PnyxDB: a Lightweight Leaderless Democratic Byzantine Fault Tolerant Replicated Datastore

Loïck Bonniot, Christoph Neumann, François Taïani

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*[1] 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

[1] The Pnyx hill was used as the main meeting place in Athenian democracy.

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 specifies 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.

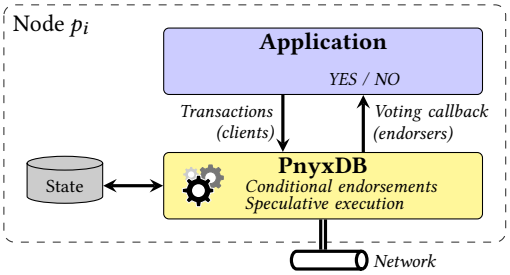


Figure 1. Overview of PnyxDB: the application submits transactions to be executed on shared state, and polls the application back for transaction approval before creating conditional endorsements.

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. Such a behavior is typically the case for malformed, corrupted or malicious nodes. Non-Byzantine nodes are said to be *correct*.

We also 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* [12]. In our implementation, we rely on eventually synchronous networks to ensure that assumption, as detailed in § 4.2. Cryptographic signatures are used to verify nodes' identity and authorizations. We make the standard assumptions that Byzantine participants cannot break these signatures, and that participants know each other beforehand. In the parlance of distributed ledgers, our system is *permissioned*: this allows for message authenticity and data access control while staying relatively dynamic.

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 guaranties 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

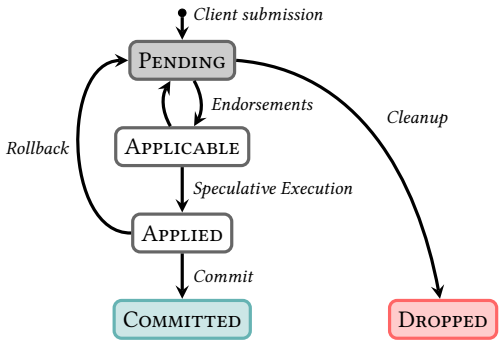


Figure 2. Transaction state diagram, as viewed by a node. From the Pending state, a transaction evolve either to Dropped or Committed given received messages. Dropped and Committed are eventually consistent across all nodes. In contrast, Pending, Applicable and Applied are intermediate states local to each node.

Table 1. Notations used in this paper.

System parameters	
n	Number of nodes
f	Number of faulty nodes
ω	Required quorum of endorsements
$\Gamma(\Delta, \bar{\Delta})$	$= \begin{cases} \textbf{true} \textit{ if } \Delta \textit{ and } \bar{\Delta} \textit{ conflict} \\ \textbf{false} \textit{ otherwise} \end{cases}$ where $\Delta, \bar{\Delta}$ are two lists of operations
Message $t \leftarrow$ **Transaction**$\langle id, d, R, \Delta \rangle$	
$t.id$	Unique identifier
$t.d$	Absolute deadline
$t.R$	Preconditions on datastore state
$t.\Delta$	List of operations
Message $e \leftarrow$ **Endorsement**$\langle id, i, C \rangle$	
$e.id$	Endorsed transaction unique identifier
$e.i$	Endorser node identifier
$e.C$	Endorsement conditions, the set of transactions that must **not** be applied for this endorsement to be valid
Variables of node p_i	
$isSpeculative_i$	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

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 ω VALID endorsements for t at node p_i, where ω 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 $(e.id = t.id)$ 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

Algorithm 1 Message callbacks at node p_i

```
1:  upon reception of TRANSACTION⟨id, d, R, Δ⟩
2:      t ← TRANSACTION⟨id, d, R, Δ⟩
3:      done ← ⊥
4:      ▷ Continue until no active conflicting transaction present
5:      while done = ⊥ do
6:          if CANENDORSE(t) then
7:              C ← {c : c ∈ T_i ∧ Γ(c.Δ, Δ)}
8:              if C = ∅ then
9:                  ▷ No conflicting transaction
10:                 ENDORSE(t, ∅)
11:                 done ← ⊤
12:             else
13:                 if ∀c ∈ C, c.d ≤ now() then
14:                     ▷ Expired conflicting transactions
15:                     ENDORSE(t, C)
16:                     done ← ⊤
17:                 ▷ Otherwise, not done, going back to start of while loop
18:         else
19:             ▷ Unable to endorse
20:             done ← ⊤
21: upon reception of ENDORSEMENT⟨id, j, C⟩
22:     E_{i,id} ← E_{i,id} ∪ {ENDORSEMENT⟨id, j, C⟩}
23:     ∀t ∈ T_i : CHECKSTATE(t)
```

Algorithm 2 Endorsement checks at node p_i

```
1:  function CANENDORSE(t)
2:      if t.d ≤ now() then
3:          return abort                          ▷ Timeout
4:      if State_i not compatible with t.R then
5:          return abort                          ▷ Consistency
6:      State' ← t.Δ(State)
7:      if State' does not comply to 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 ← T_i ∪ t.id
```

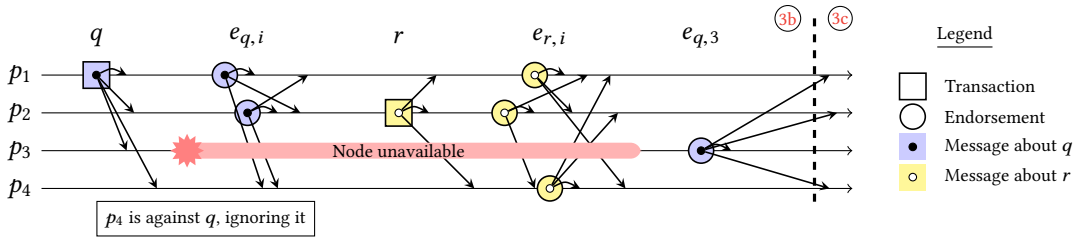
APPLICABLE, and may be speculatively executed but cannot be COMMITTED yet as q has not been DROPPED.

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.

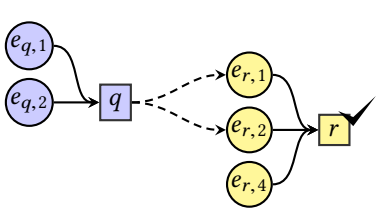
3.2 Algorithm

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.)

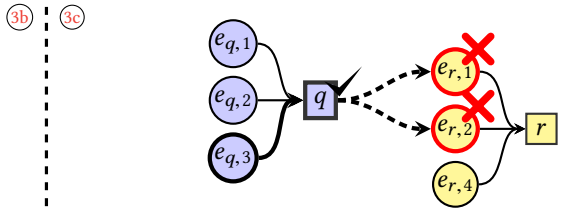
A client starts a set of operations by broadcasting a TRANSACTION$\langle id, d, R, \Delta \rangle$ to nodes, with a configurable deadline d and a set of operations Δ. On receiving this TRANSACTION



(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.



(b) r is Applicable given its 3 valid endorsements.



(c) Since q is now Applicable, $e_{r,1}$ and $e_{r,2}$ are no longer valid. Hence, r is no longer Applicable.

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).

Algorithm 3 Predicates at node p_i

```
1: function Applicable(id)
2:     E+_{i,id} ← {e : e ∈ E_{i,id} ∧ Valid(e)}
3:     return |E+_{i,id}| ≥ ω
4: function Valid(e)
5:     return ∀c ∈ e.C, ¬Applicable(c.id)
```

(line 1.1), each endorser first checks whether the transaction can be endorsed (CanEndorse() at line 1.6, 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 CanEndorse() returns *true*, each endorser p_i then checks that it has not already endorsed conflicting transactions C (lines 1.7-1.8). The predicate Γ returns *true* if the two transactions passed as arguments are in conflict. Three cases may happen:

- If no conflicting transaction exists, p_e can broadcast its Endorsement$\langle id, i, \emptyset\rangle$ *without condition* (line 1.10).
- If C only contains outdated transactions, p_e can broadcast a conditional Endorsement$\langle id, i, C\rangle$, allowing the application of the transaction given the non applicability of every outdated transactions (line 1.15).
- Otherwise, p_e must wait until conflicting deadlines are over, and restarts the while loop (line 1.5).

New endorsements are received at line 1.21, and trigger the execution of the CheckState() function (described in Algorithm 4) on all transactions already endorsed by the 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 ω 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 $\omega > \left\lfloor \frac{n+f}{2} \right\rfloor$ to tackle Byzantine endorsements. Higher ω 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 $n \geq 3f + 1$ [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 ¬Applicable(t) then
3:          if Applied(t) then
4:              State_i ← Rollback(State_i, t)
5:      else
6:          if ¬Applied(t) and isSpeculative_i then
7:              State_i ← Apply(State_i, t)            ▷ Speculative execution
8:          ▷ Endorsements that will always stay valid
9:          Σ_{i,t} = {e ∈ E_{i,t} : e.C = ∅}
10:         if |Σ_{i,t}| ≥ ω then
11:             if ¬Applied(t) then State_i ← Apply(State_i, t)
12:             State_i ← Commit(State_i, t)
13:             T_i ← T_i \ {t}
14:         ▷ Conditions that could be dropped
15:         T̄ ← {∀e ∈ E_{i,t}, c ∈ e.C : Old(c)}
16:         if |T̄| ≥ 1 then
17:             StartCheckpoint(T̄)
18: ▷ Example of checkpoint trigger for configurable delay δ
19: function Old(t)
20:     return ¬Applicable(t) ∧ t.d < (now() − δ)
```

Algorithm 5 Checkpoint at node p_i

```
1:  function StartCheckpoint(T̄)
2:      Broadcast(Checkpoint⟨T̄⟩)
3:  upon reception of Checkpoint⟨T̄⟩
4:      c ← { 0 if ∃t ∈ T̄ : Applicable(t) ∨ Committed(t)
            { 1 otherwise
5:      decision ← BVP(T̄, c)
6:      if decision = 1 then                           ▷ Cleanup
7:          T_i ← T_i \ T̄                              ▷ Drop transactions
8:          ∀t ∈ T̄ : E_{i,t} = ∅                       ▷ Forget endorsements of dropped transactions
9:          ∀t ∈ T_i, e ∈ E_{i,t} : e.C = e.C \ T̄      ▷ Forget conditions
10:         ∀t ∈ T_i ∪ T̄ : CheckState(t)
```

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 those two final states at a correct node p_i, every other correct node will eventually set the same state for that transaction. We revisit these points in § 3.4, where we formally prove some of PnyxDB's key properties.

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.

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 commiting 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.

3.4 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$\langle q.id, i, C\rangle$ 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 1.15. 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 Endorsement (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 $e \in E_{i,r.id}$ received for r (line 3.2). Each Valid call will in turn call a finite number of Applicable call for every $c \in e.C$ (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$\langle q.id, i, C\rangle$ with $r \in C$ in any $E_{j,q.id}$ set. This property implies that the recursion between the two predicates will call 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 $q \neq r$:*

$$\Gamma(q.\Delta, r.\Delta) \Rightarrow \neg(\textit{Applicable}(q) \ \wedge \ \textit{Applicable}(r))$$

Proof sketch. Without loss of generality, we suppose $q.d \leq r.d$ and that Applicable(q) and Applicable(r) hold at correct node p_i. Let us note $C_\emptyset$ a condition set such that $q \notin C_\emptyset$. Given Lemma 3.1 and instruction 3.5, the only endorsements for q and r that are valid for p_i are $E^+_{i,q} = \{\text{Endorsement}\langle q, j, C_\emptyset\rangle\}$ and $E^+_{i,r} = \{\text{Endorsement}\langle r, j, C_\emptyset\rangle\}$, for any endorser p_j. According to line 3.3, we must have $\omega \leq |E^+_{i,q}|$. Since correct nodes cannot send both kind of endorsements per algorithm 1, we also must have $\omega \leq |E^+_{i,r}| \leq n - |E^+_{i,q}| + f \leq n - \omega + f$, or more simply $\omega \leq \frac{n+f}{2}$. This leads to a contradiction with $\left\lfloor \frac{n+f}{2} \right\rfloor + 1 \leq \omega$. □

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 $E_{i,q.id} = E_{j,q.id}$, 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. Given 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 ($decision = 1$), 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 ω 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 $k > \left\lfloor \frac{n-f}{2} \right\rfloor$ endorsed a transaction t, Applicable(t) will never hold at any correct node.*

Proof sketch. Let ϵ be the number of endorsements for t. We must have $\omega \leq \epsilon \leq \left\lfloor \frac{n-f}{2} \right\rfloor + f$ for q to be Applicable, which is impossible due to our definition of ω. □

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 GossipSub [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 τ, 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 ($\mathcal{A}$) $\hat{\tau}$ such as $\hat{\tau} \geq \tau$ and ($\mathcal{B}$) $\delta_{i,j}$ 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 $\mathcal{A}$ and $\mathcal{B}$ 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 $\mathcal{A}$, 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

Algorithm 6 Byzantine Veto Procedure (BVP) at node p_i

```
1: function BVP(T̄, c_i)
2:     if c_i = 0 then                          ▷ p_i is vetoing the decision to drop T̄
3:         return 0
4:     wait until τ̂ < ∞ ∧ ∀j, δ_{i,j} < ∞       ▷ Wait for synchrony
5:     deadline = max(t.d, t ∈ T̄) + max(δ_{i,j})
6:     Stop delivering endorsements for t ∈ T̄ sent after deadline
7:     wait until either
8:         ∃t ∈ T̄ : Applicable(t) then return 0
9:         now() > deadline + τ̂ then return 1
```

(deadline + $\hat{\tau}$). 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 $\hat{\tau} = 10$ 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.)

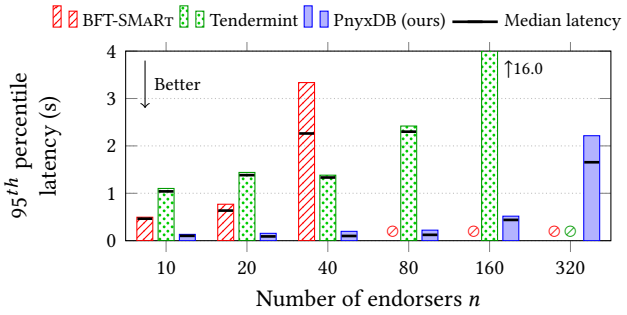


Figure 4. Single transaction commit latency with increasing number of endorsers nodes (n) and emulated WAN latencies. The ⊘ symbols mean that we were unable to perform the experiment for a specific n due to network contention. PnyxDB clearly offers best network scalability.

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 $\omega = \left\lfloor \frac{2}{3}n \right\rfloor + 1$ 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 ⊘ 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

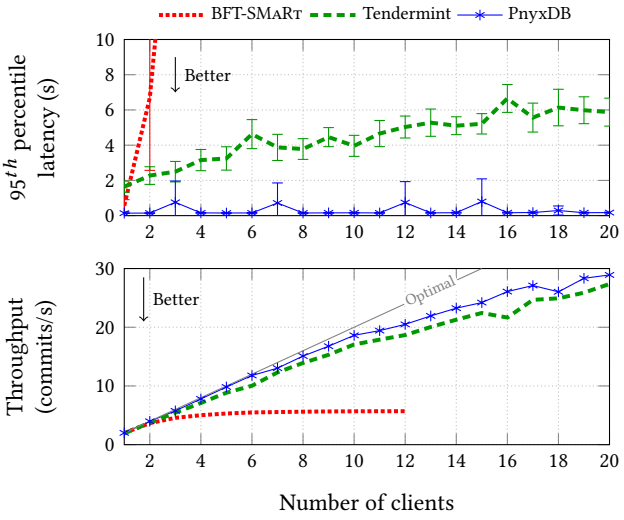


Figure 5. Commit latency and throughput with increasing load and contention, each client submitting 2 transactions per second from 100 records selected by YCSB. PnyxDB can scale with the number of clients while offering best latencies and good throughput ($n = 10$ and $\omega = 7$).

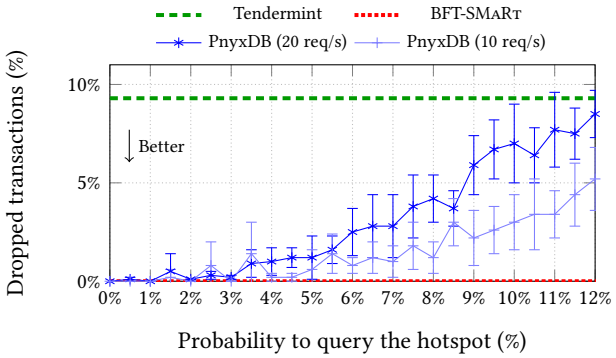


Figure 6. Drop rate analysis for increasing YCSB hotspot selection ratio. As expected for PnyxDB, the drop rate increases with the hotspot contention.

$n = 10$ 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 2, BFT-SMaRt ensured that no single transaction was dropped. However, Tendermint nodes were unable to commit around 9.3% of transactions: from our understanding, some nodes failed to keep their state synchronized with the network and gave up processing transactions. PnyxDB experienced less than 2.3% of transaction drop on average.

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 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 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 latency 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 ($n = 10, \omega = 7$).

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

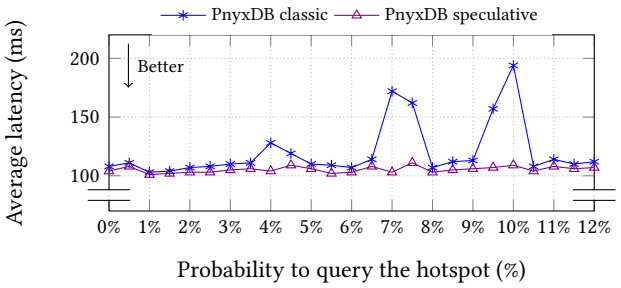


Figure 7. Difference between classic and speculative operation execution latencies. There is a significant gain with speculative execution for high levels of contention.

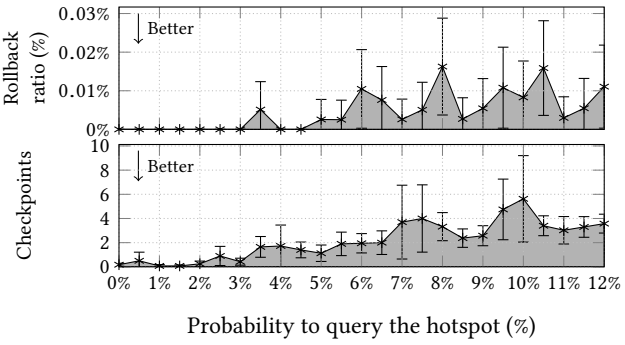


Figure 8. Top: experimental ratio between the number of rollbacks and the number of transaction applications. Bottom: number of checkpoints executed across 1000 transaction submissions.

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.

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

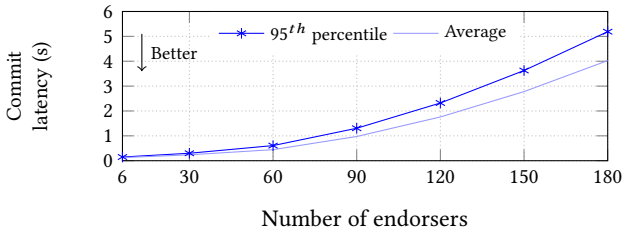


Figure 9. Worldwide AWS deployment: commit latency with increasing number of endorser nodes. ω was set to 70% of endorsers.

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).

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 pseudo-randomly 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 sytem. 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

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. In *SOSP*. 59–74. DOI: http://dx.doi.org/10.1145/1095810.1095817

[2] A. Abdul-Rahman. 1997. The PGP Trust Model. *EDI-Forum: The Journal of Electronic Commerce* 10, 3 (1997), 27–31.

[3] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. 2018. Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil. *CoRR* (2018). https://arxiv.org/abs/1803.05069

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys*. 30–45. DOI: http://dx.doi.org/10.1145/3190508.3190538

[5] Özalp Babaoglu and Sam Toueg. 1993. *Non-blocking atomic commitment*. Technical Report.

[6] Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. 2017. Proofs of Useful Work. *IACR Cryptology ePrint Archive* (2017), 203.

[7] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2017. Consensus in the Age of Blockchains. *CoRR* (nov 2017). https://arxiv.org/abs/1711.03936

[8] Joao Barreto and Paulo Ferreira. 2007. Version Vector Weighted Voting protocol: efficient and fault-tolerant commitment for weakly connected replicas. *Concurrency and Computation: Practice and Experience* 19, 17 (2007), 2271–2283. DOI: http://dx.doi.org/10.1002/cpe

[9] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-yin Yang. 2012. Ed25519: high-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (2012), 77–89.

[10] Alysson Bessani, João Sousa, and Eduardo E P Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *DSN*. 355–362. DOI: http://dx.doi.org/10.1109/DSN.2014.43

[11] Alysson Neves Bessani, Eduardo Pielison Alchieri, Miguel Correia, and Joni da Silva Fraga. 2008. DepSpace: a Byzantine fault-tolerant coordination service. In *EuroSys*. 163–176. DOI: http://dx.doi.org/10.1145/1352592.1352610

[12] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Inf. Comput.* 75, 2 (1987), 130–143. DOI: http://dx.doi.org/10.1016/0890-5401(87)90054-X

[13] Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. M.Sc. Thesis. University of Guelph, Canada.

[14] Christian Cachin, Simon Schubert, and Marko Vukolić. 2016. Non-determinism in Byzantine Fault-Tolerant Replication. *CoRR* (2016). https://arxiv.org/abs/1603.07351

[15] Wei Cai, Zehua Wang, Jason B. Ernst, Zhen Hong, Chen Feng, and Victor C.M. Leung. 2018. Decentralized Applications: The Blockchain-Empowered Software System. *IEEE Access* 6 (2018), 53019–53033. DOI: http://dx.doi.org/10.1109/ACCESS.2018.2870644

[16] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *OSDI*. 173–186. DOI: http://dx.doi.org/10.1.1.17.7523

[17] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*. 153–168. DOI: http://dx.doi.org/10.1145/1989727.1989732

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154. DOI: http://dx.doi.org/10.1145/1807128.1807152

[19] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 2018. DBFT: Efficient Byzantine Consensus with a Weak Coordinator and its Application to Consortium Blockchains. *CoRR* (2018). https://arxiv.org/abs/1702.03068

[20] George Danezis and Sarah Meiklejohn. 2016. Centrally Banked Cryptocurrencies. In *NDSS*. DOI: http://dx.doi.org/10.14722/ndss.2016.23187

[21] Wagner Saback Dantas, Alysson Neves Bessani, Joni Da Silva Fraga, and Miguel Correia. 2007. Evaluating Byzantine quorum systems. In *SRDS*. 253–262. DOI: http://dx.doi.org/10.1109/SRDS.2007.4365701

[22] John R. Douceur. 2002. The Sybil Attack. In *International workshop on peer-to-peer systems*. 251–260. DOI: http://dx.doi.org/10.1080/1461670X.2016.1175315

[23] Sisi Duan, Sean Peisert, and Karl Levitt. 2015. hBFT : Speculative Byzantine Fault Tolerance With Minimum Cost. *IEEE Transactions on Dependable and Secure Computing* 12, 1 (2015), 58–70.

[24] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT : Asynchronous BFT Made Practical. In *CCS*.

[25] Partha Dutta, Rachid Guerraoui, and M Vukolic. 2005. *Best-case complexity of asynchronous Byzantine consensus*. Technical Report.

[26] Ethereum. 2019. Create a Democracy contract in Ethereum. (2019). https://www.ethereum.org/dao

[27] Hyperledger Fabric. 2019. Docs: Ordering service implementations. (2019). https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html

[28] Davide Frey, Marc X Makkes, Pierre-Louis Roman, François Taïani, and Spyros Voulgaris. 2016. Bringing secure Bitcoin transactions to your smartphone. In *Workshop on Adaptive & Reflective Middleware (ARM)*.

[29] Roy Friedman and Roni Licher. 2017. Hardening Cassandra against Byzantine failures. In *OPODIS*. DOI: http://dx.doi.org/10.4230/LIPIcs.OPODIS.2017.27

[30] Juan Garay and Aggelos Kiayias. 2018. SoK : A Consensus Taxonomy in the Blockchain Era. *IACR Cryptology ePrint Archive* (2018), 754.

[31] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. 2011. Efficient middleware for byzantine fault tolerant database replication. In *EuroSys*. 107. DOI: http://dx.doi.org/10.1145/1966445.1966456

[32] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *SOSP*. 51–68. DOI: http://dx.doi.org/10.1145/3132747.3132757

[33] L.M. Goodman. 2014. Tezos — a self-amending crypto-ledger White paper. *CoRR* (2014). https://www.tezos.com/static/papers/white_paper.pdf

[34] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The Next 700 BFT Protocols. In *ECCS*. 363–376. DOI: http://dx.doi.org/10.1145/1755913.1755950

[35] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. Scalable Byzantine Reliable Broadcast. In *DISC*. DOI: http://dx.doi.org/10.4230/LIPIcs.DISC.2019.22

[36] Mike Hearn. 2016. Corda: A distributed ledger. *CoRR* (2016). https://docs.corda.net/releases/release-V3.1/_static/corda-technical-whitepaper.pdf

[37] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. 2018. StreamChain: Do Blockchains Need Blocks? *CoRR* (2018). https://arxiv.org/abs/1808.08406

[38] Jeromy Johnson. 2019. libp2p/go-libp2p-pubsub: A pubsub system built on libp2p. (2019). https://github.com/libp2p/go-libp2p-pubsub

[39] Peter J Keleher. 1999. Decentralized Replicated-Object Protocols. In *PODC*. 143–151.

[40] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake

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

[41] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund L. Wong. 2007. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, Vol. 41. 45–58. DOI: http://dx.doi.org/10.1145/1658357.1658358

[42] Leslie Lamport and David Peleg. 2011. Brief announcement: Leaderless Byzantine Paxos. In *DISC*. 141–142.

[43] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*. 265–278.

[44] Haonan Lu, Kaushik Veeraraghavan, Phillipe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: Measuring and understanding consistency at Facebook. In *SOSP*. 295–310. DOI: http://dx.doi.org/10.1145/2815400.2815426

[45] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. 2011. Byzantine fault-tolerant transaction processing for replicated databases. In *NCA*. 83–90. DOI: http://dx.doi.org/10.1109/NCA.2011.19

[46] Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2019. Flexible Byzantine Fault Tolerance. *CoRR* (2019). https://arxiv.org/abs/1904.10067

[47] Dahlia Malkhi and Michael Reiter. 1998. Byzantine quorum systems. *Distributed Computing* 11, 4 (1998), 203–213. DOI: http://dx.doi.org/10.1007/s004460050050

[48] Dahlia Malkhi, Michael Reiter, Avishai Wool, and Rebecca N. Wright. 1998. Probabilistic Byzantine quorum systems. In *PODC*. DOI: http://dx.doi.org/10.1145/277697.277781

[49] Jean Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine consensus. *TDSC* 3, 3 (2006), 202–215. DOI: http://dx.doi.org/10.1109/TDSC.2006.35

[50] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *CCS*. 31–42. DOI: http://dx.doi.org/10.1145/2976749.2978399

[51] Mininet Team. 2019. Mininet: An Instant Virtual Network on your Laptop (or other PC). (2019). http://mininet.org/

[52] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. 2010. Turquois: Byzantine consensus in wireless ad hoc networks. In *DSN*. IEEE, 537–546. DOI: http://dx.doi.org/10.1109/DSN.2010.5544268

[53] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. 2006. Randomized intrusion-tolerant asynchronous services. In *DSN*. 568–577. DOI: http://dx.doi.org/10.1109/DSN.2006.60

[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

[55] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *CoRR* (2008). https://bitcoin.org/bitcoin.pdf

[56] Beth Simone Noveck. 2009. *Wiki government: how technology can make government better, democracy stronger, and citizens more powerful.* Brookings Institution Press.

[57] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *NSDI*.

[58] Fernando Pedone and Nicolas Schiper. 2011. Byzantine fault-tolerant deferred update replication. In *LADC*. DOI: http://dx.doi.org/10.1007/s13173-012-0060-z

[59] Serguei Popov. 2018. The Tangle. *CoRR* (2018). https://www.iota.org/research/academic-papers

[60] Pavel Raykov, Nicolas Schiper, and Fernando Pedone. 2011. Byzantine fault-tolerance with commutative commands. In *OPODIS*. 329–342. DOI: http://dx.doi.org/10.1007/978-3-642-25873-2_23

[61] Technicolor Research. 2019. PnyxDB - An experimental democratic Byzantine Fault Tolerant datastore for consortia. (2019). https://github.com/technicolor-research/pnyxdb

[62] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. 2019. Scalable and Probabilistic Leaderless BFT Consensus through Metastability. *CoRR* (2019). https://arxiv.org/abs/1906.08936

[63] Marc Shapiro and Nuno Preguiça. 2011. Conflict-free replicated data types. In *SSS*. 386–400.

[64] Atul Singh, Pedro Fonseca, and Petr Kuznetsov. 2009. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *NSDI*. 169–184.

[65] João Sousa, Alysson Bessani, and Marko Vukolić. 2018. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *DSN*. 51–58. DOI: http://dx.doi.org/10.1145/3152824.3152830

[66] Pierre Sutra, Marc Shapiro, and João Pedro Barreto. 2006. An asynchronous, decentralised commitment protocol for semantic optimistic replication. *CoRR* (2006). https://arxiv.org/abs/0612086

[67] The Go Authors. 2019. The Go programming language. (2019). https://golang.org/

[68] Werner Vogels. 2008. Eventually consistent. *Queue* 6, 6 (2008), 14–19.

[69] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. 2019. Potential Risks of Hyperledger Fabric Smart Contracts. In *IWBOSE*. IEEE. DOI: http://dx.doi.org/10.1109/IWBOSE.2019.8666486