Clairvoyant State Machine Replications

Rida Bazzi[1(✉)] and Maurice Herlihy[2]

[1] Arizona State University, Tempe, AZ, USA
bazzi@asu.edu
[2] Brown University, Providence, RI, USA
mph@cs.brown.edu

Abstract. We propose a new protocol for the generalized consensus problem in asynchronous systems subject to Byzantine server failures. The protocol solves the consensus problem in a setting in which information about conflict between transactions is available (such information can be in the form of transaction read and write sets). The use of non-skipping timestamps permits servers to commit transactions as soon as they know that no conflicting transaction can be ordered earlier. Unlike most prior proposals (for generalized or classical consensus), which use a leader to order transactions, this protocol is leaderless, and relies on non-skipping timestamps for transaction ordering. Being leaderless, the protocol does not need to pause for leader elections. For n servers of which f may be faulty, this protocol requires $n > 4f$.

1 Introduction

A *distributed ledger* is a distributed data structure, replicated across multiple *nodes*, where *transactions* from clients are published in an agreed-upon total order. Today, Bitcoin [25] is perhaps the best-known distributed ledger protocol.

There are two kinds of distributed ledgers. In *permissionless* ledgers, such as Bitcoin, any node can participate in the common protocol by proposing transactions, and helping to order them. In *permissioned* implementations, by contrast, a node must be authorized before it can participate. Permissionless ledgers make sense for cryptocurrencies which seek to ensure that nobody can control who can participate. Permissioned ledgers make sense for structured marketplaces, such as financial exchanges, where parties do not necessarily trust one another, but where openness and anonymity are not goals. State machine replication [32] is the most common way to implement permissioned ledgers.

In state machine replication, a total order is agreed upon for all transactions and every server replica executes the transactions in the same order. If two successive transactions commute, the two transactions can be executed in different orders by different servers. To determine if two transactions commute, we can check if the state variables accessed for reading or writing (*read* and *write* sets) by one transaction are written to by the other transactions and vice-versa. Existing state machine replication protocols are limited in their ability to exploit transaction commutativity. Protocols that exploit general transaction

T. Izumi and P. Kuznetsov (Eds.): SSS 2018, LNCS 11201, pp. 254–268, 2018.
https://doi.org/10.1007/978-3-030-03232-6_17

commutativity solve what is called the *generalized consensus* problem in which a dependency structure is assumed on the transactions [18,28]. Published work on generalized consensus, [18,28,29,33] with few exceptions, is limited to systems with servers subject to crash failures. Pires et al. [30] propose a leader-based generalized state machine replication algorithm and Abd-El-Malek et al. [1] propose a client-driven quorum-based protocol called Q/U that is very efficient under low contention, but that requires $n > 5f$ and can suffer from livelock due to contention even in synchronous periods.

The contribution of this paper is a novel permissioned ledger algorithm, which we call *Byblos*. Byblos has three properties of interest.

- **Generalized**. Byblos exploits *semantic* knowledge about client requests to reduce transaction latency. Client transactions include statically-declared read and write sets. The technical key to effectively exploiting semantic knowledge is a novel use of *non-skipping timestamp* [5] to bound the set of in-flight transactions that might end up ordered before a particular transaction. If an otherwise-complete transaction does not conflict with any of its potential predecessors, that transaction can be committed without further delay. For loads with few conflicts, solution for generalized consensus can be much more efficient than solutions for traditional consensus [18].
- **Leaderless**. Byblos is *leaderless*. With some exceptions [1,9,21], prior replicated state machine algorithms use a *leader* to order client requests. Leader-based algorithms typically have two kinds of phases: a relatively simple normal phase where the leaders send and receive messages to the others, and a complicated reconciliation phase [11,16,20,34] used to detect and replace faulty leaders. Leader election comes at a cost: client requests are typically blocked during leader election even in periods of synchrony. Such delays are especially problematic if valuable periods of synchrony are spent electing leaders instead of making progress. (Other leaderless protocols, such as EPaxos [23], make similar observations.)
  In Byblos, transactions are guaranteed to terminate in periods of synchrony. Technically, Byblos does not need a leader because it is centered around a leaderless non-skipping timestamp algorithm.
- **Simple**. Byblos is simple to explain and understand. While simplicity is subjective, readers who are familiar with other protocols for Byzantine fault tolerance will note that the full protocol is described in this paper.

In Byblos, transactions are ordered by timestamp, with ties broken canonically. For a given timestamp value t, Byblos can determine an upper bound on the set of in-flight *pending* transactions that *might* have assigned timestamp t. This ability to bound the set of potentially conflicting pending transactions makes Byblos clairvoyant. If a transaction T with timestamp t is the next one to be executed by a replica among those transactions with timestamp t, and T does not conflict with any of the *pending* transactions, then T can be executed without waiting for the status of the pending transactions to be resolved. Byblos guarantees progress using an "off-the-shelf" underlying asynchronous Byzantine

agreement algorithm, preferably early deciding [7,35], to CANCEL or COMMIT pending transactions[1].

Byblos tolerates $f < n/4$ faulty servers, assuming the underlying consensus algorithm does the same. If there are no conflicts between pending transactions and transactions waiting for execution, Byblos can make progress even in periods of complete asynchrony. This does not contradict the FLP impossibility [15].

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the problem and the system model. Section 4 gives a detailed description of Byblos and Sect. 5 states the theorems for correctness. Section 6 describes how the protocol can be optimized to eliminate the exchange of whole pending sets and Sect. 7 describes how Byzantine clients can be tolerated. Section 8 discusses the performance.

2 Related Work

Leader-based distributed ledgers such as Paxos [17] and Raft [27] do not exploit knowledge of read-write sets to reduce latency and increase throughput. Distributed ledgers that do exploit such information include Generalized Paxos [18], Egalitarian Paxos [23], Hyperledger Fabric [10], NEO [26], and Bitcoin itself [25].

There is a large body of literature on state machine replication, most of which is leader-based. Clement *et al.* [12] observe that many Byzantine fault-tolerant (BFT) protocols can perform poorly in the presence of Byzantine failures. They define the notion of a *fragile optimization*, where a single misbehaving party can knock the system off an optimized path. They also define *gracious* (synchronous, non-faulty) and *uncivil* (synchronous, limited Byzantine faults) executions. They argue that while most BFT protocols are optimized for *gracious* executions, it is also important that protocols perform well in uncivil executions. They propose *Aardvark*, a BFT protocol designed to perform well under uncivil executions. Aardvark uses a leader, with regularly-scheduled view changes. The protocol includes safeguards against censorship by the leader. Amir *et al.* [2] introduce *bounded delay* as a performance goal for BFT protocols. They introduced *Prime*, a BFT protocol that uses a leader that is monitored by other servers to provide bounded delay in the presence of limited Byzantine failures.

Paxos [17] and Raft [27] are perhaps the best-known non-Byzantine replication protocols. Other Paxos-related non-Byzantine protocols include *Mencius* [19] and *EPaxos* [23]. These protocols, with the exception of EPaxos [23], use some form of leader (or leaders) and view changes. Milosevic *et al.* [22] proposed a *BFT-Mencius* which also uses performance monitoring and view changes to limit the effects of slow servers. Byblos does not use view changes or performance monitoring and hence allows unbounded variance below the timeout threshold.

Existing protocols that perform relatively well under uncivil executions, perform less well in civil executions, compared to protocols optimized only for civil executions. Byblos is different. Its latency, measured in the number of message

[1] Asynchronous consensus algorithms are those that guarantee safety at all times, and progress under eventual synchrony.

round trips, is comparable to protocols optimized for civil execution. In the absence of slow or faulty clients, its latency in uncivil executions is also comparable to that of protocol optimized for civil executions. On the down side, Byblos uses signatures, whereas some protocols use faster message authentication codes.

Many BFT protocols that do not exploit commutativity. BFT protocols that do not use leaders or view changes include *HoneyBadgerBFT* [21]. Unlike most BFT protocols, HoneyBadgerBFT does not assume eventual (or partial) synchrony, but relies on a randomized atomic broadcast protocol with a cryptographic shared coin. HoneyBadgerBFT ensures censorship resistance through a cryptographic subprotocol. Unlike Byblos, HoneyBadgerBFT does not exploit transaction semantics. The *RBFT* BFT protocol [4] uses multiple leaders, who track one another, and provide censorship resistance. It is designed for systems in which clients can have multiple parallel pending requests. Aublin *et al.* [3] describe a family of protocols, some of which have low (2-message) latency in synchronous executions.

As noted, the protocols discussed, with the exception of EPaxos [23] which only tolerates crash failures, do not solve the generalized consensus problem [18, 28]. Abd-El-Malek et al. [1] propose a client-driven quorum-based protocol called Q/U that is very efficient under low contention, but that requires $n > 5f$ and can suffer from livelock due to contention even in synchronous periods. The algorithm is leaderless and uses exponential backoff in the presence of contention. Other work that aims at improving Q/U reverts to using a leader [13]. Recently Pires et al. [30] proposed a leader-based Byzantine version of generalized Paxos.

In general, faulty clients in Byblos can force servers to revert to an "off-the-shelf" binary Byzantine consensus protocol to resolve the outcome of "stuck" transactions. Triggering the agreement protocol might incur a timeout which can be significantly larger than typical communication delay even for fast protocols (for example, Ben-Or *et al.* [8] or Mostefaoui *et al.* [24]). It might seem that Byblos replaces one source of delay (faulty leader) with another (faulty clients), but this replacement allows us to exploit transaction semantics which can be a significant improvement in some settings. In systems in which faulty servers can delay the processing of transactions (which is almost all systems), everyone is delayed. (These issues are discussed in Sect. 8.)

3 Problem and System Model

A *ledger* (Fig. 1) can be thought of as an automaton consisting of a set of *states* (for example, clients' account balances), a set of deterministic state transitions called *transactions* (for example, deposits, withdrawals, and transfers), and a *log* recording the sequence of transactions. The state is needed to efficiently compute transactions' return values (for example, your account is overdrawn). The log provides an audit trail: one can reconstruct any prior state of the ledger, and trace who was responsible for each transaction.

Our solution encompasses the following components. There are n *servers* that maintain the ledger's long-lived state via a set of replicated tamper-proof logs.

```
1  state = initialState
2  log = []
3  while true:
4    on receive T from c:
5      log.append(T)
6      state, result = apply(T, state)
7      send result to c
```

Fig. 1. Ledger abstraction

Up to f of $n = 4f + 1$ servers may be Byzantine (capable of departing from the protocol). The rest of the servers are *honest*. The logs of honest servers are only modified by appending new transactions. The servers satisfy the following safety property: for any pair of honest servers, one server's log is a prefix of the other's. It follows that honest servers execute all transactions in the same order.

There is a potentially unbounded number of *clients* who originate transactions. It is the servers' job to accept transactions from clients, order them, and publish this order. We assume that the clients are not Byzantine; in Sect. 7 we explain how to handle Byzantine clients.

Communication is handled by an underlying message-diffusion system. Clients broadcast messages, which are eventually delivered to all honest servers. All messages are signed, and cannot be forged. Servers communicate with one another though the same diffusion infrastructure.

The ledger state is a key-value store. Each client transaction declares a *read set*, the set of keys it might possibly read, and a *write set*, the set of keys it might possibly write. Any transaction that violates its declaration is rejected. (Systems such as Generalized Paxos [18], Egalitarian Paxos [23], and NEO [26] all make use of similar conflict declarations.)

4 Byblos Description

In Byblos, transactions are assigned integer *timestamps*, which partially determine the order in which they are applied. If two transactions do not overlap in time, the later one will be assigned the later timestamp, but overlapping transactions may be assigned the same timestamp. The timestamps assigned to transactions are non-skipping [5]. This means that if a timestamp t is assigned to a transaction, then every timestamp whose value is less than t must have been previously assigned to some other transaction.

The non-skipping timestamp protocol [5] at the heart of the algorithm is simple. A client broadcasts a timestamp request to the servers, and collects at least $n - f$ timestamps in response. The client selects the $(f + 1)^{\text{st}}$ latest timestamp, which is guaranteed to be less than or equal to the latest timestamp assigned to any transaction. The client increments that timestamp by one, and later broadcasts it to the servers. It can be shown [5] that this way of choosing timestamps ensures that no timestamp values are skipped.

The properties of non-skipping timestamps suggest a simple way for servers to execute transactions in a deterministic order in the absence of client failures. For each timestamp value t, starting with 0, execute all transactions whose timestamp is t in a deterministic order. Once all transactions with timestamp t are executed, transactions with timestamp $t + 1$ can be executed and so on. This seems too simple (even in the absence of client failures) and indeed it is. The catch is that servers will need to be able to determine when all transactions with a given timestamp value have been received.

To determine when all transactions with timestamp t have been received, Byblos calculates for each transaction T a set of *pending* transactions: transactions that were detected to be concurrent with T. The set of pending transactions contains transactions, but not their assigned timestamps, because those timestamps might not be determined at the time a transaction is added to a pending set. The crucial property is the following: if a transaction T has timestamp t, then its set of pending transactions is guaranteed to contain all transactions whose assigned timestamp will be t. However, it may also include transactions whose assigned timestamp will be larger than t. With the set of pending transaction, in the absence of client failures, servers can execute all transactions in a deterministic order as follows. If there are no pending transactions that conflict either with T or with any transaction with the same timestamp ordered before T, then T can be executed. Eventually, T will be executed when the timestamps of all conflicting transactions in T's pending set become known.

The description so far assumes no client failures. If clients can fail, some transactions in the pending set might never complete and the servers will be stuck, unable to determine when all potentially conflicting transactions with timestamp t have been received. We resolve this situation by executing a binary consensus algorithm, over the values COMMIT and CANCEL, to resolve the fates of orphaned transactions. Each client tries to commit its own transaction using the consensus algorithm, and servers try to cancel pending set transactions that are slow to arrive (with their timestamp). We can use any "off-the-shelf" consensus algorithm guaranteed to terminate if the system is eventually synchronous, including known algorithms that terminate in one round if the system is well-behaved [35]. Transaction execution proceeds as follows. Once all conflicting transactions in the pending set of some transaction with timestamp t are either cancelled or committed, the set of transactions with timestamp t is also known. The execution can then proceed as outlined above for all transactions that have not been cancelled.

The protocol guarantees safety at all times and liveness under eventual synchrony [11]. The rest of this section describes the client and server code in details.

4.1 Client Code

The client code (Fig. 2) proceeds in three stages. In the first stage (Lines 6–12), the client sends a Propose message to all servers, and collects at least $n - f$ ProposeAck responses. The client calculates $\hat{t}$ which is equal to 1 plus the $(f+1)^{\text{st}}$ largest amongst the timestamps it received and assigns it to its transaction. It

```
1   received = ∅ // messages from servers
2   ∀ s in Servers timestamp[s] = 0
3   ∀ s in Servers proposed[s] = ∅
4
5   // get valid timestamp
6   broadcast Propose(T) to Servers
7   repeat
8       on receive m: received = received ∪ {m}
9   until ProposeAck(T, t) received from ≥ n − f servers
10  ∀ s in Servers: if ∃ t : ProposeAck(T, t) received from s
11      timestamp[s] = t
12  t̂ = ((f + 1)st largest value in timestamp[∗]) + 1
13
14  // broadcast timestamp, get set of pending Txns
15  broadcast Confirm(T,t̂) to Servers
16  repeat
17      on receive m: received = received ∪ {m}
18  until ConfirmAck messages received from ≥ n − f servers
19  ∀ s ∈ Servers: if ConfirmAck(T, txns) received from s
20      pending = pending ∪ txns
21
22  // broadcast pending set, get transaction result
23  broadcast Resolve(T, t̂, pending) to Servers
24  repeat
25      on receive m: received = received ∪ {m}
26  until ≥ f + 1 identical ResolveAck(T,code,result) messages received
27  if code == COMMIT (in ≥ f + 1 messages) then
28      return result
29  else:
30      return ⊥
```

Fig. 2. Client code

is important to note that this particular way of choosing timestamps is what guarantees timestamps to be non-skipping. In the second stage (Line 15–20), the client broadcasts a Confirm message with $\hat{t}$, waits for at least $n - f$ responses, each containing a set txn of transactions that have been proposed at a server, and calculates the set pending, which is the union of these sets. The set pending is guaranteed to contain every transactions whose timestamp is less than or equal to $\hat{t}$. We implicitly assume that the client verifies responses for well-formedness, and for authenticity by checking signatures. In the third stage (Line 23–30), the client broadcasts a Resolve message with the set pending, and waits to receive $f + 1$ identical ResolveAck responses to determine the transaction's outcome. A ResolveAck message has three fields: (1) the transaction, (2) a code, either COMMIT or CANCEL, and (3) a result. If the return code is COMMIT, the call was successful, and the result is returned, otherwise a failure indication is returned.

```
state = initialState
clock = 0
proposed = ∅          // Set transaction
pending[*] = ∅        // timestamp ↦ Set transaction
confirmed[*] = ∅      // timestamp ↦ Set transaction
committed = ∅         // timestamp ↦ Set transaction
cancelled = ∅         // Set transaction
resolving = ∅         // Set transaction
log = []              // Sequence transaction
timer[*] = ∞          // timestamp ↦ real time timer
time : real time clock value at a server
```

Fig. 3. Server state with initializations

4.2 Server Code

Server State. The server state (Fig. 3) is composed of the following fields.

- state is the ledger state. A transaction is applied to state when it commits.
- clock is an integer counter that tracks the latest timestamp assigned to a transaction. We assume this counter does not overflow.
  Since timestamps are non-skipping, a 128-bit counter should be more than sufficient in practice.
- proposed is set of transactions that have been proposed. When a transaction is added to proposed, its timestamp might not be known.
- pending is a map from timestamps to sets of transactions. For timestamp t, pending[t] is the set of transactions that might be assigned timestamp t.
- confirmed is a map from timestamps to sets of transactions. For timestamp t, confirmed[t] is the set of known transactions that will either commit with timestamp t or will be cancelled.
- committed is set of transactions known to have committed.
- cancelled is the set of transactions known to be cancelled.
- log is the sequence of committed transactions.
- timer is an array of timers used to timeout pending transactions.
- time is a local clock at the server to measure real time for timeouts. The local clocks of servers are independent and need not be synchronized.

Server Actions. The server continually receives messages (Fig. 4). When it receives a message from client c, it does the following. For Propose(T) (Lines 2–4), it adds T to proposed, and returns the current clock value to the client.

For Confirm(T,$\hat{t}$) (Line 6–12), the server advances clock to the maximum of $\hat{t}$ and its current value, and adds the transaction to the set of confirmed transactions. The server also launches a consensus protocol with the other servers to try to to COMMIT T. Then, it returns the current proposed set to the client.

For Resolve(T,$\hat{t}$, txns) (Lines 14–29), the server adds the set of concurrent transactions, txns, to the pending set. If this is a Resolve for a new transaction,

```
1  repeat
2     if Propose received from c:
3        proposed = proposed ∪ {T}
4        send ProposeAck(T, clock) to c
5
6     if Confirm(T, t̂) received from c:
7        clock = max(clock, t̂)
8        confirmed[t̂] = confirmed[t̂] ∪ {T}
9        if T ∉ resolving :
10           resolving = resolving ∪ {T}
11           fork ResolveThread(T, COMMIT)      // try to commit this transaction
12       send ConfirmAck(T, proposed) to c
13
14    if Resolve(T, t̂, txns) received from c or s:
15       if pending[t̂] = ∅:
16          pending[t̂] = txns
17          send Resolve(T, t̂, txns) to all servers
18          timer[t̂] = time+δ
19
20    if StartResolution (T, code) received from s:  // join another consensus
21       if T ∉ resolving :                          // code is either COMMIT or CANCEL
22          resolving = resolving ∪ {T}
23          fork ResolveThread(T, code);
24
25    if timer[t] expired for some t:                // if timer expired, try to cancel
26       for each txn in pending[t]:
   // any obstructing transactions for
27          if txn ∉ resolving :
28             resolving = resolving ∪ {txn}
29             fork ResolveThread(txn, CANCEL)
30
31    ApplyResolvedTransaction()                     // attempt apply transactions
32 until false
```

Fig. 4. Server code

the server propagates the resolve message in case other servers do not hear directly from the client. At this point, at least $2f+1$ servers must have initiated a consensus protocol to commit T (a client does not send a resolve message until it has received $n-f$ confirm messages). The only remaining point that can obstruct T's execution are pending transactions. So, the server sets a timer to give pending transaction the chance to arrive without being timed out. If the timer expires and a transaction in the pending set is not confirmed, the transaction is considered obstructing and an attempt is made to CANCEL it. In practice the delay can be increased dynamically to guarantee that eventually it reaches a value that works for periods of synchrony [20].

We assume that the consensus protocol executed by a server adds T to the set committed if the server decides to COMMIT and adds T to the set cancelled if it decides to CANCEL the transaction. We also assume that the first message sent by the consensus protocol is a StartResolution (T,code) message which lets

```
1   OrderBefore(T,T') =
2         T ∈ confirmed[t] ∧
3         ((T' ∈ confirmed[t'] ∧ t < t') ∨ (T' ∈ confirmed[t] ∧ T < T') ∨
4          (T' ∈ pending[t'] ∧ ∀t'' < t : T' ∉ pending[t''] ∧  t < t') ∨
5          (T' ∈ pending[t] ∧ ∀t'' < t : T' ∉ pending[t'']  ∧ T < T'))
6     ∨
7         T ∈ cancelled ∧
8         ((T' ∉ cancelled ) ∨ (T' ∈ cancelled ∧ T < T'))
9     ∨
10        (T ∈ pending[t] ∧ ∀t' < t : T' ∉ pending[t']) ∧
11        ((T' ∈ confirmed[t] ∧ conflict (T,T') ∧ T < T') ∨
12         (T' ∈ confirmed[t'] ∧ conflict (T,T') ∧ t < t'))
13
14  ApplyResolvedTransaction()
15     if ∃ T: T ∈ committed ∧ ∀ T': OrderBefore(T',T) ⇒ T' ∈ cancelled ∨ T' ∈ log
16        log.append(T)
17        state, result = apply(T, state))
18        send ResolveAck(T,COMMIT,result) to T sender
19     if ∃ T: T ∈ cancelled ∧ T ∉ log
20        log.append(T)
21        send ResolveAck(T,CANCEL,⊥) to T sender
```

Fig. 5. Applying resolved transactions

a server that has not heard directly from a client join the consensus for a given transaction (Lines 20–23).

Finally, a server attempts to apply transactions (Fig. 5). For every timestamp t, we have three groups of transactions: (1) those that have been committed, (2) those that have been cancelled, and (3) those that are pending. For a pending transaction we assign it to the timestamp t for which it first appeared in a pending set. Note that it is possible that pending transactions might appear in groups with different timestamps at different honest servers, but if they become committed, they will have the same timestamp at all honest servers, and if they are cancelled, they will be cancelled by all honest servers. Servers order all transactions according to their timestamp and for a given timestamp, the transaction (in all three groups) are ordered by taking a hash of the transaction request. The OrderBefore() predicate is used to determine the order of transactions at a given time. A transaction that is confirmed is ordered before another confirmed transaction if it has a smaller timestamp or the same timestamp but $T < T'$ in the canonical order (Line 3). A transaction that is confirmed with timestamp t is ordered before another pending transaction whose first appearance in a pending set is for timestamp t' if $t < t'$ or $t = t'$ but $T < T'$ in the canonical order (Lines 4–5). A transaction that is cancelled is ordered before any other transaction because such transactions do not conflict with other transactions (Lines 6–7). A transaction T that is pending is ordered before another confirmed transaction T' if the first timestamp for which T is pending is the same as the timestamp for T', the two transactions conflict and either T appears before T' in the canonical order or the timestamp of T is smaller than that of T'.

5 Correctness

Safety and progress are established by the following lemmas and theorems. Proofs are omitted for lack of space.

Lemma 1 (Same order for applied transaction). *If two honest servers apply two non-cancelled transactions T_1 and T_2 to the log, they apply them in the same order.*

Lemma 2 (Agreement on committed transaction). *If an honest server decides to commit or cancel a transaction, then every honest server eventually makes the same decision.*

Theorem 1 (Linearizability). *The implementation is linearizable.*

Theorem 2 (Progress in periods of synchrony). *In periods of synchrony, all transactions of correct clients are applied.*

6 Eliminating Pending Sets

For ease of exposition, the protocol as presented so far requires clients and servers to exchange proposed and pending sets that can grow without bounds. We explain how the protocol can be modified to eliminate the exchange of these sets.

At a given correct server, the pending set of a confirmed transaction T with timestamp $\hat{t}$ is the set of all previously proposed transactions received by the time the server receives the Confirm message for T. This is the txns set that the client receives from the server then propagates as part of the pending set.

Instead of sending pending sets to clients, every server sends to every other server confirmed transactions (with their timestamps) and proposed transactions that it receives in the order in which they are received together with the clock value at the time they are received. This is done once for every proposed and confirmed transaction. The pending set for T can be given by the formula

$$\mathsf{pending}_T = \bigcup_{s:s\in S\wedge|S|\geq n-f} \mathsf{previous}(s,\mathsf{T})$$

where $\mathsf{previous}(s,\mathsf{T})$ is the set of proposed transactions received before receiving the Confirm message for T. In the original protocol, the client itself collects $n-f$ previous, which are simply the proposed sets. In the modified protocol, the servers can only be guaranteed to receive $n-2f$ previous sets because f correct servers might not have heard from the client and another f faulty servers might deny having received the Confirm message for T. This can be easily fixed by requiring every server to treat a Confirm message forwarded by another server as a Confirm received from the client (if it has not previously received it). This way, we guarantee that every server can calculate a pending set for every timestamp.

7 Byzantine Clients

The solution as presented assumes clients fail by crashing. Also, it assumes that some implicit checks are done by clients. For instance, it is possible for a Byzantine server to send some fake pending transactions. We assume that the server provides proof that all transactions in a pending set were indeed received by the server. Conversely, when the client send a pending set to the servers, it can be required to provide proof in the form of signatures that every transaction in the set was indeed received by a server. Similarly, the client should provide proof at the calculated hash is justified based on the individual timestamp received from servers. Avoiding replay attacks is straightforward by having the servers sign a cryptographic hash of the messages they send to the clients. These messages include the transaction identifiers.

8 Performance

To evaluate the performance of our solution, we adopt the definitions of *gracious* and *uncivil* executions from Clement et al. [12].

Definition 1 (Gracious execution [12]). *An execution is* gracious *if and only if (a) the execution is synchronous with some implementation-dependent short bound on message delay and (b) all clients and servers behave correctly.*

Definition 2 (Uncivil execution [12]). *An execution is* uncivil *if and only if (a) the execution is synchronous with some implementation-dependent short bound on message delay, (b) up to f servers and an arbitrary number of clients are Byzantine, and (c) all remaining clients and servers are correct.*

8.1 Performance in Gracious Executions

In gracious execution, and in the *absence of contention*, the protocol requires 3 round-trip message delay from the time a client makes a request to the time it gets the result. It takes one round-trip delay to receive the first response and calculate the timestamp $\hat{t}$. It takes 1/2 round-trip delay for the servers to receive $\hat{t}$. At that time correct servers initiate a consensus protocol to commit the transaction and another one round-trip delay is needed to decide to COMMIT the transaction (this is possible because all correct servers will be proposing the same COMMIT value). The client replies to the confirm message after two round-trip delays and gets a response to its resolve message after 3 round-trip delays (there is no need to wait for the result of the consensus which will arrive at the same time as the resolve message).

In the presence of contention, the processing can be delayed by conflicting transactions that have the same timestamp. The latest a transaction started after T can get the same timestamp as T is just short of 1.5 round-trip delay from the time T started (we assume that previous transactions that are not concurrent

with T have already been cleared). In fact, a transaction that starts 1.5 round-trip delay after T cannot reach the servers before the time T's timestamp is propagated and will get a later timestamp (we are assuming that the Propose message for the contending transaction will propagate instantaneously in the worst case). So, in the presence of contention, a response might not arrive before 4.5 round-trip delays.

We expect that a closer integration of the solution with a particular consensus protocol will further reduce the delay by another one half of a round-trip which would make it more competitive in terms of latency (PBFT [11] achieves 2 round-trip delay with a number of optimizations including speculative execution, but PBFT does not perform well in uncivil executions).

8.2 Performance in Uncivil Executions

In uncivil executions, the delay depends on the level of contention. If a transaction is initiated and is not overlapping with any other conflicting transaction, its delay will be the same as in gracious executions.

In the presence of contention, a transaction can be delayed further. As in the gracious execution case, we consider the latest time a transaction can be added to the pending set of transaction T. As in the case of gracious executions, the time is 1.5 round-trip delay after T is initiated. If the client of the contending transaction fails, the full timeout would need to be incurred and a consensus protocol would need to be executed. So, the delay in this case would be the timeout value δ plus the consensus time. The client will get a response by 0.5 round-trip delay after the consensus has ended (because the other message exchanges of the client overlap with the timeout time).

8.3 Other Performance Considerations

It is important to note that the delays are not additive. If we have transactions with different timestamps and for each timestamp there is a pending transaction that is slow, no transaction incurs more than one timeout plus consensus delay because the timers are started in a pipelined fashion. This ensures that Byblos average throughput under client delays is minimally affected by slow clients. Also, recall that this delay is only incurred by conflicting transactions whereas in systems in which faulty servers are the source of the delay, all transactions are affected by server delays.

Another potential performance improvement that we did not consider is *transaction batching* [11]. In our solution, servers communicate information about individual transactions. On the positive side, in Byblos, in the presence of contention, more transactions will get the same timestamp and the delay incurred for that timestamp is one for all transactions. This should improve throughput.

As described, Byblos uses public-key signatures [14,31], which can add significant overhead. Replacing signatures with message authentication codes [6] is a subject for future work. Finally, the message complexity of our solution is

rather high: $O(n^2)$ messages per transaction. Such high message complexity is not unusual for protocols that aim to achieve bounded delay ([2,4,12,22] for example).

References

1. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. ACM SIGOPS Oper. Syst. Rev. **39**(5), 59–74 (2005)
2. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Prime: Byzantine replication under attack. IEEE Trans. Dependable Secur. Comput. **8**(4), 564–577 (2011)
3. Aublin, P.L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 BFT protocols. ACM Trans. Comput. Syst. **32**(4), 12:1–12:45 (2015)
4. Aublin, P.L., Mokhtar, S.B., Quéma, V.: RBFT: redundant Byzantine fault tolerance. In: Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems, pp. 297–306 (2013)
5. Bazzi, R.A., Ding, Y.: Non-skipping timestamps for Byzantine data storage systems. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 405–419. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30186-8_29
6. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_1
7. Ben-Or, M.: Another advantage of free choice (extended abstract): completely asynchronous agreement protocols. In: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, pp. 27–30. ACM (1983)
8. Ben-Or, M., Kelmer, B., Rabin, T.: Asynchronous secure computations with optimal resilience (extended abstract). In: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 183–192. ACM, New York (1994)
9. Borran, F., Schiper, A.: A leader-free Byzantine consensus algorithm. In: Kant, K., Pemmaraju, S.V., Sivalingam, K.M., Wu, J. (eds.) ICDCN 2010. LNCS, vol. 5935, pp. 67–78. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11322-2_11
10. Cachin, C.: Architecture of the hyperledger blockchain fabric. In: Workshop on Distributed Cryptocurrencies and Consensus Ledgers (2016)
11. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
12. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine fault tolerant systems tolerate Byzantine faults. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, pp. 153–168 (2009)
13. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 177–190 (2006)
14. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Inf. Theory **31**(4), 469–472 (1985)
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM (JACM) **32**(2), 374–382 (1985)
16. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 45–58. ACM (2007)

17. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
18. Lamport, L.: Generalized consensus and paxos. Technical report, Microsoft, March 2005
19. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machines for WANs. In: Proceedings of the 8th OSDI Conference, pp. 369–384 (2008)
20. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Trans. Dependable Secur. Comput. **3**(3), 202–215 (2006)
21. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of BFT protocols. In: ACM CCS, pp. 31–42 (2016)
22. Milosevic, Z., Biely, M., Schiper, A.: Bounded delay in Byzantine-tolerant state machine replication. In: 2013 IEEE 32nd International Symposium on Reliable Distributed Systems, pp. 61–70, September 2013
23. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in Egalitarian parliaments. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 358–372. ACM, New York (2013)
24. Mostefaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous byzantine consensus with $t < n/3$, $O(n^2)$ messages and $O(1)$ expected time. In: 2014 Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, pp. 2–9. ACM (2014)
25. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
26. NEO: Neo contract whitepaper. http://docs.neo.org/en-us/basic/neocontract.html. Accessed 6 May 2018
27. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the USENIX Annual Technical Conference, pp. 305–320 (2014)
28. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distrib. Comput. **15**(2), 97–107 (2002)
29. Peluso, S., Turcu, A., Palmieri, R., Losa, G., Ravindran, B.: Making fast consensus generally faster. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 156–167. IEEE (2016)
30. Pires, M., Ravi, S., Rodrigues, R.: Generalized paxos made Byzantine (and less complex). In: Spirakis, P., Tsigas, P. (eds.) SSS 2017. LNCS, vol. 10616, pp. 203–218. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69084-1_14
31. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
32. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. (CSUR) **22**(4), 299–319 (1990)
33. Sutra, P., Shapiro, M.: Fast genuine generalized consensus. In: 2011 30th IEEE Symposium on Reliable Distributed Systems (SRDS), pp. 255–264. IEEE (2011)
34. Van Renesse, R., Altinbuken, D.: Paxos made moderately complex. ACM Comput. Surv. (CSUR) **47**(3), 42 (2015)
35. Zielinski, P.: Optimistically terminating consensus: all asynchronous consensus protocols in one framework. In: 2006 The Fifth International Symposium on Parallel and Distributed Computing, ISPDC 2006, pp. 24–33. IEEE (2006)