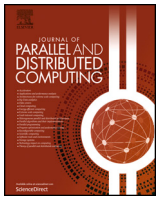
Journal of Parallel and Distributed Computing 187 (2024) 104839

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



Antipaxos: Taking interactive consistency to the next level

Chunyu Mao [a], Wojciech Golab [a,*], Bernard Wong [b]

[a] Department of Electrical and Computer Engineering, University of Waterloo, Canada
[b] David R. Cheriton School of Computer Science, University of Waterloo, Canada

ARTICLE INFO

Keywords:
Consensus
Fault tolerance
Leaderless protocols
Parameterized protocols

ABSTRACT

Classical Paxos-like consensus protocols limit system scalability due to a single leader and the inability to process conflicting proposals in parallel. We introduce a novel agreement protocol, called Antipaxos, that instead reaches agreement on a collection of proposals using an efficient leaderless fast path when the environment is synchronous and failure-free, and falls back on a more elaborate slow path to handle other cases. We first specify the main safety property of Antipaxos by formalizing a new agreement problem called *k-Interactive Consistency* (*k-IC*). Then, we present a solution to this problem in the Byzantine failure model. We prove safety and liveness, and also present an experimental performance evaluation in the Amazon cloud. Our experiments show that Antipaxos achieves several-fold higher failure-free peak throughput than Mir-BFT. The inherent efficiency of our approach stems from the low message complexity of the fast path: agreement on n batches of conflict-prone proposals is achieved using only $\Theta(n^2)$ messages in one consensus cycle, or $\Theta(n)$ amortized messages per batch.

1. Introduction

Consensus [27] has been used extensively to implement state machine replication [45], including in several blockchain systems [2,11,17]. The classical consensus problem entails a set of processes agreeing on a single proposal among multiple proposals, despite the possibility of failures. If the processes and communication network are both synchronous, then there are simple solutions to solve this problem. However, these assumptions are not always satisfied in real world environments, and the FLP impossibility result [16] states that if there is a crash failure, it may not be possible to reach consensus in an asynchronous environment. This inherent limitation can be overcome through the introduction of randomization [4,6,7,38] or stronger synchrony assumptions [10,14,51], but the consensus problem remains notoriously difficult to solve efficiently and correctly.

Although there exist many consensus protocols, most of them are primarily concerned with handling failures and competing proposals instead of scalability. This is accomplished through complex message passing and processing, which imposes overhead. Notably, the problem is traditionally solved using single-leader quorum-based techniques, such as variations of Lamport's Paxos protocol [28]. In such protocols, the leader plays a central role and limits scalability. Using these protocols to implement a replicated state machine (RSM) is inherently costly as every batch of state transitions has to be proposed by the leader and requires at least one dedicated consensus cycle.

To relieve the bottleneck of the leader, some consensus protocols incorporate a fast path mechanism that improves common-case performance, and a slow path to handle other cases, such as failures. For example, Fast Paxos [29] uses an enlarged quorum to bypass the Paxos leader in the absence of competing proposals, though suffers a performance penalty when it falls back on its slow path. Other performance optimizations include rotating the leader [34], as well as computing decisions hierarchically [1]. Some other works involve different types of hardware optimizations [22,41,49]. Zyzzyva [26] additionally offloads processing to clients. Furthermore, batching [44] and pipelining [25,31,43] are two general techniques to increase the efficiency of consensus protocols. However, most of the above techniques still suffer from the limitation of the single-leader approach, and scalability remains a weak point when implementing an RSM.

Recent research investigates leaderless [10–12,36,37] and multi-leader [34,46] designs to increase scalability, which essentially reach agreement on multiple proposals from different proposers in every consensus cycle. In the leaderless approach, every process is a proposer. In contrast, only designated leaders can be the proposers in the multi-leader approach. EPaxos [37] allows all replicas to propose requests concurrently. While non-conflicting proposals can be committed on the

* Corresponding author.
E-mail address: wgolab@uwaterloo.ca (W. Golab).

https://doi.org/10.1016/j.jpdc.2024.104839
Received 6 March 2023; Received in revised form 2 November 2023; Accepted 7 January 2024
Available online 17 January 2024

fast path, concurrent conflicting proposals have to be committed on the slow path after careful analysis of dependencies. Canopus [43] and RCanopus [25] divide the replicas into multiple groups, where each group proposes a batch of requests in one consensus cycle. The final consensus is achieved by interleaving the requests from each group in round-robin order. On the other hand, Mir-BFT [46] uses multiple instances of PBFT [8] to reach agreement on multiple proposals. The major challenge faced by these leaderless and multi-leader techniques is achieving high performance despite the possibility of both conflicting proposals and failures, while relying on minimal synchrony assumptions.

Rising to the above challenge, we make the following technical contributions in this paper:

1. We formalize a new agreement problem in Section 3 called k-interactive consistency (k-*IC*), in both the crash failure model and the Byzantine failure model. k-*IC* combines Interactive Consistency [40] and Vector Consensus [13] with a tunable parameter k that suits different synchrony and failure scenarios that may occur in one system. Compared to traditional agreement problems, k-*IC* requires processes to agree on an ordered collection of values instead of a single value in each consensus cycle.

2. We present Antipaxos (AP), a solution to k-*IC*, in Section 4. Conceptually, AP takes the fast path concept to the next level by using a simple and efficient all-to-all broadcast as the primary means to disseminate multiple proposals concurrently, and resorting to a more complex mechanism only on the slow path to deal with failures and asynchrony. More importantly, when a failure or asynchrony causes some processes to perform fast path execution while others do not, AP still ensures a consistent view. AP solves the k-*IC* problem for two distinct k values under different conditions. We establish the safety and liveness properties of AP under Byzantine failures in Section 5.

3. We implement a prototype of Antipaxos in Golang. We evaluate our prototype in the Amazon cloud against Mir-BFT [46]. The results, which we present in Section 6, show that AP achieves superior peak throughput.

2. Model

The system comprises n processes that communicate by sending and receiving messages over a network of point-to-point authenticated communication channels. Up to f processes may be faulty, meaning that they are corrupted by an adversary and may exhibit Byzantine failures. The communication channels are reliable in the sense that they do not lose, corrupt, duplicate, or reorder messages. The process limit n and failure threshold f are known to all processes. We assume a public key infrastructure (PKI) whereby each process owns a key pair for signing messages and knows the public key of every other process. We denote by $\sigma_x(d)$ the signature created using process P_x's private key on a data item d. The adversary is computationally bounded, and the collection of processes it corrupts cannot forge cryptographic signatures.

The protocol executed by each non-faulty process is modelled as an infinite loop that executes successive *cycles of consensus*, where each cycle is the execution of a collection subprocedures (Propose, Report, and Commit) parameterized by the *cycle number*. For simplicity, we present the protocol in this manuscript for a single cycle and omit the cycle number from the pseudocode. In practice, the cycle number must be embedded in protocol messages and protected by a signature to prevent replay attacks.

Regarding synchrony, the environment may behave differently in different time periods. To model such a realistic environment, we consider two distinct modes of operation: *synchronous mode* and *asynchronous mode*. Normally, all the processes and the network operate in the synchronous mode, in which there is a known upper bound on the message and processing delays, and all processes execute their prescribed protocol faithfully. Any process that suffered a Byzantine failure earlier, for example by crashing or deviating from its protocol, must be repaired by manual intervention prior to the start of synchronous mode. A repaired process is released from the adversary's control, receives a new key pair, and restarts executing the protocol at a consensus cycle loosely synchronized with other processes.[1] Such a process will be considered non-faulty for the purpose of analysis in Section 5. The repair procedure also includes broadcasting the new public key to all processes, and discarding any delivered but unprocessed messages that were signed with the old key pair. The environment transitions to the asynchronous mode if the processing or network delays exceed the upper bound due to asynchrony, or if a failure occurs, and may eventually transition back to synchronous mode. In both modes, processes have access to local clocks that can be used to compute timeouts for unreliable failure detection.

Since the consensus problem is not solvable in a fully asynchronous environment in the presence of failures, we assume the existence of a classical consensus protocol that guarantees safety at all times while maintaining liveness if the environment is in synchronous mode for a sufficiently long period of time. This protocol is used to implement a crucial building block of Antipaxos, which not only handles failures but also ensures consistent outputs across transitions between synchronous mode and asynchronous mode.

3. k-Interactive Consistency

Classical consensus algorithms reach agreement on a single proposal (or batch of inputs) at a time, which is inefficient and limits scalability. If an RSM is implemented with this kind of consensus protocol, every state transition would require at least one distinct consensus cycle, or round of consensus, to commit. To achieve high scalability in a wide-area network (WAN), we introduce *k-interactive consistency* which aims to reach agreement on an ordered list of proposals from different processes within a single consensus cycle. A solution to this problem can be used to implement a scalable RSM in the WAN. The k-*IC* problem evolves from a combination of Interactive Consistency (IC) [40] and Vector Consensus (VC) [13]. Given n processes, f of which may be faulty, IC entails reaching agreement on at least $n - f$ values, and was proposed to tackle certain problems (synchronization of clocks and stabilization of input from sensors) in synchronous fault-tolerant systems where failures can be detected easily. VC proposes to reach agreement on a vector of values in an asynchronous setting and ensures there are at least $f + 1$ *non-null* proposal values in the vector that correspond to non-faulty processes. Thus, IC and VC always allow missing proposal values, even in some executions that are both synchronous and free of failures, such as when up to f processes can be Byzantine but all processes conform to the protocol long enough for decisions to be computed. To fit a broader range of synchrony and failure situations in one environment, we bound the number of missing proposals in k-*IC* using the tunable parameter k.

We now present the formal definition of k-*IC*. To simplify the description, we assume that each proposal is a single value that may represent a (possibly empty) batch of inputs in practice. Instead of agreeing on one such value, we modify the classical consensus problem so that a set of n processes, $P = \{P_1, ..., P_n\}$, agrees on an ordered list of n values, V, which tries to include the proposal value v_i of each process P_i but may also return *null*, indicating a missing value (e.g., due to a failure or asynchrony). The order of these values is predetermined, such as ordered numerically by the process ID, namely $V = [v_1, ..., v_n]$. We will refer to the ith element in this order as $V[i]$.

A solution to the k-*IC* problem in the Byzantine failure model[2] must satisfy the following properties:

[1] The degree of synchronization achieved determines how long it takes the collection of processes to return to the fast path, as we explain later on in the analysis of Theorem 5.5.

[2] A simpler problem specification for the crash failure model is presented in the conference version of this manuscript [33].

Validity: If a non-faulty process decides a *non-null* value $V[i]$ and process P_i is non-faulty, then $V[i]$ was proposed by process P_i.

Agreement: No pair of non-faulty processes ever decides different V.

k-Completeness: If a non-faulty process decides a list of values V then the number of *null* values in V is no greater than k, where $k \geq 0$.

Termination: Every non-faulty process eventually decides a list of values V.

The parameterized k-Completeness property rules out the trivial solution where the decision values are all *null*. Note that a system may satisfy k-Completeness with different k in different situations, including $k = 0$ under favorable conditions.

By rotating the proposers (e.g., in round-robin order of the processes), a classical consensus protocol can reach *k-IC* in multiple rounds of consensus by agreeing on one value per round as long as the proposers continue to participate in a timely manner. Otherwise, the solution becomes complex, and additional mechanisms have to be carried out to handle the failure or asynchrony of the proposers. Antipaxos instead reaches agreement on n values in parallel, even when these values represent conflict-prone transactions.

4. BFT Antipaxos protocol

In this section, we introduce the design of AP under the Byzantine failure model (APBFT). Suppose there are $n \geq 3f + 1$ processes, where at most f processes may be faulty, and we would like to reach *k-IC* for the smallest possible k. When the environment stays in synchronous mode long enough, processes perform an all-to-all exchange of their proposal values and 0-*IC* is achieved, meaning that each process contributes a *non-null* value to the output. Note that *empty* is a special *non-null* value, indicating that the client process participated but had no value to contribute (e.g., its batch of transactions was empty). On the other hand, if any process suffers a timeout while waiting for messages from others, due to a failure or asynchrony, then the slow path is triggered for one or more processes and *f-IC* is achieved. Moreover, all processes eventually return to the fast path if the environment returns to synchronous mode for long enough.

The protocol involves three phases: propose a value, broadcast a *liveness report* on the values received from others, and finally commit a list of values. Fig. 1 illustrates the principal flow of messages, with some details omitted for simplicity. The first two phases require each process to send a message to all other processes. In the absence of a timeout, the commit phase does not require any extra communication, and each process proceeds to commit the decision independently by analyzing the reports received in the second phase. This fast path realizes consensus on a collection of n values (i.e., 0-*IC*) using $\Theta(n^2)$ messages, and the latency is two message delays. If a failure or asynchrony causes a timeout, the affected processes seek assistance from a component called the Decision Module (DM), which is a layer of code implemented on top of a replicated state machine (RSM) that internally uses a conventional BFT consensus protocol, as the slow path in the commit phase. The RSM can be implemented, for example, using PBFT [8] or Mir-BFT [46,47]. Although some processes may proceed in the fast path while others fall back on the slow path, the reports exchanged in the second phase allow the DM to compute a decision that always agrees with the fast path.

Each of the n processes in the system executes three roles: the main AP protocol, the DM protocol, and a replica of the DM's RSM. Furthermore, each role entails the execution of multiple procedures, some invoked from a loop that counts consensus cycles and others driven by external events (i.e., triggered by receipt of specific messages). Each process interleaves the execution of all these procedures using a collection of threads. The AP protocol activates the DM by broadcasting a request for assistance to all processes. These requests are processed by the DM threads, which submit commands to the RSM by sending messages to the co-located RSM replica. The RSM only needs to process one valid command per consensus cycle, where validity is determined by verifying cryptographic signatures and checking structural properties (see Table 1), and the first valid command committed becomes the DM's decision. The co-located RSM replica returns this decision to the DM thread along with a quorum certificate that proves it was accepted by the RSM (at a particular position in the RSM's sequence of decisions), which can be shared with other processes and verified by them. The DM ensures that any two processes compute the same decision, even if one follows the fast path and the other follows the slow path.

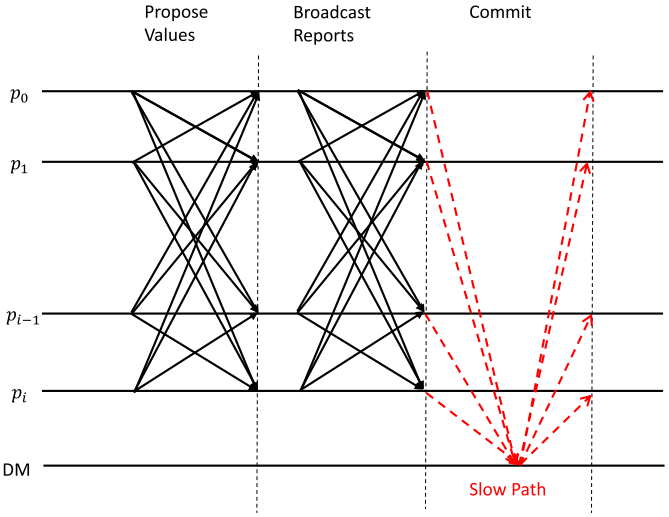


Fig. 1. Example of AP message flow.

Table 1
Validity criteria for different message types.

Message type	Sent	Valid message structure
proposal message from P_x	line 2	$\langle v, \sigma_x(v) \rangle$ where v is a non-*null* value
report message from P_x	line 15	$\langle x, v[1..n], sig[1..n], \sigma_x(v[1..n]) \rangle$ where each $v[i]$ is either a value or *null*, $sig[i] = \sigma_i(v[i])$ if $v[i] \neq null$, and $sig[i] = null$ if $v[i] = null$
assistance request	line 26	$\langle R \rangle$ where R is a set of valid report messages from at least $n - f$ distinct processes
final decision message	line 49	$\langle v[1..n], proof \rangle$ where each $v[i]$ is a value or *null*, $proof$ is a set of valid report messages from at least $n - f$ distinct processes, and each $v[i]$ satisfies the following: if there exists a value $w \neq null$ such that $\lvert\{rep \in proof \mid rep.v[i] = w\}\rvert \geq f + 1$ then $v[i]$ is one such a value; else if there exists a value $w \neq null$ such that $\lvert\{rep \in proof \mid rep.v[i] = w\}\rvert > 0$ then $v[i]$ is one such a value; else $v[i] = null$.

4.1. Detailed design

We present pseudocode for the prepare, report, and commit phases of the protocol in Algorithms 1 to 3, respectively. Aside from the assistance request (line 26), each message sent by a process P_x includes a signature σ_x computed using P_x's private key (see lines 2, 15), or a quorum certificate generated by the RSM (see line 49). Any invalid message (see Table 1 for validity criteria), including any message with an invalid signature or quorum certificate, is discarded by the recipient.

In the propose value phase (Algorithm 1), each process P_x broadcasts its signed proposal value v_x (lines 1 to 3). A faulty process may maliciously send distinct values to different recipients, and this behavior is mitigated later on in the report phase.

Algorithm 1: Proposal phase algorithm for process P_x.

```
1  Procedure AP_Propose (v_x)
2      send proposal message ⟨v_x, σ_x(v_x)⟩ to each process P_i
3  end
```

Algorithm 2: Report phase algorithm for process P_x.

```
4  Procedure AP_Report ()
5      wait for valid proposal messages from n − f processes
6      wait for valid proposal messages from up to f remaining processes until a
         timeout expires
7      for each process P_i do
8          if valid proposal message ⟨v_i, σ_i(v_i)⟩ received from P_i then
9              v[i] = v_i and sig[i] = σ_i(v_i)
10         else
11             v[i] = null and sig[i] = null
12         end
13     end
14     rep_x = ⟨x, v[1..n], sig[1..n], σ_x(v[1..n])⟩
15     send report message rep_x to each process P_i
16 end
```

Next, in the report phase (Algorithm 2), P_x collects proposal values (lines 5 to 6). At most one valid proposal message is accepted from each process. A *liveness report* structure rep_x is then generated comprising a value $rep_x.v[i]$ and signature $rep_x.sig[i]$ for each process P_i. If P_x did not receive a proposal message from P_i then a *null* value and signature are recorded (line 11). The list of values in the report structure is signed and the report message is broadcast to all processes (lines 14 to 15). To reduce the size of the report message, the values can be replaced with hashes, and any values that are missed in the proposal phase but appear later in the reports can be retrieved from other processes.

Algorithm 3: Commit phase algorithm for process P_x.

```
17 Procedure AP_Commit ()
18     wait for valid report messages from n − f processes
19     wait for valid report messages from up to f remaining processes until a
         timeout expires
20     if valid reports received from all n processes and they all indicate the same
         non-null value v_i for each process P_i then
21         for each process P_i do
22             V[i] = v_i
23         end
24     else
25         R = set of valid report messages received at lines 18 to 19
26         send assistance request message ⟨R⟩ to each process P_i
27         wait for valid final decision message dec from process P_x
28         for each process P_i do
29             V[i] = dec.v[i]
30         end
31     end
32     output V as the list of values decided
33 end
```

In the commit phase (Algorithm 3), P_x first collects reports (lines 18 to 19). At most one valid report is accepted from each process. P_x computes a list of values V from these reports according to two execution paths. If P_x received reports from all n processes, and the reports agree on the input value $v_i \neq null$ of each process P_i, then these n values are used directly to compute the output V in the fast path and attain 0-*IC* (lines 21 to 23). If not, then P_x enters the slow path (lines 25 to 30) where it first creates an *assistance request message* comprising the collection of liveness reports received in the report phase. The reports in this message contain much redundant data (i.e., mostly the same values and value signatures), and can be compressed effectively in practice. P_x then sends the assistance request to the DM (line 26), waits for a decision (line 27), and populates its list V with the values received from the DM (lines 28 to 30). As we show later on in Section 5, the fast path is eventually always followed if the environment is in synchronous mode for sufficiently long, and f-*IC* is always attained.

Algorithm 4: DM main procedure for process P_x.

```
34 Procedure DM_Main ()
35     wait for valid assistance request message ⟨R⟩ from some process
36     for each process P_i do
37         if R contains at least f + 1 reports that specify the same non-null value
             v_i for P_i then
38             final_dec.v[i] = v_i
39         else if R contains at least one report that specifies a non-null value v_i
             for P_i then
40             final_dec.v[i] = v_i
41         else
42             final_dec.v[i] = null
43         end
44     end
45     final_dec.proof = R
46     submit final_dec as a command to the RSM
47     wait for RSM to commit at least one valid final decision message
48     dec = first valid final decision message committed by the RSM
49     send dec with RSM's quorum certificate to all processes
50 end
```

The DM's algorithm (Algorithm 4) waits for an assistance request message (line 35) and analyzes it (lines 36–44) to compute a *final decision* structure $final\_dec$. Only one valid assistance request message is needed. For each process P_i, if $f + 1$ or more of the reports specify the same non-*null* value v_i for P_i then the final decision includes such a value (lines 37–38). Otherwise, if at least one report specifies a non-*null* value v_i for P_i then the final decision includes this value (lines 39–40). Finally, if all the reports specify the *null* value for P_i then the final decision includes the same (line 42). The set of reports obtained from the assistance request is included in the final decision message as a proof. Next, the computed final decision structure is submitted as a command to the RSM (line 46). The decision of the RSM is the first *valid* final decision message committed by the RSM (line 48). This decision is broadcast to all processes along with a quorum certificate obtained from the RSM (line 49). As an optimization, the RSM can internally discard all invalid commands, and commits at most one valid command.

A non-faulty process P_x that requests assistance must ensure that the decision it receives (line 27) is indeed the first valid final decision message committed by the RSM (line 48). The validity criterion of the decision message (Table 1) is checked easily, but P_x cannot verify in the commit phase that it received the *first* such message committed by the RSM. To compensate for this, P_x sends its request for assistance to all the processes (line 26), and then accepts a decision only from the DM procedure (Algorithm 4) running in the trusted DM thread of the same process P_x (line 27).

4.2. APBFT running examples

As a running example, consider the case where four processes P_1, P_2, P_3, and P_4 propose values v_1, v_2, v_3, and v_4, respectively ($n = 4$ and $f = 1$). We omit signatures in this example for readability.

Case 1: Suppose that P_1 receives values v_2, v_3 and v_4 from P_2, P_3 and P_4, respectively. Similarly P_2, P_3 and P_4 receive values from all other processes. If no process is faulty and no timeouts occur then all four processes produce the following report:

$\langle v[1] = v_1, v[2] = v_2, v[3] = v_3, v[4] = v_4 \rangle$

In this case all processes decide $V = \langle v_1, v_2, v_3, v_4 \rangle$ in the fast path without seeking assistance from the DM.

Case 2: P_2 crashes in the proposal phase and P_3 does not receive v_2. In this case P_1's and P_4's reports include v_2 but P_3's does not:

rep_1 : $\langle v[1] = v_1, v[2] = v_2, v[3] = v_3, v[4] = v_4 \rangle$

rep_2 : never generated

rep_3 : $\langle v[1] = v_1, v[2] = null, v[3] = v_3, v[4] = v_4 \rangle$

$rep_4 : \langle v[1] = v_1, v[2] = v_2, v[3] = v_3, v[4] = v_4 \rangle$

P_1, P_3, and P_4 all proceed to the slow path in the commit phase due to the *null* value and P_2's missing report. The DM receives reports from these three processes and merges them. The decision reached is $V = \langle v_1, v_2, v_3, v_4 \rangle$.

Case 3: P_2 crashes before sending its proposal value to any process. In this case P_1, P_3, and P_4 include *null* for P_2:

$rep_1 : \langle v[1] = v_1, v[2] = null, v[3] = v_3, v[4] = v_4 \rangle$
rep_2 : never generated
$rep_3 : \langle v[1] = v_1, v[2] = null, v[3] = v_3, v[4] = v_4 \rangle$
$rep_4 : \langle v[1] = v_1, v[2] = null, v[3] = v_3, v[4] = v_4 \rangle$

P_1, P_3, and P_4 all proceed to the slow path in the commit phase due to the *null* value and P_2's missing report. The DM receives reports from these three processes and merges them. Since all reports specify $v[2] = null$, the decision reached is $V = \langle v_1, null, v_3, v_4 \rangle$.

Case 4: P_2 is corrupted and sends distinct values, v_2', v_2'' and v_2''', to different processes. Divergent reports are generated and none contains a *null* value:

$rep_1 : \langle v[1] = v_1, v[2] = v_2', v[3] = v_2, v[4] = v_4 \rangle$
rep_2 : never generated
$rep_3 : \langle v[1] = v_1, v[2] = v_2'', v[3] = v_2, v[4] = v_4 \rangle$
$rep_4 : \langle v[1] = v_1, v[2] = v_3''', v[3] = v_2, v[4] = v_4 \rangle$

P_1, P_3, and P_4 all proceed to the slow path since their reports disagree. The DM receives reports from these three processes and merges them. The decision reached could be $V = \langle v_1, v_2', v_3, v_4 \rangle$, $V = \langle v_1, v_2'', v_3, v_4 \rangle$, or $V = \langle v_1, v_2''', v_3, v_4 \rangle$.

5. Proofs of correctness

This section proves the safety and liveness properties of the APBFT protocol. We consider a single cycle of consensus for analysis. Recall that a faulty process is a process that suffers a Byzantine failure in this cycle, and a non-faulty process is a process that either never suffered such a failure, or was repaired prior to the start of the current cycle.

Theorem 5.1 *(Validity). If a non-faulty process decides a non-*null *value $V[i]$ and process P_i is non-faulty, then $V[i]$ was proposed by process P_i.*

Proof. The non-*null* value $V[i]$ can only be decided in the fast path at line 22, or on the slow path at line 29. In the fast path, the value $V[i]$ is extracted from a report message *rep* received earlier at lines 18 to 19, and is accompanied in that report by P_i's signature (line 9, lines 14 to 15). Since the adversary cannot forge the signatures of non-faulty processes, and since P_i is non-faulty, *rep* is valid and $V[i]$ is the proposal value signed by P_i (line 2, lines 8 to 9), as required.

In the slow path, the decision is obtained from the DM at lines 27 to 30, and is derived from a valid final decision message dec committed by the DM's RSM (line 48). Specifically, $V[i] = dec.v[i]$. Since we assume that $V[i] \neq null$, the validity criterion (Table 1) for the final decision message dec implies that $dec.proof$ is a set containing at least one valid report structure *rep* such that $rep.v[i] = dec.v[i]$. Since each such *rep* is valid, $rep.sig[i] = \sigma_i(rep.v[i])$, and so $rep.v[i]$ is P_i's proposal value (line 2). This completes the proof since $rep.v[i] = dec.v[i] = V[i]$. □

Theorem 5.2 *(Agreement). If any two non-faulty processes determine a list of values, then both determine the same list of values.*

Proof. Suppose that non-faulty processes P_x and P_y decide on lists of values V_x and V_y, respectively. Each non-faulty process determines its list of values either in the fast path or the slow path. We must show that $V_x = V_y$ no matter which combination of paths is followed.

Case 1: P_x and P_y both decide in the fast path. Then both processes receive reports free of *null* values from all n processes, and moreover these reports agree on the value $v[i]$ of each process P_i (line 20). Since the reports of non-faulty processes cannot be forged, this implies that P_x's and P_y's reports have identical lists of values. Since the value lists output by P_x and P_y in the fast path are consistent with their own reports (lines 20 to 23), this implies that $V_x = V_y$.

Case 2: P_x and P_y both decide in the slow path. Then both processes receive their lists of values in a valid final decision message from the DM (line 27). Such a decision message cannot be forged since it includes the RSM's quorum certificate (line 49). Moreover, since any non-faulty process P_i deciding in the slow path receives this decision from the DM thread in the same process P_i, the decision is indeed the first valid final decision message committed by the DM's RSM (line 48). Thus, all decisions computed in the slow path by non-faulty processes have the same list of values, and this implies that $V_x = V_y$.

Case 3: P_x decides in the fast path and P_y decides in the slow path. Suppose for contradiction that $V_x[i] \neq V_y[i]$ for some $1 \leq i \leq n$.

Subcase 3a: $V_y[i] = null$. Then the decision message received by P_y includes *null* as the i'th value, and a proof comprising a set that includes a valid report *rep* from at least $2f + 1$ processes such that $rep.v[i] = null = V_y[i]$ (line 42, line 45). At least one such *rep* is from a non-faulty process P_z, and could not have been forged. Process P_z sent the same report, which contains at least one *null*, to P_x. This contradicts the assumption that P_x commits in the fast path due to the condition evaluated by P_x at line 20.

Subcase 3b: $V_y[i] \neq null$. Then the decision message received by P_y includes $V_y[i]$ as the ith value, and a proof comprising a set that includes a valid report *rep* from at least $2f + 1$ processes, where at least one such *rep* satisfies $rep.v[i] = V_y[i]$ (line 38, line 40, line 45). If there are $f + 1$ or more such structures with $v[i] = V_y[i]$, then at least one is from a non-faulty process P_z that sent the same report to P_x. Since $V_x[i] \neq V_y[i]$, P_z's report and P_x's decision differ, and this contradicts the assumption that P_x commits in the fast path due to the condition evaluated by P_x at line 20. Alternatively, there is at least one and fewer than $f + 1$ report structures in the proof such that $v[i] = V_y[i]$. In this case, since the decision message is valid, there is no subset of $f + 1$ or more reports *rep* in the proof that share a common non-*null* value of $rep.v[i]$. Moreover, since the proof contains at least $n - f = 2f + 1$ reports, there is a subset of such reports from at least $n - 2f = f + 1$ non-faulty processes that includes two reports with either *null* or unequal $v[i]$. Since both reports were sent to P_x, this once again contradicts the assumption that P_x commits in the fast path due to the condition evaluated by P_x at line 20.

Case 4: P_x decides in the slow path and P_y decides in the fast path. The analysis is analogous to Case 3. □

Theorem 5.3 *(k-Completeness). If a non-faulty process P_x outputs a list of values V, then the number of* null *values in V is 0 as long as no non-faulty process suffers a timeout (i.e., 0-IC is achieved in the best case), and f in the worst case (i.e., f-IC is always achieved).*

Proof. Suppose first that no non-faulty process suffers a timeout. If P_x commits in the fast path then it always outputs n non-*null* values (and zero *null* values) by the condition at line 20. On the other hand, if P_x commits in the slow path then the decision is made by the DM at lines 36 to 45, where it computes a final decision message based on valid reports from $n - f = 2f + 1$ or more processes, including $f + 1$ or more from non-faulty processes. Since no non-faulty process times out, each report from a non-faulty process contains exactly $n = 3f + 1$ non-*null* values, and it follows from the validity criterion for the final decision message (Table 1) that none of the values in the final decision are *null*. Since P_x is non-faulty, it checks the validity criterion at line 27 and outputs a list V comprising 0 *null* values.

Next, consider the general case. If P_x commits in the fast path, then it outputs n non-*null* values, as explained in the earlier special case. The slow path analysis is similar, except that each report from a non-faulty process now contains at least $n - f = 2f + 1$ non-*null* values by line 5,

and so it follows that P_x outputs a list V comprising at most f *null* values. □

Theorem 5.4 *(Termination). Suppose that the RSM of the DM maintains both safety and liveness for a sufficiently long period of time. Then eventually every non-faulty process decides a list of values V.*

Proof. Consider the execution of the prepare, report, and commit phases by a non-faulty process P_x. We proceed by analyzing each line of pseudocode, in order of execution, where P_x may potentially become stuck without reaching a decision.
Algorithm 1 (propose phase): P_x only sends a proposal message at line 2, and does not wait for any input.
Algorithm 2 (report phase): P_x may only wait at line 5 or line 6. At line 5, P_x eventually receives valid proposal messages from at least $n-f$ non-faulty processes, which are sent in the propose phase at line 2. At line 6, P_x either receives f additional proposals or times out.
Algorithm 3 (commit phase): P_x may only wait at line 18, line 19, or line 27. At line 18, P_x waits for $n-f$ valid report messages, which it eventually receives since every non-faulty process eventually computes a report at line 14 in the report phase, as explained earlier in the analysis of Algorithm 2. At line 19, P_x either receives f additional reports or times out. Finally, at line 27, P_x waits for a valid final decision message from the DM thread in the same process P_x. Now consider the execution of the DM main procedure in Algorithm 4, where P_x may wait at line 35 or line 47. At line 35, P_x waits for only one valid assistance request message, and eventually receives such a message from another process or from itself since P_x is non-faulty. Since we assume the RSM is safe and live, P_x eventually completes line 47, where it waits for the RSM to commit at least one valid final decision message after P_x itself computed such a message at lines 36–45. This completes the analysis of termination for the commit phase, including the DM's algorithm. □

Theorem 5.5 *(Eventual synchronicity). There exists a configuration of timeout values in AP such that all processes eventually run in the fast path continuously if the environment remains in synchronous mode for long enough.*

Proof. Suppose that the environment enters synchronous mode, and then consensus cycle c starts. Let D be the known upper bound on network and processing delays in synchronous mode. More concretely, suppose that D is sufficiently large for a process to receive messages from all processes, check any incoming message for validity (see Table 1), generate a new message (including computing any necessary cryptographic signatures), and send this message to all processes, and also for the network to deliver all the sent messages. We know from Theorem 5.4 that each non-faulty process eventually reaches a decision in cycle c, and since all processes are non-faulty in synchronous mode, it follows that every process eventually reaches a decision in cycle c. Furthermore, it follows that from the moment when the *last* process begins cycle c, the cycle is completed within a bounded amount of time. However, the total latency of cycle c, as measured from the moment when the *first* process begins cycle c, cannot be bounded so straightforwardly because the collection of processes may start cycle c at points scattered across an arbitrarily large time interval $[t, t+\Delta]$. At best, we can prove that the width of this interval, which we refer subsequently as the *spread* of processes with respect to some point in the execution, decreases over time.

Consider the effect of lines 5 to 6 and lines 18 to 19, where processes wait for delivery of messages broadcast one step earlier in the protocol (line 2 and line 15, respectively). Suppose that processes begin such a code segment within some interval $[t, t+\Delta]$. Then all the messages under consideration are sent by time $t+\Delta$, and all processes complete the code segment by time $t+\Delta+2D$ since the segment has two statements. On the other hand, since the last message is sent at time $t+\Delta-D$ or later, the earliest that a process can complete the code segment is $\max(t, t+\Delta-D)$ by receiving the message, or $t+T_{AP}$ by timing out, where T_{AP} is the timeout value. Thus, processes complete the code segment in the following time interval:

$$[\min(t+\Delta-D, t+T_{AP}),\ t+\Delta+2D] \quad (1)$$

Case 1: $\Delta < T_{AP} - D$. Then all the messages are delivered by time $t+\Delta+D < t+T_{AP}$, and so no timeout occurs. Furthermore, the width of the time interval in eq. (1) is at most $3D$, and so the spread is confined to at most $3D$.
Case 2: $\Delta \geq T_{AP} - D$. In this case one or more processes may time out, but the width of the time interval in eq. (1) is $\max(3D, \Delta+2D-T_{AP}) \leq \Delta+4D-T_{AP}$.

Now let the AP timeout be $T_{AP} = 4D+\epsilon$ for some $\epsilon > 0$. The case analysis shows that the code segments under consideration in the AP's algorithm either confine the spread among processes to at most $3D$ (Case 1) or else lessen it from Δ to at most $\Delta-\epsilon$ (Case 2). Choosing a large enough ϵ ensures that the spread among processes with respect to the beginning of a cycle shrinks over consecutive cycles, despite differences in execution speed between the fast path and slow path, until the spread is at most $3D$ and the protocol converges onto Case 1 where no timeouts occur. From that cycle onward, all processes decide exclusively in the fast path, as required. □

6. Evaluation

We evaluate our prototype on Amazon EC2 using c4.xlarge virtual machines (4 vCPUs, 7.5 GiB main memory) running Amazon Linux 2023. We compare BFT Antipaxos (APBFT) against PBFT [8] – a widely-studied classic leader-based protocol – and Mir-BFT [46] – a modern multi-leader protocol. Mir-BFT was chosen primarily because of its high efficiency and scalability relative to both PBFT and state-of-the art protocols such as Honeybadger [36] and HotStuff [51]. All three systems are implemented in Golang as replicated state machines, and the third party systems are both obtained from the research branch of the Mir-BFT codebase [50]. We consider both single data center (in the us-east-2 region) and geo-distributed (across the seven regions shown in Table 2) deployments. The main experiments (Fig. 2) are conducted at a scale of 63 hosts: 56 servers and 7 dedicated clients. The geo-distributed configuration distributes servers and clients uniformly across AWS regions.

The open-loop multi-threaded clients each connect to multiple servers and generate 500-byte requests, which is approximately the average size of a Bitcoin transaction. The execution latency of each request is the time from when the client sends the request to when a reply is received, confirming that the request was committed (i.e., a consensus cycle was completed for a batch that includes the request). APBFT and Mir-BFT servers batch transactions up to 2 MB or 200 ms. APBFT further supports pipelining, which parallelizes the execution of consecutive consensus cycles to improve network and CPU utilization. Each process is uniquely identified by an ECDSA [23] key pair for signing proposal values (i.e., the requests) and report messages. As suggested earlier in Section 4.1, the report messages in APBFT are replaced with SHA256 [35] hashes, and the Decision Module's state machine is implemented using PBFT [8]. The latter is not accessed at all in failure-free experiments.

6.1. Scalability experiments

The main scalability experiment is presented in Fig. 2. Each point shows the average throughput and latency computed over a sample of three 90-second failure-free runs. Error bars in both dimensions indicate one standard deviation of the sample. Different points are generated by tuning the request rate of the client until throughput stops increasing. Separate data series are plotted for mean latency and 99% latency.

The first experiment is conducted in the us-east-2 (Ohio) data center. We see that APBFT achieves roughly 5× higher peak throughput

Table 2
Round-trip network latency (ms).

	US-east-2	SA-east-1	EU-central-1	AP-north-1	US-west-2	AP-south-1
SA-east-1	123	-				
EU-central-1	100	205	-			
AP-north-1	146	270	260	-		
US-west-2	49	173	141	96	-	
AP-south-1	196	302	126	130	222	-
CA-central-1	23	123	89	156	59	189

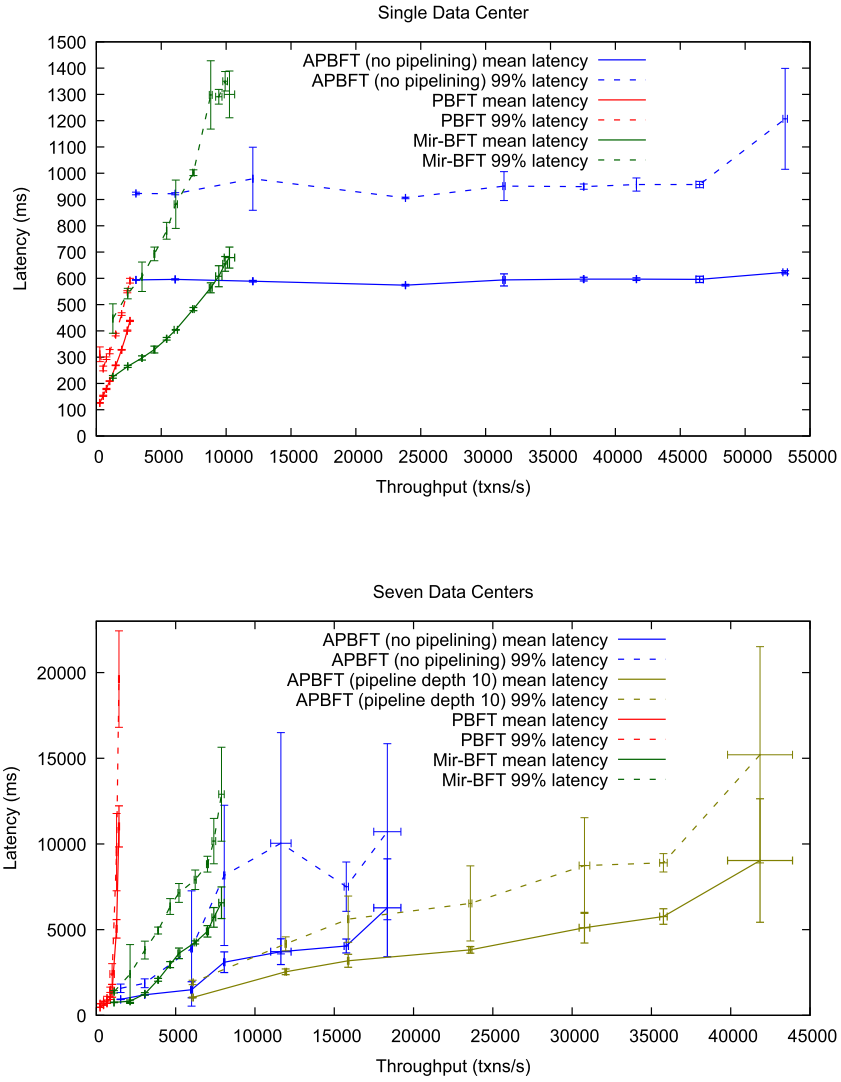


Fig. 2. Scalability comparison in the Amazon Elastic Compute Cloud (EC2).

than Mir-BFT. The mean latency of APBFT is also more steady, though generally higher, which is expected since the APBFT fast path requires communication among all pairs of servers whereas Mir-BFT uses quorums.

The second experiment is deployed in multiple Amazon regions: sa-east-1 (São Paulo), eu-central-1 (Frankfurt), ap-northeast-1 (Tokyo), ap-south-1 (Mumbai), us-east-2 (Ohio), us-west-2 (Oregon), and ca-central-1 (central Canada). The round trip network latency between these datacenters, measured using ping, is presented in Table 2.[3] APBFT achieves roughly 2× higher peak throughput than Mir-BFT and similar though more variable latency. A further 2.5× gain in throughput is achieved by pipelining APBFT consensus cycles with a pipeline depth of 10. Overall, APBFT maintains a single-digit mean latency across a range of throughput values, which is suitable for applications such as online payment processing.

6.2. Fault tolerance

The final experiment examines how APBFT responds to failures. We deploy 4 APBFT processes and 1 client in the same datacenter in the us-east region. We use an open-loop client to send request at a constant speed – 10,000 requests per second – to a non-faulty APBFT process.

[3] Ping times vary from day to day. The measured values are accurate to ±10 ms.

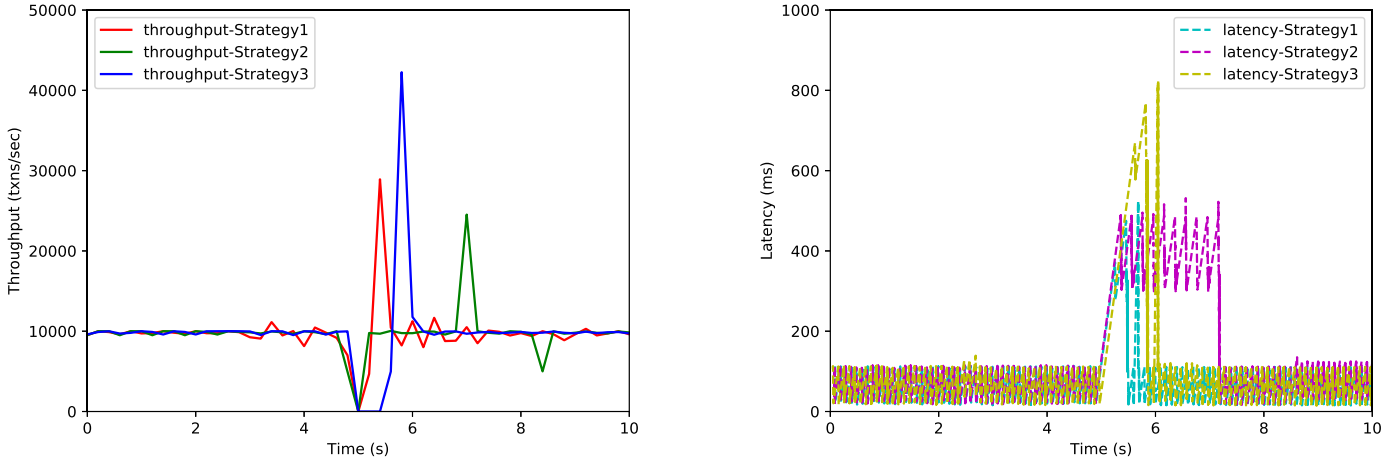


Fig. 3. APBFT fault recovery experiment, one replica is killed after 5 s into the experiment.

We measure the latency of each request and calculate the throughput every 0.2 s. We simulate a replica crash by killing a replica roughly 5 s after the experiment starts. Three different strategies are applied to handle the failure. Strategy 1 (S1) excludes the failed replica from the computation if the DM detects a single failure in any consensus cycle. The timeout of each step is set to 100 ms. Strategy 2 (S2) excludes the failed replica if a failure occurs for 20 consecutive cycles. The timeout is 100 ms, as in S1. Strategy 3 (S3) excludes the failed replica immediately if a failure is detected, as in S1, but the timeout is set to a larger value, 200 ms.

The results are presented in Fig. 3. There is a fluctuation in all three strategies when the failure occurs. The throughput decreases and the latency increases at about 5 s, but the system maintains liveness. This throughput and latency fluctuation are due to the mechanism of the DM. When an AP process detects a failure, it has to report to the DM and waits for a decision; the requests are committed only after the decision is received. S1 takes the least time to recover to normal as it excludes the failed replica from the computation immediately when the failure is detected. S2 takes a longer time to recover due to multiple assistance cycles. S3 performs better than S2 but worse than S1 due to a longer timeout. For practical applications, a good approach to ensure that AP runs in the fast path in most cycles is to measure and estimate the distribution of message and processing delay, and tune the timeout according to the tail of the distribution. If a timeout occurs, either due to process failure or asynchrony, the system can exclude the suspected process from the computation for one or more cycles, as in S1 and S2, and repeat offenders can be manually removed from the system for repairs. A process that legitimately fails must copy missed data from others on recovery, and determine the current consensus cycle to resume participation.

7. Related work

Paxos [28] is a widely-studied crash-tolerant consensus protocol. A process is selected as the proposer to lead the consensus protocol, and communicates with a majority quorum of acceptors. Acceptors are the fault-tolerant memory of the protocol, and learners are the processes that execute the requests. A process can take any or all the roles in the protocol. At the beginning of the protocol, a leader sends a message to all the acceptors in the first phase and waits for a quorum of replies. Next, the leader sends an accept message to all the acceptors, which includes the proposed value, in phase two. In the absence of competing proposals, the acceptors accept the proposed value and send a reply to both the leader and the learners. Paxos always guarantee safety, but liveness is conditioned on the existence of a stable leader. It requires $2f + 1$ acceptors to tolerate f crash failures, and does not solve the leader election problem.

Due to the reliance on a single proposer, the performance of Paxos is limited. Many variants of Paxos (see [42]) have been proposed to improve scalability. For example, Fast Paxos [29] reduces the agreement time by having proposers bypass the leader and send directly to a larger quorum acceptors. Mencius [34] improves throughput by rotating the leaders across different consensus cycles. Outside the Paxos family of protocols, Raft [39] has similar fault-tolerance and performance characteristics, but is designed to be more understandable than Paxos. ZAB [24] is an atomic broadcast protocol employed by Zookeeper [21] that follows similar principles.

EPaxos [37] is an improvement on Paxos where every replica handles requests from the clients. When a replica receives a request from a client, the replica becomes the request leader and forwards the request to at least a fast path quorum of replicas. If the request leader receives enough replies where all the replies have the same attributes, which constrain the commit order, then the request can be committed in the fast path. Otherwise, the request has to commit through the slow path, which is based on Paxos. If the request leader commits a request, it simply broadcasts this request to everyone else to execute. In the fast path, EPaxos only requires one round-trip communication latency to commit an non-interfering request. EPaxos also employs batching techniques to increase throughput. However, each EPaxos replica runs consensus one cycle at a time, which limits its throughput in WANs. Furthermore, concurrent conflicting commands, which are common when commands are batched, always fall to the slow path. Compared to EPaxos, Altas [15] improves performance by using a minimized quorum for its fast path. Canopus [43] is a consensus protocol with a hierarchical tree structure. The top layer of this hierarchy uses an efficient all-to-all broadcast, similarly to Antipaxos, which allows Canopus to outperform EPaxos in terms of throughput for read-heavy workloads. However, Canopus lacks the ability to tolerate a network partition or the failure of an entire subtree.

Moving on beyond crash fault-tolerant consensus protocols, much attention has been paid to tolerating Byzantine failures [30]. PBFT [8] is the most widely used single-leader Byzantine fault-tolerant consensus protocol. FastBFT [32] proposes a novel message aggregation technique based on hardware-based trusted execution environments to reduce message complexity. Gosig [31] supports multiple proposers, but only one of the proposals can be selected in each cycle. There are a number of other single-leader BFT protocols such as SBFT [19], HotStuff [51], Zyzzyva [26], Aliph [18], etc.

With regard to leaderless and multi-leader approaches, set-union consensus [12] proposes that a set of processes reach agreement on a set of values instead of a single value in a consensus cycle, which is similar to *k-IC*. Set-union consensus requires that there is no conflict or invalid element in the decision set. The definition of a conflict or invalid is application-specific. There are many other variants of the leaderless

approach to consensus, including Honey Bager [36], DBFT [10], and Red Belly [11].

Mir-BFT [46] is a recent multi-leader approach consensus protocol where each leader runs an instance of PBFT. It splits the consensus into epochs. In each epoch, each leader is assigned a fixed number of buckets. Requests are matched to the buckets using its hash, and buckets are rotated across different epochs. Finally, requests are interleaved in an unique sequence according to the order of the buckets. Mir-BFT can emulate PBFT by setting only one leader and making every epoch stable. However, there is no mechanism for a slow leader to catch up. As a result, a slow leader limits performance. In addition, Mir-BFT requires $O(n^3)$ messages if all the nodes are leaders.

RCC [20] is a transformation that enables execution of multiple instances of consensus in parallel. This allows different replicas to serve as leaders concurrently for different instances of consensus, and improves resource utilization as compared to the rotating coordinator scheme [9,34]. RCC relies on a checkpointing mechanism to ensure that a faulty leader cannot keep f non-faulty replicas "in the dark," and uses a separate instance of consensus to coordinate agreement after a failure. The latter is conceptually similar to the Decision Module in AP, though our approach is fundamentally different as it replaces the primary layer of consensus with a simple all-to-all broadcast. The best-case latency of RCC depends on the choice of consensus algorithm for this layer, whereas AP always achieves the optimal latency of two message delays in the fast path.

Steward [1] and RCanopus [25] are Byzantine consensus protocols that improve performance by using a hierarchical architecture. They comprise multiple sites, each using Byzantine consensus internally to reach local decisions and certify them. Agreement across sites is achieved using a Paxos-like single-leader consensus protocol in Steward. In contrast, RCanopus uses an all-to-all broadcast to synchronize sites, similarly to Canopus [43]. Fault tolerance is limited, as in Canopus, since the protocol stalls when a site becomes unavailable. AP is an elaboration of the convergence module (CM) in RCanopus (Appendix A of [25]), which was introduced to make RCanopus resilient against network partitions and site failures. In the original design, which predates RCC [20] and DQBFT [3] by several years, the CM was incompletely specified and lacked rigorous analysis.

The use of a trusted hardware has been proposed to improve the efficiency of Byzantine consensus in several works [3,5,48]. A trusted execution environment, such as Intel SGX, can prevent equivocation and reduce the number of replicas required to tolerate f failures from $3f+1$ to $2f+1$. DQBFT [3] uses multiple instances of consensus concurrently, similarly to RCC, and an additional instance of consensus for global ordering. In contrast, AP does not use consensus at all in the fast path.

8. Conclusion

In this paper, we first formalized the *k-IC* problem for the Byzantine failure model. Then we proposed Antipaxos, a scalable leaderless protocol, to solve this problem. In the absence of failures and asynchrony, the protocol bypasses classical consensus protocols and reaches agreement efficiently using simple all-to-all broadcast. Otherwise, the algorithm still maintains safety in all executions, and liveness under sufficient synchrony, with the help of the Decision Module. In addition, proposal batching and pipelining allow Antipaxos to mitigate the impact of asynchrony and achieve high scalability in WANs. We implemented a system prototype of Antipaxos and compared its scalability in the Amazon cloud against the highly parallel multi-leader Mir-BFT [46] protocol. Our experiments show that Antipaxos achieves several-fold higher peak throughput.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Chunyu Mao and Wojciech Golab report financial support was provided by Ripple Labs. Bernard Wong reports financial support was provided by Huawei Technologies Canada.

Acknowledgments

This research was supported in part by Ripple Labs, Huawei Technologies Canada, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada. We are grateful to the anonymous reviewers of this manuscript and its shorter conference version [33] for their insightful feedback and helpful suggestions.

References

[1] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, D. Zage, Steward: scaling Byzantine fault-tolerant replication to wide area networks, IEEE Trans. Dependable Secure Comput. 7 (1) (2010) 80–93.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A.D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al., Hyperledger fabric: a distributed operating system for permissioned blockchains, in: Proceedings of European Conference on Computer Systems (EuroSys), 2018, pp. 1–15.

[3] B. Arun, B. Ravindran, Scalable Byzantine fault tolerance via partial decentralization, Proc. VLDB Endow. 15 (9) (2022) 1739–1752.

[4] J. Aspnes, Randomized protocols for asynchronous consensus, Distrib. Comput. 16 (2–3) (2003) 165–175.

[5] J. Behl, T. Distler, R. Kapitza, Hybrids on steroids: SGX-based high performance BFT, in: Proceedings of the 12th European Conference on Computer Systems (EuroSys), ACM, 2017, pp. 222–237.

[6] M. Ben-Or, Another advantage of free choice (extended abstract): completely asynchronous agreement protocols, in: Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 1983, pp. 27–30.

[7] G. Bracha, S. Toueg, Asynchronous consensus and broadcast protocols, J. ACM 32 (4) (1985) 824–840.

[8] M. Castro, B. Liskov, Practical Byzantine fault tolerance, in: Proceedings of Symposium on Operating Systems Design and Implementation (OSDI), 1999, pp. 173–186.

[9] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (1996) 225–267.

[10] T. Crain, V. Gramoli, M. Larrea, M. Raynal, DBFT: efficient leaderless Byzantine consensus and its application to blockchains, in: Proceedings of IEEE International Symposium on Network Computing and Applications (NCA), 2018, pp. 1–8.

[11] T. Crain, C. Natoli, V. Gramoli, Red Belly: a secure, fair and scalable open blockchain, in: Proceedings of IEEE Symposium on Security and Privacy (S&P), 2021, pp. 466–483.

[12] F. Dold, C. Grothoff, Byzantine set-union consensus using efficient set reconciliation, in: Proceedings of International Conference on Availability, Reliability and Security (ARES), 2016, pp. 29–38.

[13] A. Doudou, A. Schiper, Muteness detectors for consensus with Byzantine processes, in: Proceedings of Annual ACM Symposium on Principles of Distributed Computing (PODC), 1998, p. 315.

[14] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.

[15] V. Enes, C. Baquero, T.F. Rezende, A. Gotsman, M. Perrin, P. Sutra, State-machine replication for planet-scale systems, in: Proceedings of European Conference on Computer Systems (EuroSys), 2020, pp. 1–15.

[16] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.

[17] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: scaling Byzantine agreements for cryptocurrencies, in: Proceedings of Symposium on Operating Systems Principles (SOSP), 2017, pp. 51–68.

[18] R. Guerraoui, N. Knežević, V. Quéma, M. Vukolić, The next 700 BFT protocols, in: Proceedings of European Conference on Computer Systems (EuroSys), 2010, pp. 363–376.

[19] G.G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M.K. Reiter, D.-A. Seredinschi, A.T. Orr Tamir, SBFT: a scalable and decentralized trust infrastructure, in: Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), 2019, pp. 568–580.

[20] S. Gupta, J. Hellings, M. Sadoghi, RCC: resilient concurrent consensus for high-throughput secure transaction processing, in: Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE), IEEE, 2021, pp. 1392–1403.

[21] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, ZooKeeper: wait-free coordination for Internet-scale systems, in: Proceedings of USENIX Annual Technical Conference (ATC), 2010, p. 11.

[22] Z. István, D. Sidler, G. Alonso, M. Vukolic, Consensus in a box: inexpensive coordination in hardware, in: Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2016, pp. 425–438.

[23] D. Johnson, A. Menezes, S. Vanstone, The elliptic curve digital signature algorithm (ECDSA), Int. J. Inf. Secur. 1 (1) (2001) 36–63.

[24] F.P. Junqueira, B.C. Reed, M. Serafini, ZAB: high-performance broadcast for primary-backup systems, in: Proceedings of International Conference on Dependable Systems & Networks (DSN), 2011, pp. 245–256.
[25] S. Keshav, W. Golab, B. Wong, S. Rizvi, S. Gorbunov, RCanopus: making Canopus resilient to failures and Byzantine faults, arXiv:1810.09300 [cs.DC], 2018.
[26] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, Zyzzyva: speculative Byzantine fault tolerance, in: Proceedings of ACM SIGOPS Symposium on Operating Systems Principles, 2007, pp. 45–58.
[27] A.D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, 2011.
[28] L. Lamport, Paxos made simple, ACM SIGACT News 32 (4) (2001) 18–25.
[29] L. Lamport, Fast Paxos, Distrib. Comput. 19 (2) (2006) 79–103.
[30] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, J. ACM 30 (3) (1983) 668–676.
[31] P. Li, G. Wang, X. Chen, F. Long, W. Xu, Gosig: a scalable and high-performance Byzantine consensus for consortium blockchains, in: Proceedings of ACM Symposium on Cloud Computing (SOCC), 2020, pp. 223–237.
[32] J. Liu, W. Li, G.O. Karame, N. Asokan, Scalable Byzantine consensus via hardware-assisted secret sharing, IEEE Trans. Comput. 68 (1) (2018) 139–151.
[33] C. Mao, W. Golab, B. Wong, Antipaxos: taking interactive consistency to the next level, in: Proceedings of the 23rd International Conference on Distributed Computing and Networking (ICDCN), 2022, pp. 128–137.
[34] Y. Mao, F.P. Junqueira, K. Marzullo, Mencius: building efficient replicated state machine for WANs, in: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association, 2008, pp. 369–384.
[35] A.J. Menezes, S.A. Vanstone, P.C.V. Oorschot, Handbook of Applied Cryptography, 1st edition, CRC Press, 1996.
[36] A. Miller, Y. Xia, K. Croman, E. Shi, D. Song, The honey badger of BFT protocols, in: Proceedings of ACM Conference on Computer and Communications Security (SIGSAC), 2016, pp. 31–42.
[37] I. Moraru, D.G. Andersen, M. Kaminsky, There is more consensus in egalitarian parliaments, in: Proceedings of ACM Symposium on Operating Systems Principles (SOSP), 2013, pp. 358–372.
[38] A. Mostefaoui, H. Moumen, M. Raynal, Signature-free asynchronous Byzantine consensus with t< n/3 and o (n2) messages, in: Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 2014, pp. 2–9.
[39] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: Proceedings of USENIX Annual Technical Conference (ATC), 2014, pp. 305–319.
[40] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.
[41] M. Poke, T. Hoefler, DARE: high-performance state machine replication on RDMA networks, in: Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2015, pp. 107–118.
[42] R.V. Renesse, D. Altinbuken, Paxos made moderately complex, ACM Comput. Surv. 47 (3) (2015) 1–36.
[43] S. Rizvi, B. Wong, S. Keshav, Canopus: a scalable and massively parallel consensus protocol, in: Proceedings of ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT), 2017, pp. 426–438.
[44] N. Santos, A. Schiper, Optimizing Paxos with batching and pipelining, Theor. Comput. Sci. 496 (2013) 170–183.
[45] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: a tutorial, ACM Comput. Surv. 22 (4) (1990) 299–319.
[46] C. Stathakopoulou, T. David, M. Pavlovic, M. Vukolić, [Solution] Mir-BFT: scalable and robust BFT for decentralized networks, J. Syst. Res. 2 (1) (2022).
[47] C. Stathakopoulou, M. Pavlovic, M. Vukolić, State machine replication scalability made simple, in: Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys), 2022, pp. 17–33.
[48] G.S. Veronese, M. Correia, A.N. Bessani, L.C. Lung, P. Veríssimo, Efficient Byzantine fault-tolerance, IEEE Trans. Comput. 62 (1) (2013) 16–30.
[49] C. Wang, J. Jiang, X. Chen, N. Yi, H. Cui, APUS: fast and scalable Paxos on RDMA, in: Proceedings of ACM Symposium on Cloud Computing (SOCC), 2017, pp. 94–107.
[50] J. Yellick, M. Pavlovic, C. Stathakopoulou, M. Vukolić, MirBFT library, https://github.com/hyperledger-labs/mirbft/tree/research, 2019.
[51] M. Yin, D. Malkhi, M.K. Reiter, G.G. Gueta, I. Abraham, HotStuff: BFT consensus with linearity and responsiveness, in: Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 2019, pp. 347–356.

Chunyu Mao received his Ph.D. degree in Electrical and Computer Engineering from the University of Waterloo in 2023. He is a licensed professional engineer (P.Eng). Dr. Mao is broadly interested in blockchain technology and enthusiastic about Web3 research and development. He currently serves as a Software Engineer at Capital One Canada.

Wojciech Golab received his Ph.D. degree in Computer Science from the University of Toronto in 2010. He spent two years at Hewlett-Packard Labs in Palo Alto as a Research Scientist before joining the University of Waterloo in 2012 as a faculty member in the Department of Electrical and Computer Engineering. Dr. Golab is broadly interested in concurrency and fault tolerance in distributed systems, with a special focus on bridging the gap between theory and practice.

Bernard Wong is an Associate Professor in the Cheriton School of Computer Science at the University of Waterloo. His research interests span distributed systems and networking, with particular emphasis on problems involving decentralized services, self-organizing networks, and distributed storage systems.