{\tau} = \max(\delta_{i,j}) = \infty$  and blocking BVP’s progress.

This is not a concern for PnyxDB: non-conflicting transactions are still allowed to commit if enough nodes are available and BVP will eventually return after failure resolution. This simple approach is sound during periods of synchrony but may introduce significant delays due to the use of a conservative deadline. As BVP is not the main contribution of this paper, we leave the optimization of this primitive to future work.

**Proposition 1** *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  $t \in \bar{T}$  sent after this shared deadline can be accepted. Thanks to assumption A, endorsements sent before ‘deadline’ are delivered before  $(\text{deadline} + \hat{\tau}) > (\text{deadline} + \tau)$ , leading to the same set of endorsements for  $\bar{T}$  being received for every correct node after  $(\text{deadline} + \hat{\tau})$ . This implies the **Agreement** property given the decisions

---

**Algorithm 6** Byzantine Veto Procedure (BVP) at node  $p_i$ 

---

```
1: function BVP( $\bar{T}, c_i$ )
2:   if  $c_i = 0$  then ▷  $p_i$  is vetoing the decision to drop  $\bar{T}$ 
3:     return 0
4:   wait until  $\hat{\tau} < \infty \wedge \forall j, \delta_{i,j} < \infty$  ▷ Wait for synchrony
5:    $\text{deadline} = \max(t.d, t \in \bar{T}) + \max(\delta_{i,j})$ 
6:   Stop delivering endorsements for  $t \in \bar{T}$  sent after deadline
7:   wait until either
8:      $\exists t \in \bar{T} : \text{APPLICABLE}(t)$  then return 0
9:      $\text{now}() > \text{deadline} + \hat{\tau}$  then return 1
```

---

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.

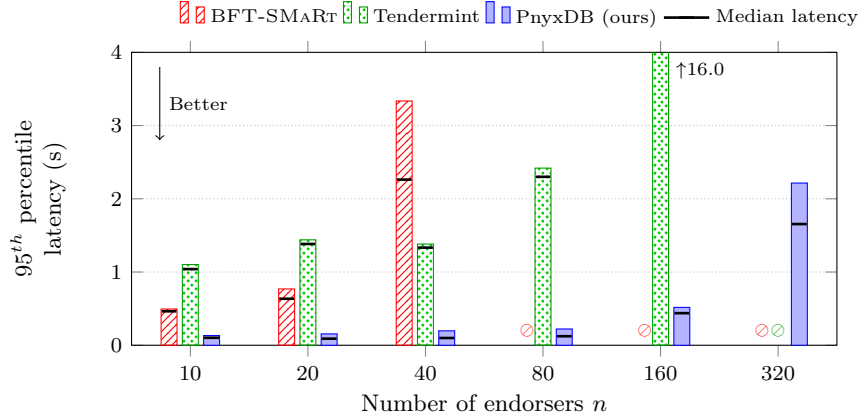
## 5 Evaluation

We tested our implementation of PnyxDB in two settings: (i) in an emulation setup that allows to reproducibly vary and measure the impact of different parameters and (ii) in a large-scale worldwide deployment for practical validation. Except in § 5.5, we did not use speculative execution. We first present the results of the emulation-based evaluation and then detail our large-scale deployment (§ 5.7).

### 5.1 Emulation setup and baselines

The emulation was performed on a server able to sustain several hundreds of nodes with the Mininet network emulation tool (48 threads of Intel(R) Xeon(R) Gold 6136 CPU 3.00GHz, 188GB RAM). We drew latency values for Linux Traffic Control 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 small asynchrony between participants. For the BVP algorithm, we chose the conservative value  $\hat{\tau} = 10$  seconds: this leads to a practical checkpoint timeout of 20 second. Each experiment was run 40 times and averaged.

To compare our work with available solutions, we executed the same experiments with the BFT-SMART v1.2 server [3] and a Tendermint v0.32.5 voting application [22]. BFT-SMART is an efficient Java library for BFT consensus. 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 allowed 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



**Figure 4:** Single transaction commit latency with increasing number of endorsers ( $n$ ) and emulated WAN latencies. The  $\circ$  symbols mean that we were unable to perform the experiment for a specific  $n$  due to network contention. PnyxDB clearly offers best network scalability.

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.)

## 5.2 Network size ( $n$ )

This first experiment measures the latency from a single transaction submission by one client to its commit by every node. We set the required number of endorsers to  $\omega = \lfloor \frac{2}{3}n \rfloor + 1$ . 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  $\circ$  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.

## 5.3 Number of clients

To measure the effect of client workload, we configured a varying 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) [35], a well-known non-relational datastore testing tool. We customized the benchmark workload to create only

**Table 1:** 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 ( $n = 10, \omega = 7$ ).

	Average latency	95 <sup>th</sup> perc. latency	Throughput	Drop rate	Disk usage per node	Transfer per node	Bandwidth per node average / max
BFT-SMART	89 s	170 s	5.68 tx/s	<b>0.0%</b>	-	36 MB	0.16 / 0.21 MB/s
Tendermint	1.7 s	3.9 s	17.0 tx/s	9.3%	26 MB	26 MB	0.40 / 1.40 MB/s
PnyxDB	<b>0.15 s</b>	<b>0.16 s</b>	<b>18.6 tx/s</b>	2.3%	<b>1.4 MB</b>	<b>20 MB</b>	0.28 / 1.27 MB/s

*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 § 5.4. Additionally, each tested network required a quorum of  $\omega = 7$  endorsers among  $n = 10$  to tolerate at most  $f = 3$  faulty nodes.

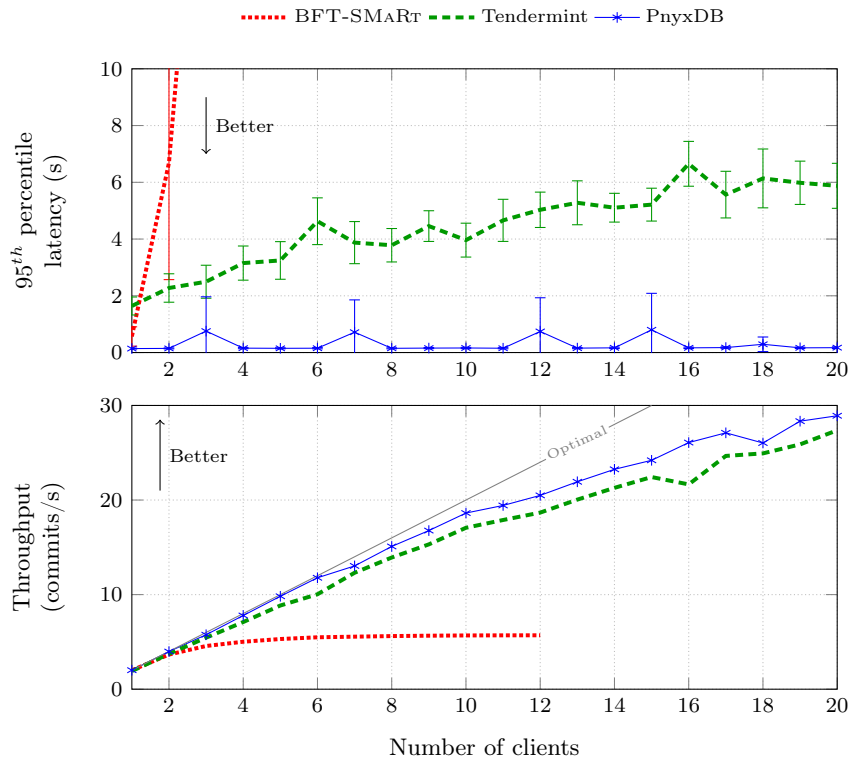
The transaction commit latencies and throughput are shown in Figure 5. While Tendermint and PnyxDB were able to deal with up to 30 transaction commits per second, BFT-SMART was quickly saturated with client transactions: this is due to the large number of messages emitted during the successive rounds of consensus, and our realistic setup with realistic network latencies. PnyxDB performed well for the very large majority of transactions, providing an order of magnitude of latency improvement compared to Tendermint, and approached the optimal throughput while ensuring a low number of dropped transactions. As summarized in Table 1, BFT-SMART ensured that no single transaction was dropped. However, 9.3% of transactions were not committed by all Tendermint nodes: from our analysis, this was caused by several timeouts triggered by these nodes that failed to commit all transactions in time, due to the imposed load and network latencies, leading to deadlocks and consensus halt. PnyxDB experienced less than 2.3% of transaction drop.

In this evaluation, we did not use operation batching [24, 36], a well-known method to boost the throughput of system operations. We focused instead on latency optimization of single-operation transactions, and we see batching as an orthogonal optimization that would further increase the throughput. As an example, a recent BFT-SMART evaluation [37] reports 3,000 op/s for a batch size of 500, which would be equivalent to 6 tx/s in our setup. With PnyxDB, batching would group together multiple operations, but each operation would still be checked independently, and endorsements of conflicting operations left out in the resulting batch of endorsements.

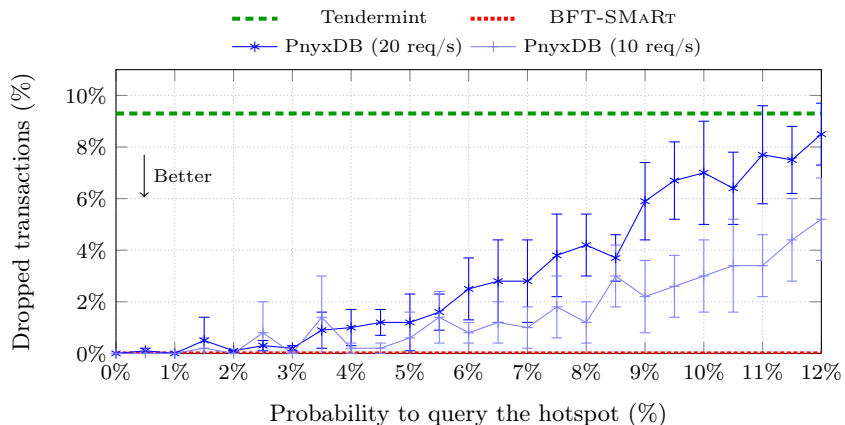
## 5.4 Effect of contention

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 trans-





**Figure 5:** Commit latency and throughput with increasing load and contention. 2 transactions per second per client, from 100 records selected by YCSB. PnyxDB scales with the number of clients while offering best latencies and good throughput ( $n = 10$ ,  $\omega = 7$ ).



**Figure 6:** Drop rate analysis for increasing YCSB hotspot selection ratio. For PnyxDB, the drop rate increases with the hotspot contention.

actions. (The artificially-added clock shift tends to increase the probability of conflicts with high contention levels.) We kept 10 clients for these experiments, leading to an average of 20 transactions per second (tx/s). We also tested a 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.

The contention level impacts transaction drops as follows: 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 Speculative execution

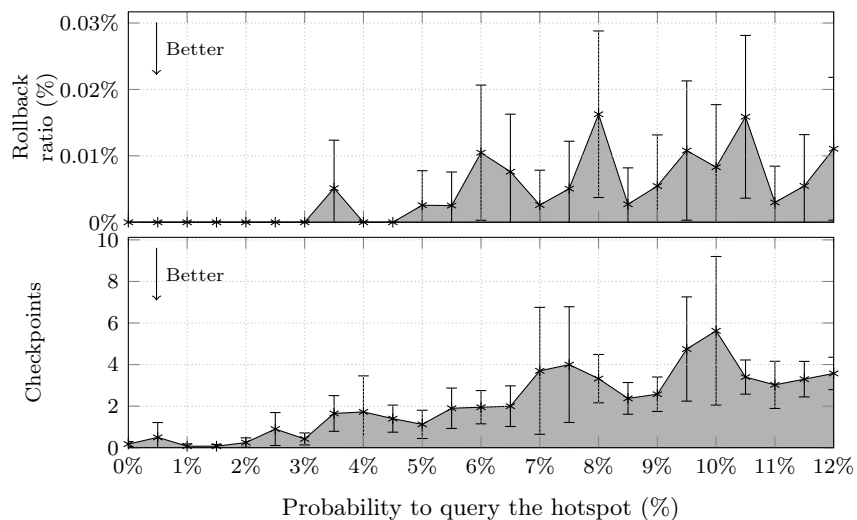
We implemented speculative execution (§ 3) to compare its latencies against classic commit latencies as presented in the previous subsections. First, let us recall that once a transaction is APPLIED by speculative execution, there is no guarantee that it will eventually become COMMITTED. However, we can expect that the probability of rolling back to a non-APPLICABLE (or DROPPED) state stays very low (Figure 7).

In the classic configuration, the commit latency can be severely affected by pending checkpoints, leading to less-predictable performance. We observed that speculative execution provides reduced *and* more predictable latency ( $\mu = 105\text{ms}, \sigma = 145\text{ms}$ ) than commits in the classic configuration ( $\mu = 117\text{ms}, \sigma = 645\text{ms}$ ). This improvement, valid for every evaluated level of contention, came with no additional cost: *at most* 0.03% of speculative executions had to rollback.

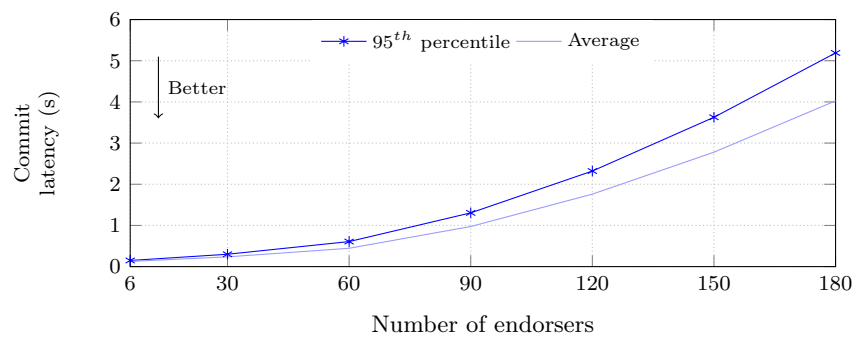
## 5.6 Impact of checkpoints for conflict resolution

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 by elapsed time 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 7). Given that checkpoints happen mainly in times of high levels of contention, we can conclude that their number is still practical, thanks to this 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.



**Figure 7:** Top: experimental ratio between the number of rollbacks and the number of transaction applications. Bottom: number of checkpoints executed across 1000 transaction submissions.



**Figure 8:** Worldwide AWS deployment: commit latency with increasing number of endorser nodes.  $\omega$  was set to 70% of endorsers.

## 5.7 Large scale experiment

To assess PnyxDB’s scalability in a geo-distributed setting, we deployed up to 180 nodes uniformly in 6 AWS regions (Table 2) using t2.micro EC2 instances. We used the same asynchrony parameters and clock deviations (§ 5.1). During our experiments, we observed a steady maximum inter-region round-trip time of 315 ms between the São Paulo and the Sydney regions. Figure 8 shows PnyxDB’s commit latencies for 5 independent transactions submitted by one client, and exhibits an average latency under 5 seconds.

## 6 Related work

Many algorithms have emerged for blockchain-like consensus [38, 33]. For public systems, proof-of-stake has in particular attracted much attention: a small subset of participants are pseudo-randomly selected to lead the consensus for a specific round, based on their account balance (i.e. *stake* in the network) [13]. 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 [39], less communication rounds [28], sharding [40, 41] and pipelining [42].

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 [43] relies on a trusted central bank and on distributed sets of authorities for improved scalability. Similarly, Corda [44] puts trust in small notary clusters running consensus algorithms such as BFT-SMART [3]. In its default configuration, Hyperledger Fabric [14] also requires that a centralized ordering service is trusted by every party. These solutions, while delivering good scalability promises, rely on central points of trust, that if manipulated by a malicious actor would violate the safety of the network. The common assumption is that such entities would be legally accountable through audits: to remove that assumption, Sousa et. al. [24] 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.

Other consortium systems have also been proposed that exploit eventual synchrony [12, 22], and randomization techniques [5, 45, 46]. Such systems usually

**Table 2:** Worldwide AWS deployment: inter-region round-trip time (May 6, 2019). Nodes were evenly shared between regions.

(in ms)	Virginia	São Paulo	Paris	Frankfurt	Sydney
California	61	194	138	147	149
Virginia		147	79	88	204
São Paulo			223	226	315
Paris				10	283
Frankfurt					292

require that correct nodes present deterministic execution for consistency [47, 48]. PnyxDB relies on a consensus algorithm specifically designed for non-deterministic democratic decisions, and exploits operations commutativity in a similar way to [29, 44, 49, 36, 50, 51]. In particular, BFT Generic Broadcast [29] provides optimal latency for non-conflicting messages with  $n > 5f$  nodes, but requires parallel atomic broadcasts in case of conflicting messages and does not allow local transaction policies. The authors of BEAT [5] report a latency of around 500 ms for 6 nodes and more than 1 minute for 92 nodes, while PnyxDB yields 130 ms and less than 1 second in similar AWS deployments, respectively (Figure 8).

Speculative execution has been proposed in BFT consensus to reduce latencies [4, 9, 36, 52]. 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 [53, 54], but they apply only for non-Byzantine fault models.

## 7 Discussion

This section discusses some elements that can affect the liveness of PnyxDB.

**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, 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 [55].

**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 transaction’s 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 common in classic and BFT replicated databases [9], 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 synchrony before allowing to drop 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.

## 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 scales up to hundreds of replicas on a geodistributed cloud deployment.

## Acknowledgements

This work was partially supported by the CominLabs excellence laboratory (ANR-10-LABX-07-01).

## References

- [1] Dahlia Malkhi et al. “Probabilistic Byzantine quorum systems”. In: *PODC*. 1998. DOI: [10.1145/277697.277781](https://doi.org/10.1145/277697.277781).
- [2] Miguel Castro and Barbara Liskov. “Practical Byzantine fault tolerance”. In: *OSDI*. 1999. DOI: [10.1.1.17.7523](https://doi.org/10.1.1.17.7523).
- [3] Alysson Bessani, João Sousa, and Eduardo E P Alchieri. “State Machine Replication for the Masses with BFT-SMART”. In: *DSN*. 2014. DOI: [10.1109/DSN.2014.43](https://doi.org/10.1109/DSN.2014.43).
- [4] Ramakrishna Kotla et al. “Zyzyva: Speculative Byzantine fault tolerance”. In: *SOSP*. Vol. 41. 6. 2007. DOI: [10.1145/1658357.1658358](https://doi.org/10.1145/1658357.1658358).
- [5] Sisi Duan, Michael K Reiter, and Haibin Zhang. “BEAT : Asynchronous BFT Made Practical”. In: *CCS*. 2018.
- [6] Rachid Guerraoui et al. “The Next 700 BFT Protocols”. In: *ECCS*. 2010. DOI: [10.1145/1755913.1755950](https://doi.org/10.1145/1755913.1755950).
- [7] Wagner Saback Dantas et al. “Evaluating Byzantine quorum systems”. In: *SRDS*. 2007. DOI: [10.1109/SRDS.2007.4365701](https://doi.org/10.1109/SRDS.2007.4365701).
- [8] Michael Abd-El-Malek et al. “Fault-scalable Byzantine fault-tolerant services”. In: *SOSP*. 2005. DOI: [10.1145/1095810.1095817](https://doi.org/10.1145/1095810.1095817).
- [9] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. “Byzantine fault-tolerant transaction processing for replicated databases”. In: *NCA*. 2011. DOI: [10.1109/NCA.2011.19](https://doi.org/10.1109/NCA.2011.19).
- [10] Leslie Lamport and David Peleg. “Brief announcement: Leaderless Byzantine Paxos”. In: *DISC*. 2011.

- [11] Iulian Moraru, David G. Andersen, and Michael Kaminsky. “There is more consensus in Egalitarian parliaments”. In: *SOSP*. 2013. DOI: [10.1145/2517349.2517350](https://doi.org/10.1145/2517349.2517350).
- [12] Tyler Crain et al. “DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains”. In: *NCA*. 2018.
- [13] Yossi Gilad et al. “Algorand: Scaling Byzantine Agreements for Cryptocurrencies”. In: *SOSP*. 2017. DOI: [10.1145/3132747.3132757](https://doi.org/10.1145/3132747.3132757).
- [14] Elli Androulaki et al. “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains”. In: *EuroSys*. 2018. DOI: [10.1145/3190508.3190538](https://doi.org/10.1145/3190508.3190538).
- [15] L.M. Goodman. “Tezos — a self-amending crypto-ledger White paper”. In: *CoRR* (2014). URL: [https://www.tezos.com/static/papers/white\\_paper.pdf](https://www.tezos.com/static/papers/white_paper.pdf).
- [16] Beth Simone Noveck. *Wiki government: how technology can make government better, democracy stronger, and citizens more powerful*. 2009.
- [17] Wei Cai et al. “Decentralized Applications: The Blockchain-Empowered Software System”. In: *IEEE Access* 6 (2018). DOI: [10.1109/ACCESS.2018.2870644](https://doi.org/10.1109/ACCESS.2018.2870644).
- [18] Dahlia Malkhi, Kartik Nayak, and Ling Ren. “Flexible Byzantine Fault Tolerance”. In: *CoRR* (2019). URL: <https://arxiv.org/abs/1904.10067>.
- [19] Werner Vogels. “Eventually consistent”. In: *Queue* 6.6 (2008).
- [20] Marc Shapiro and Nuno Preguiça. “Conflict-free replicated data types”. In: *SSS*. 2011.
- [21] Haonan Lu et al. “Existential consistency: Measuring and understanding consistency at Facebook”. In: *SOSP*. 2015. DOI: [10.1145/2815400.2815426](https://doi.org/10.1145/2815400.2815426).
- [22] Ethan Buchman. “Tendermint: Byzantine Fault Tolerance in the Age of Blockchains”. M.Sc. Thesis. University of Guelph, Canada, 2016.
- [23] Gabriel Bracha. “Asynchronous Byzantine agreement protocols”. In: *Inf. Comput.* 75.2 (1987). DOI: [10.1016/0890-5401\(87\)90054-X](https://doi.org/10.1016/0890-5401(87)90054-X).
- [24] João Sousa, Alysson Bessani, and Marko Vukolić. “A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform”. In: *DSN*. 2018. DOI: [10.1145/3152824.3152830](https://doi.org/10.1145/3152824.3152830).
- [25] Partha Dutta, Rachid Guerraoui, and M Vukolic. *Best-case complexity of asynchronous Byzantine consensus*. Tech. rep. 2005.
- [26] Jean Philippe Martin and Lorenzo Alvisi. “Fast Byzantine consensus”. In: *TDSC* 3.3 (2006). DOI: [10.1109/TDSC.2006.35](https://doi.org/10.1109/TDSC.2006.35).
- [27] Hyperledger Fabric. *Docs: Ordering service implementations*. 2020. URL: [https://hyperledger-fabric.readthedocs.io/en/release-2.1/orderer/ordering\\_service.html](https://hyperledger-fabric.readthedocs.io/en/release-2.1/orderer/ordering_service.html) (visited on 05/05/2020).

- [28] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. “Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil”. In: *CoRR* (2018). URL: <https://arxiv.org/abs/1803.05069>.
- [29] Pavel Raykov, Nicolas Schiper, and Fernando Pedone. “Byzantine fault-tolerance with commutative commands”. In: *OPODIS*. 2011. DOI: [10.1007/978-3-642-25873-2\\_23](https://doi.org/10.1007/978-3-642-25873-2_23).
- [30] Cheng Li et al. “Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary”. In: *OSDI*. 2012.
- [31] Özalp Babaoglu and Sam Toueg. *Non-blocking atomic commitment*. Tech. rep. 1993.
- [32] Daniel J Bernstein et al. “Ed25519: high-speed high-security signatures.” In: *JCE* 2.2 (2012).
- [33] Shehar Bano et al. “Consensus in the Age of Blockchains”. In: *CoRR* (2017). URL: <https://arxiv.org/abs/1711.03936>.
- [34] Rachid Guerraoui et al. “Scalable Byzantine Reliable Broadcast”. In: *DISC*. 2019. DOI: [10.4230/LIPIcs.DISC.2019.22](https://doi.org/10.4230/LIPIcs.DISC.2019.22).
- [35] Brian F. Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *SoCC*. 2010. DOI: [10.1145/1807128.1807152](https://doi.org/10.1145/1807128.1807152).
- [36] Atul Singh, Pedro Fonseca, and Petr Kuznetsov. “Zeno: Eventually Consistent Byzantine-Fault Tolerance”. In: *NSDI*. 2009.
- [37] Christian Berger et al. “Resilient wide-area byzantine consensus using adaptive weighted replication”. In: *SRDS*. 2019. DOI: [10.1109/SRDS47363.2019.00029](https://doi.org/10.1109/SRDS47363.2019.00029).
- [38] Juan Garay and Aggelos Kiayias. “SoK : A Consensus Taxonomy in the Blockchain Era”. In: *Cryptology* (2018).
- [39] Serguei Popov. “The Tangle”. In: *CoRR* (2018). URL: <https://www.iota.org/research/academic-papers>.
- [40] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. “Mir-BFT: High-Throughput BFT for Blockchains”. In: *CoRR* (2019). URL: <https://arxiv.org/abs/1906.05552>.
- [41] Jiaping Wang and Hao Wang. “Monoxide: Scale out blockchain with asynchronous consensus zones”. In: *NSDI*. 2019.
- [42] Gauthier Voron and Vincent Gramoli. “Dispel: Byzantine SMR with Distributed Pipelining”. In: (2019).
- [43] George Danezis and Sarah Meiklejohn. “Centrally Banked Cryptocurrencies”. In: *NDSS*. 2016. DOI: [10.14722/ndss.2016.23187](https://doi.org/10.14722/ndss.2016.23187).
- [44] Mike Hearn. “Corda: A distributed ledger”. In: *CoRR* (2016). URL: [https://docs.corda.net/releases/release-V3.1/\\_static/corda-technical-whitepaper.pdf](https://docs.corda.net/releases/release-V3.1/_static/corda-technical-whitepaper.pdf).



- [45] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. “Turquoise: Byzantine consensus in wireless ad hoc networks”. In: *DSN*. 2010. DOI: [10.1109/DSN.2010.5544268](https://doi.org/10.1109/DSN.2010.5544268).
- [46] Andrew Miller et al. “The Honey Badger of BFT Protocols”. In: *CCS*. 2016. DOI: [10.1145/2976749.2978399](https://doi.org/10.1145/2976749.2978399).
- [47] Christian Cachin, Simon Schubert, and Marko Vukolić. “Non-determinism in Byzantine Fault-Tolerant Replication”. In: *CoRR* (2016). URL: <https://arxiv.org/abs/1603.07351>.
- [48] Kazuhiro Yamashita et al. “Potential Risks of Hyperledger Fabric Smart Contracts”. In: *IWBOSE*. 2019. DOI: [10.1109/IWBOSE.2019.8666486](https://doi.org/10.1109/IWBOSE.2019.8666486).
- [49] Prince Mahajan et al. “Depot : Cloud storage with minimal trust”. In: *OSDI* (2010). DOI: [10.1145/2063509.2063512](https://doi.org/10.1145/2063509.2063512).
- [50] Seo Jin Park and John Ousterhout. “Exploiting Commutativity For Practical Fast Replication”. In: *NSDI*. 2019.
- [51] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. “Gryff : Unifying Consensus and Shared Registers”. In: *NSDI*. 2020.
- [52] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. “Efficient middleware for byzantine fault tolerant database replication”. In: *EuroSys*. 2011. DOI: [10.1145/1966445.1966456](https://doi.org/10.1145/1966445.1966456).
- [53] Peter J Keleher. “Decentralized Replicated-Object Protocols”. In: *PODC*. 1999.
- [54] Joao Barreto and Paulo Ferreira. “Version Vector Weighted Voting protocol: efficient and fault-tolerant commitment for weakly connected replicas”. In: *Concurrency and Computation: Prac. and Exp.* 19.17 (2007). DOI: [10.1002/cpe](https://doi.org/10.1002/cpe).
- [55] Allen Clement et al. “Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults”. In: *NSDI*. 2009. DOI: [10.1145/1989727.1989732](https://doi.org/10.1145/1989727.1989732).

## A Proof sketches

### A.1 Liveness

**Lemma 1 (Acyclic conditions)** *Let  $q, r$  be two transactions with  $q.d \leq r.d$ . There cannot be any  $\text{ENDORSEMENT}\langle q.id, i, C \rangle$  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 1.7. For this line to be executed,  $\text{CANENDORSE}(q)$  must have returned **OK**. Hence, per line 2.2, we must have  $q.d > \text{now}()$ . Given line 1.6, every element  $r$  of  $C$  must fulfill  $r.d \leq \text{now}()$ . If we assume that local operations execute instantaneously, the value of  $p_i$ 's local clock  $\text{now}$  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 2 (Termination)** *Assuming the BVP protocol terminates, every proposed functions and callbacks terminate.*

**Proof sketch.** This is trivial for functions  $\text{CANENDORSE}$  and  $\text{ENDORSE}$  in Algorithm 2. When receiving a  $\text{TRANSACTION}$  message at line 1.1, a node may execute the while loop (lines 1.4-1.8) several times if conflicts are detected. A node is, however, guaranteed to exit the while loop when  $\text{CANENDORSE}$  returns false because the transaction's deadline is over (timeout clause). Upon reception of the messages  $\text{ENDORSEMENT}$  (line 1.9), and  $\text{CHECKPOINT}$  (line 4.3), and for the function  $\text{CHECKSTATE}$  (line 4.1), the termination is conditioned by the termination of the Byzantine Veto Procedure (BVP) and the  $\text{APPLICABLE}$  predicate.

BVP terminates by assumption. The case of  $\text{APPLICABLE}$  is somewhat more involved, as the functions  $\text{APPLICABLE}$  and  $\text{VALID}$  recursively call each other. Every  $\text{APPLICABLE}$  call of transaction  $r$  will trigger a finite number of  $\text{VALID}$  calls on endorsements  $e \in E_{i,r.id}$  received for  $r$  (line 3.2). Each  $\text{VALID}$  call will in turn call a finite number of  $\text{APPLICABLE}$  call for every  $c \in e.C$  (line 3.5). Because of Lemma 1, we know that for two transactions  $q, r$  with  $q.d \leq r.d$ , there can be no  $\text{ENDORSEMENT}\langle q.id, i, C \rangle$  with  $r \in C$  in any  $E_{j,q.id}$  set. This property implies that the recursion between the two predicates will call  $\text{APPLICABLE}$  with transactions ordered by decreasing deadline  $q.d$ , thus eliminating any loop, and ensuring that the recursion terminates.

### A.2 Safety

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

$$\Gamma(q.\Delta, r.\Delta) \Rightarrow \neg(\text{APPLICABLE}(q) \wedge \text{APPLICABLE}(r)),$$

where  $q.\Delta$ , and  $r.\Delta$  represent the operations of  $q$  and  $r$ , respectively, and  $\Gamma(-, -)$  indicates if the two sets of operations are in conflict.

**Proof sketch.** Let us assume that  $\text{APPLICABLE}(q)$  and  $\text{APPLICABLE}(r)$  hold at correct node  $p_i$ . By Algorithm 3, this means that at least  $\omega$  distinct nodes have endorsed  $q$  with endorsements  $\{e : r \notin e.C\}$ . We apply the same reasoning for  $r$ . By using the well-known property of Byzantine Quorums [1], we conclude that at least  $k \geq f + 1$  nodes endorsed both  $q$  and  $r$  without including the other in its conflict set  $C$ .

$$k \geq \omega + \omega - n \geq 2 \left( \left\lfloor \frac{n+f}{2} \right\rfloor + 1 \right) - n \geq f + 1$$

Some correct node(s) violated Algorithm 1 by ignoring  $q$  during the endorsement of transaction  $r$  (or conversely by symmetry of  $\Gamma$ ), leading to a contradiction.

**Proposition 3 (Agreement)** *For any two correct nodes  $\{i, j\}$ , no transaction can be both COMMITTED by  $i$  and DROPPED by  $j$ .*

**Proof sketch.** A transaction  $t$  can only be DROPPED after a successful checkpoint for  $t \in \bar{T}$  (BVP returning a decision of 1, line 5.7). By the validity property of BVP, “if a correct node decides 1, then all correct nodes proposed 1”. Line 5.4 forces correct nodes to propose 0 if  $t \in \bar{T}$  is COMMITTED. Hence, once COMMITTED by  $i$ ,  $t$  cannot be DROPPED anymore by other correct nodes. For the other direction: if one correct node decides to drop  $t$  during checkpointing, it removes  $t$  from its sets of pending transactions  $T_i$  and endorsements  $E_i$ , making COMMIT of  $t$  impossible according to Algorithm 4. By BVP agreement, every correct node will apply this pruning as long as one decided to.

**Proposition 4 (Eventual consistency)** *If a transaction  $t$  is COMMITTED (resp. DROPPED) at a correct node, every correct node will eventually COMMIT (resp. DROP)  $t$ .*

**Proof sketch.** By definition of our reliable broadcast primitive, every correct node will eventually receive all endorsements for  $t$ . If the number of broadcasted unconditional endorsements is sufficient (line 4.9), the proof for COMMITTED is straightforward. If a checkpoint is proposed due to conditional endorsements blocking progress, the checkpoint mechanism guarantees that every correct node will remove the same conditions from received endorsements (line 5.9). Hence, every correct node will eventually hold the same set of unconditional endorsements and COMMIT the same transactions. DROP case is straightforward by BVP agreement, since it can only happen during checkpointing and all correct node decide the same value.

**Proposition 5 (Safety)** *No two conflicting transactions  $q, r$  can be both COMMITTED by any correct node.*

**Proof sketch.** A COMMIT operation only happens when a Byzantine Quorums of unconditional endorsements is available (line 4.9). In Lemma 2, we already showed that two conflicting transactions cannot reach a Byzantine Quorum at the same time by definition of  $\omega$ . However, we need to verify the pruning of conditions (for conditional endorsements) during checkpoints. Conditions pointing to transaction  $q$  are removed if and only if  $q$  is DROPPED (lines 5.7 and 5.9). Even if  $r$  is COMMITTED after conditions pruning, we can be sure that  $q$  cannot be COMMITTED anymore thanks to Proposition 3. Proposition 4 states that every correct node reaches the same decision for  $q$  and  $r$ .

### A.3 Fairness

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 6 (Fairness)** *If no majority of correct nodes  $k > \lfloor \frac{n-f}{2} \rfloor$  endorsed a transaction  $t$ , APPLICABLE( $t$ ) will never hold at any correct node.*

**Proof sketch.** Let  $\epsilon$  be the total number of endorsements for  $t$ . We assume that APPLICABLE( $t$ ) hold, hence we have  $\omega \leq \epsilon \leq \lfloor \frac{n-f}{2} \rfloor + f$  since no majority of correct nodes endorsed  $t$ . Given the definition of  $\omega$ , it yields the following contradiction:

$$\lfloor \frac{n+f}{2} \rfloor < \lfloor \frac{n-f}{2} \rfloor + f = \lfloor \frac{n+f}{2} \rfloor$$

**Table A.I:** Notations used in this paper.

---

<b>System parameters</b>	
$n$	Number of nodes
$f$	Number of faulty nodes
$\omega$	Required quorum of endorsements
$\Gamma(\Delta, \bar{\Delta})$	$= \begin{cases} \mathbf{true} & \text{if } \Delta \text{ and } \bar{\Delta} \text{ are in conflict} \\ \mathbf{false} & \text{otherwise} \end{cases}$ where $\Delta, \bar{\Delta}$ are two lists of operations

---

<b>Message <math>t \leftarrow \mathbf{TRANSACTION}\langle id, d, R, \Delta \rangle</math></b>	
$t.id$	Unique identifier
$t.d$	Absolute deadline
$t.R$	Preconditions on datastore state
$t.\Delta$	List of operations

---

<b>Message <math>e \leftarrow \mathbf{ENDORSEMENT}\langle id, i, C \rangle</math></b>	
$e.id$	Endorsed transaction unique identifier
$e.i$	Endorser node identifier
$e.C$	Endorsement conditions, the set of transactions that must <b>not</b> be applied for this endorsement to be valid

---

<b>Variables of node <math>p_i</math></b>	
$isSpeculative_i$	Whether $p_i$ speculatively applies transactions
$State_i$	Datastore state
$T_i$	Transactions endorsed by $p_i$ so far
$E_{i,id}$	Endorsements received by $p_i$ for transaction $id$
$Policy_i$	Set of rules that define if $p_i$ agrees to apply given transactions. This is not necessarily a deterministic function and may involve human interaction

---