EZBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication

Balaji Arun, Sebastiano Peluso, Binoy Ravindran
ECE, Virginia Tech
Blacksburg, USA
{*balajia,peluso,binoy*}*@vt.edu*

Abstract—We present EZBFT, a novel leaderless, distributed consensus protocol capable of tolerating byzantine faults. EZBFT's main goal is to minimize the client-side latency in WAN deployments. It achieves this by (i) having no designated primary replica, and instead, enabling every replica to order the requests that it receives from clients; (ii) using only three communication steps to order requests in the common case; and (iii) involving clients actively in the consensus process. In addition, EZBFT minimizes the potentially negative effect of a byzantine replica on the overall system performance. We developed EZBFT's formal specification in TLA+, show that it provides the classic properties of BFT protocols including consistency, stability, and liveness, and developed an implementation. Our experimental evaluation reveals that EZBFT improves client-side latency by as much as 40% over state-of-the-art byzantine fault-tolerant protocols including PBFT, FaB, and Zyzzyva.

Keywords-Fault Tolerance, Consensus, Byzantine Fault Tolerance, State Machine Replication

I. INTRODUCTION

State Machine Replication (SMR) is a common technique employed in today's distributed applications to tolerate server failures and maintain high availability [1], [2]. The replication servers, or *replicas*, employ consensus protocols (e.g., Raft [3], Paxos [4], Zab [5]) to replicate the application state and ensure that the same sequence of client commands is executed on the shared state in the same order (i.e., a total order), ensuring consistency of the replicated state.

Consensus solutions can be broadly classified as Crash Fault-Tolerant (CFT) and Byzantine Fault-Tolerant (BFT), with the former being a subset of the latter. While CFT protocols have found their niche in datacenter applications [6], [7], [8] of a single enterprise-class organization, BFT protocols are increasingly used in applications involving multiple independent organizations. For example, for distributed *ledger* or blockchain-based applications in the *permissioned* setting, consensus need to be reached among a set of known participants (e.g., independent organizations), despite no complete trust among them. In such cases, BFT protocols can ensure replica state consistency while withstanding malicious actors and other non-crash related faults. An example of a *permissioned* blockchain system that uses a BFT protocol is the Hyperledger Fabric [9], [10].

PBFT [11] is, arguably, the first practical implementation of BFT consensus. Since its advent, a number of BFT protocols have been invented. Some reduce client-side latency [12], [13], [14], [15]; some optimize for higher server-side throughput [16], [1]; some have greater tolerance to faults [17]; and some reduce design and implementation complexity [18].

The use of permissioned blockchain applications is growing in many domains (e.g., supply chain [19], real-estate [20], finance [21]) and many such systems are increasingly deployed in geographically distributed settings to cater to different application needs. Geographical-scale (or "geo-scale") deployments of BFT systems have an additional challenge: achieving low client-side latencies and high server-side throughput under the high communication latencies of a WAN. Since replicas need to communicate with each other and the clients to reach consensus, the number of communication steps incurred directly impacts the latency, as each step involves sending messages to potentially distant nodes. Thus, protocols such as Zyzzyva [14], Q/U [13], and HQ [15] use various techniques to reduce communication steps. These techniques, however, do not reduce client-side latencies in a geo-scale setting, where, the latency per communication step is as important as the number of communication steps. In other words, a protocol can achieve significant cost savings if the latency incurred during a communication step can be reduced.

The downside of such lack of locality is most manifest for primary-based BFT protocols such as PBFT and Zyzzyva: a replica is bestowed the *primary* status and is responsible for proposing the total-order for all client requests. While the clients that are geographically nearest to the primary may observe optimal client-side latency, the same is not true for distant clients. A distant client will incur higher latency for the first communication step (of sending the request to the primary). Additionally, the primary-based design limits throughput as the primary carries significantly higher load.

To validate these hypotheses, we deployed Zyzzyva [14] in a 4-replica geo-scale setting with nodes located in the US (Virginia), Japan, India, and Australia, using Amazon's EC2 infrastructure [22]. We deployed clients alongside each replica to inject requests, and measured the client-side latencies by changing the primary's location. Table I shows the results. We observe that the lowest latencies are when the primary is co-located within the same region.

Past work suggests that it is important to perform regular

Table I: Zyzzyva's [14] latencies (in ms) in a geo-scale deployment with primary at different locations. Columns indicate the primary's location. Rows indicate average client-side latency for commands issued from that region. For example, the entry at the 4th row and the 3rd column shows the client-side latency for commands issued from India to the primary in Japan. Lowest latency per primary location is highlighted.

	Virginia (US)	Japan	India	Australia
Virginia (US)	**198**	238	306	303
Japan	236	**167**	239	246
India	304	242	**229**	305
Australia	303	232	304	**229**

view changes (i.e., elect a new primary) to reduce the negative impacts of byzantine replicas [23]. For geo-scale applications, this technique will induce suboptimal latency SLAs, because the appropriate latency SLA, under this technique, is the maximum of client-side latencies obtained by placing the primary at each and every site.

A leaderless protocol can solve the aforementioned problems. A client can send its requests to the nearest replica and can continue to do so as long as the replica is *correct*. The replica can undertake the task of finding an order among all the concurrent requests in the system, executing the request on the shared state, and return the result. Leaderless protocols [24], [25], [26] have been previously proposed for the CFT model. However, to the best of our knowledge, such investigations have not been made for the BFT model.

The transition from CFT to BFT is not straightforward and will yield a sub-optimal solution. As shown in [27], additional communication steps as well as more number of messages within each step are fundamental for such transformations. This will result in sub-optimal server-side throughput since more messages should be certified by replicas, (exponentially) increasing the degree of computation. Moreover, additional communication steps will increase client-side latency.

Motivated by these concerns, we present EZBFT, a leaderless BFT protocol. EZBFT enables every replica in the system to process the requests received from the clients. Doing so (i) significantly reduces the client-side latency, (ii) distributes the load across replicas, and (iii) tolerates faults more effectively. Importantly, EZBFT delivers requests in *three* communication steps in normal operating conditions.

To enable leaderless operation, EZBFT exploits a particular characteristic of client commands: *interference*. In the absence of concurrent interfering commands, EZBFT's clients receive a reply in an optimal three communication steps. When commands interfere, both clients and replicas coherently communicate to establish a consistent total-order, consuming an additional zero or two communication steps. EZBFT employs additional techniques such as client-side

validation of replica messages and speculative execution to reduce communication steps in the common case, unlike CFT solutions [24], [25].

We developed EZBFT's formal specification in TLA+ (available in an accompanying technical report [28]) and show that it provides the classic properties of BFT protocols including consistency, stability, and liveness.

To understand how EZBFT fares against state-of-the-art BFT protocols, we implemented EZBFT and conducted an experimental evaluation using the AWS EC2 infrastructure, deploying the implementations in different sets of geographical regions. Our evaluation reveals that EZBFT improves client-side latency by as much as 40% over PBFT, FaB, and Zyzzyva.

The paper's central contribution is the EZBFT protocol. To the best of our knowledge, EZBFT is the first BFT protocol to provide decentralized, deterministic consensus in the eventually synchronous model. By minimizing the latency at each communication step, EZBFT provides a highly effective BFT solution for geo-scale deployments.

The rest of the paper is organized as follows: Section II presents EZBFT's assumptions. Section III overviews EZBFT, and Section IV presents a complete algorithmic design and correctness arguments. An experimental evaluation of EZBFT is presented in Section V. We summarize past and related work in Section VI and conclude in Section VII.

II. SYSTEM MODEL

We consider a set of nodes (replicas and clients), in an asynchronous system, that communicate via message passing. The replica nodes have identifiers in the set $\{R_0, ..., R_{N-1}\}$. We assume the byzantine fault model in which nodes can behave arbitrarily. We also assume a strong adversary model in which faulty nodes can coordinate to take down the entire system. Every node, however, is capable of producing cryptographic signatures [29] that faulty nodes cannot break. A message m signed using R_i's private key is denoted as $\langle m \rangle_{R_i}$. The network is fully connected and quasi-reliable: if nodes R_1 and R_2 are correct, then p_2 receives a message from R_1 exactly as many times R_1 sends it.

To preserve safety and liveness, EZBFT requires at least $N = 3f + 1$ replica nodes in order to tolerate f Byzantine faults. EZBFT uses two kinds of quorums: a fast quorum with $3f + 1$ replicas, and a slow quorum with $2f + 1$ replicas. Safety is guaranteed as long as only up to f replicas fail. Liveness is guaranteed during periods of synchronous communication. Furthermore, there is no assumption on the number of faulty clients.

III. PROTOCOL OVERVIEW

We now overview EZBFT and highlight the novelties that enable it to tolerate byzantine failures and provide optimal wide-area latency.

EZBFT can deliver decisions in three communication steps from the client's point-of-view, if there is no contention, no byzantine failures, and synchronous communication between replicas. The three communication steps include: (i) a client sending a request to any one of the replicas (closest preferably); (ii) a replica forwarding the request to other replicas with a proposed order; and (iii) other replicas (speculatively) executing the request as per the proposed order and replying back to the client. These three steps constitute EZBFT's core novelty. To realize these steps, EZBFT incorporates a set of techniques that we summarize below and explain in detail in the next section.

First, the EZBFT replicas perform speculative execution of the commands after receiving the proposal messages from their respective command-leaders (see below). With only one replica-to-replica communication, there is no way to guarantee the final commit order for client commands. Thus, the replica receiving the proposal assumes that other replicas received the same proposal (i.e., the command-leader is not byzantine) and that they have agreed to the proposal. With this assumption, replicas speculatively execute the commands on their local state and return a reply back to the client.

Second, in EZBFT, the client is actively involved in the consensus process. It is responsible for collecting messages from the replicas and ensuring that they have committed to a single order before delivering the reply. The client also enforces a final order, if the replicas are found to deviate.

Third, and most importantly, there are *no* designated primary replicas. Every replica can receive client requests and propose an order for them. To clearly distinguish the replica proposing an order for a command from other replicas, we use the term *command-leader*. A *command-leader* is a replica that proposes an order for the commands received from its clients. For clarity, all replica can be command-leaders. To ensure that client commands are consistently executed across all correct replicas, EZBFT exploits the following concepts.

EZBFT uses the concept of *command interference* to empower replicas to make independent commit decisions. If replicas concurrently propose commands that do not interfere, they can be committed and executed independently, in parallel, in any order, and without the knowledge of other non-interfering commands. However, when concurrent commands do interfere, replicas must settle on a common sequential execution.

Command Interference: A command encapsulates an operation that must be executed on the shared state. We say that two commands L_0 and L_1 are interfering if the execution of these commands in different orders on a given state will result in two final states. That is, if there exists a sequence of commands Σ such that the serial execution of Σ, L_0, L_1 is not equivalent to Σ, L_1, L_0, then L_0 and L_1 are interfering.

Instance Space: An instance space can be visualized as a sequence of numbered slots to which client-commands can be associated with. The sequence defines the execution order of requests, and the role of a consensus protocol is to reach agreement among a set of replicas on a common order. However, to accommodate our requirement that every replica can be a command-leader for their received requests, each replica has its own instance space. Thus, EZBFT's role is not only to reach consensus on the mapping of client-commands to the slots within an instance space, but also among the slots in different instance spaces.

Instance Number: An instance number, denoted I, is a tuple of the instance space (or replica) identifier and a slot identifier.

Instance Owners: An owner number, O, is a monotonically increasing number that is used to identify the owner of an instance space. Thus, there are as many owner numbers as there are instance spaces (or replicas). This number becomes useful when a replica is faulty, and its commands must be recovered by other replicas. When replicas fail, another correct replica steps up to take ownership of the faulty replica's instance space. The owner of a replica R_0's instance space can be identified from its owner number using the formula $O_{R0} \bmod N$, where N is the number of replicas. Initially, the owner number of each instance space is set to the owner replica's identifier (e.g., $O_{R0} = 0$, $O_{R1} = 1$, and so on).

Dependencies: Due to the use of per-replica instance spaces, the protocol must agree on the relationship between the slots of different instances spaces. EZBFT does this via dependency collection, which uses the command interference relation. The dependency set $\mathcal{D}$ for command L is every other command L' that interferes with L.

Sequence Number ($\mathcal{S}$): is a globally shared number that is used to break cycles in dependencies. It starts from one and is always set to be larger than all the sequence numbers of the interfering commands. Due to concurrency, it is possible that interfering commands originating from different command-leaders are assigned the same sequence number. In such cases, the replica identifiers are used to break ties.

Protocol Properties: EZBFT has the following properties:
1) **Nontriviality.** Any request committed and executed by a replica must have been proposed by a client.
2) **Stability.** For any replica, the set of committed requests at any time is a subset of the committed requests at any later time. If at time t_1, a replica R_i has a request L committed at some instance I_L, then R_i will have L committed in I_L at any later time $t_2 > t_1$.
3) **Consistency.** Two replicas can never have different requests committed for the same instance.
4) **Liveness.** Requests will eventually be committed by every non-faulty replica, as long as at least $2f + 1$ replicas are correct.

IV. Protocol Design

In this section, we present EZBFT in detail, along with an informal proof of its properties. We have also developed a TLA+ specification of EZBFT and model-checked the protocol correctness; this can be found in the technical report [28].

A client command may either take a *fast path* or a *slow path*. The fast path consists of three communication steps, and is taken under no contention, no byzantine failures, and during synchronous communication periods. The *slow path* is taken otherwise to guarantee the final commit order for commands, and incurs two additional steps.

A. The Fast Path Protocol

The fast path consists of three communication steps in the critical path and one communication step in the non-critical path of the protocol. Only the communication steps in the critical path contribute to the client-side latency. The fast path works as follows.

> 1. Client sends a request to a replica.

The client c requests a command L to be executed on the replicated state by sending a message $\langle \text{REQUEST}, L, t, c \rangle_{\sigma_c}$ to a EZBFT replica. The closest replica may be chosen to achieve the optimal latency. The client includes a timestamp t to ensure exactly-once execution.

> 2. Replica receives a request, assigns an instance number, collects dependencies and assigns a sequence number, and forwards the request to other replicas.

When a replica R_i receives the message $m = \langle \text{REQUEST}, L, t, c \rangle_{\sigma_c}$, it becomes the command-leader for L. It assigns c to the lowest available instance number I_L in its instance space and collects a dependency set $\mathcal{D}$ using the command interference relation that was previously described. A sequence number $\mathcal{S}$ assigned for c is calculated as the maximum of sequence numbers of all commands in the dependency set. This information is relayed to all other replicas in a message $\langle \langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d \rangle_{\sigma_{Ri}}, m \rangle$, where $d = H(m)$, h is the digest of R_i's instance space, and O_{Ri} is its owner number.

Nitpick. Before taking the above actions, R_i ensures that the timestamp $t > t_c$, where t_c is the highest time-stamped request seen by R_i thus far. If not, the message is dropped.

> 3. Other replicas receive the SPECORDER message, speculatively executes the command according to its local snapshot of dependencies and sequence number, and replies back to the client with the result and an updated set of dependencies and sequence number, as necessary.

When replica R_j receives a message $\langle \langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d \rangle_{\sigma_{Ri}}, m \rangle$ from replica R_i, it ensures that m is a
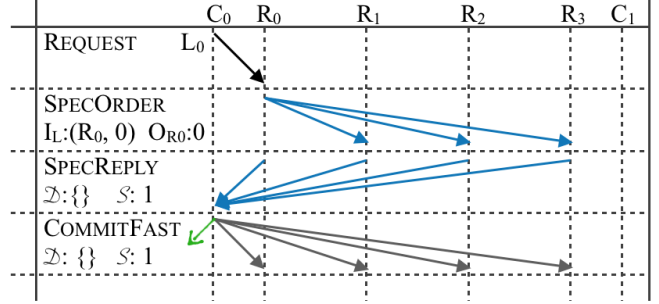

Figure 1: An example of a fast path execution.

valid REQUEST message and that $I_L = maxI_{Ri} + 1$, where $maxI_{Ri}$ is the largest occupied slot number in R_i's instance space. Upon successful validation, R_j updates the dependencies and sequence number according to its log, speculatively executes the command, and replies back to the client. A reply to the client consists of a message $\langle \langle \text{SPECREPLY}, O_{Ri}, I_L, \mathcal{D}', \mathcal{S}', d, c, t \rangle_{\sigma_{Ri}}, R_j, rep, SO \rangle$, where rep is the result, and $SO = \langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d \rangle_{\sigma_{Ri}}$.

> 4. The client receives the speculative replies and dependency metadata.

The client receives messages $\langle \langle \text{SPECREPLY}, O, I_L, \mathcal{D}', \mathcal{S}', d, c, t \rangle_{\sigma_{Rk}}, R_k, rep, SO \rangle$, where R_k is the sending replica. The messages from different replicas are said to match if they have identical O, I_L, $\mathcal{D}'$, $\mathcal{S}'$, c, t, and rep fields. The number of matched responses decides the fast path or slow path decision for command L.

Nitpick. Any two dependency sets are equal if both the sets have the same elements (commands).

> 4.1 The client receives $3f + 1$ matching responses.

The receipt of $3f+1$ matching responses from the replicas constitutes a fast path decision for command L. This happens in the absence of faults, network partitions, and contention. The client returns reply rep to the application and then asynchronously sends a message $\langle \text{COMMITFAST}, c, I_L, CC \rangle$, where CC is the commit certificate consisting of $3f + 1$ matching SPECREPLY responses, and returns.

> 5. The replicas receive either a COMMITFAST or a COMMIT message.

> 5.1 The replicas receive a COMMITFAST message.

Upon receipt of a $\langle \text{COMMITFAST}, c, I_L, CC \rangle$ message, the replica R_i marks the state of L as committed in its local log and enqueues the command for final execution. The replica does not reply back to the client.

Example. Figure 1 shows an example case. The client sends a signed REQUEST message to replica R_0 to execute

a command L_0 on the replicated service. Replica R_0 assigns the lowest available instance number in its instance space to L_0. Assuming that no instance number was used previously, the instance number assigned to L_0 is $I_{L_0} = \langle r_0, 0 \rangle$. Then, R_0 collects dependencies and assigns a sequence number to L_0. As the first command, there exists no dependencies, so the dependency set $\mathcal{D} = \{\}$. Thus, the sequence number is $\mathcal{S} = 1$.

A signed SPECORDER message is sent to other replicas in the system with the command and compiled metadata. Other replicas – R_1 through R_3 – receive this message, add the command to their log, and start amassing dependencies from their log that are not present in $\mathcal{D}$. No other replica received any other command, thus they produce an empty dependency set as well, and the sequence number remains the same. Since there are no dependencies, all replicas immediately execute the command, speculatively, on their copy of the application state. The result of execution, unchanged dependency set, sequence number, and the digest of log are sent in a SPECREPLY message to the client. The client checks for identical replies and returns the result to the application. The replies are identical in this case because no other command conflicts with L_0 in any of the replicas and the replicas are benign.

B. Execution Protocol

EZBFT uses *speculative execution* as a means to reply to the client quickly. However, the protocol must ensure that every correct replica has identical copies of the state. This means that, when necessary (as described in Section IV-C), the speculative state must be rolled back and the command must be re-executed correctly; this is called *final execution*.

Moreover, differently from existing BFT solutions, EZBFT collects command dependencies that form a directed graph with potential cycles. The graph must be processed to remove cycles and obtain the execution order for a command.

Each replica takes the following steps:
1) Waits for the command to be enqueued for execution. For final execution, wait for the dependencies to be committed and enqueued for final execution as well.
2) A dependency graph is constructed by including R and all its dependencies in $\mathcal{D}$ as nodes and adding edges between nodes indicating the dependencies. The procedure is repeated recursively for each dependency.
3) Strongly connected components are identified and sorted topologically.
4) Starting from the inverse topological order, every strongly connected component is identified, and all the requests within the component are sorted in the sequence number order. The requests are then executed in the sequence number order, breaking ties using replica identifiers. During speculative execution, the execution is marked *speculative* on the shared state.

During final execution, the speculative results are invalidated, command re-executed, and marked *final*.

Note that speculative execution can happen in either the *speculative* state or in the *final* version of the state, which ever is the latest. However, for final execution, commands are executed only on the previous *final* version.

C. The Slow Path Protocol

The slow path is triggered whenever a client receives either unequal and/or insufficient SPECREPLY messages that is necessary to guarantee a fast path. The client will receive unequal replies if the replicas have different perspectives of the command dependencies, possibly due to contention or due to the presence of byzantine replicas. The case of insufficient replies happen due to network partitions or byzantine replicas.

The steps to commit a command in the slow path are as follows.

> 4.2 The client receives at least $2f + 1$ possibly unequal responses.

The client c sets a timer as soon as a REQUEST is issued. When the timer expires, if c has received at least $2f + 1$ $\langle\langle \text{SPECREPLY}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, d, c, t \rangle_{\sigma_{Ri}}, R_j, rep, SO \rangle$ messages, it produces the final dependency set and sequence number for L. The dependency sets from a known set of $2f + 1$ replicas are combined to form $\mathcal{D}'$. A new sequence number $\mathcal{S}'$ is generated if the individual dependency sets were not equal. c sends a $\langle \text{COMMIT}, c, I_L, \mathcal{D}', \mathcal{S}', CC \rangle_{\sigma_c}$ message to all the replicas, where CC is the commit certificate containing $2f + 1$ SPECREPLY messages that are used to produce the final dependency set.

Nitpick. Each command-leader specifies a known set of $2f + 1$ replicas that will form the slow path quorum, which is used by the client to combine dependencies when more than $2f + 1$ reply messages are received. This information is relayed to the clients by the respective command-leaders and is cached at the clients.

> 5.2 The replicas receive a COMMIT message.

Upon receipt of a $\langle \text{COMMIT}, c, I_L, \mathcal{D}', \mathcal{S}', CC \rangle_{\sigma_c}$ message, replica r updates command L's metadata with the received dependency set $\mathcal{D}'$ and sequence number $\mathcal{S}'$. The state produced after the speculative execution of L is invalidated, and L is enqueued for final execution. The result of final execution, rep is sent back to the client in a $\langle \text{COMMITREPLY}, L, rep \rangle$ message.

> 6.2 The client receives $2f + 1$ COMMITREPLY messages and returns to the application.

The client returns rep to the application upon receipt of $2f + 1$ $\langle \text{COMMITREPLY}, L, rep \rangle$ messages. At this point, execution of command L is guaranteed to be safe in the
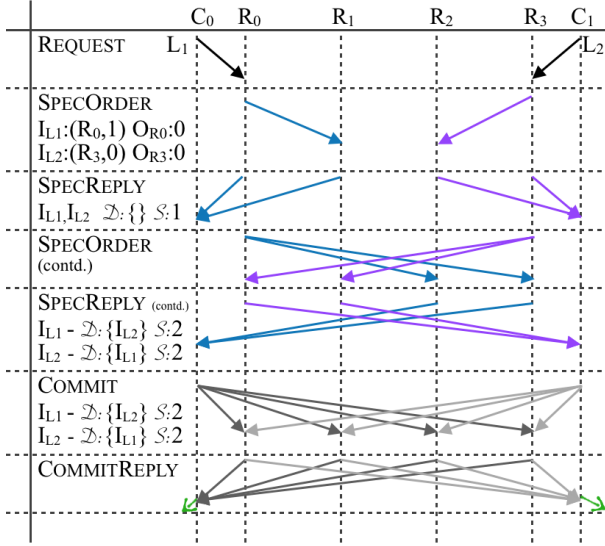
Figure 2: EZBFT: An example of a slow path execution.

Figure 3: An example of a slow path with a faulty replica.

system, while tolerating upto f byzantine failures. Moreover, even after recovering from failures, all correct replicas will always execute L at this same point in their history to produce the same result.

Example: Figure 2 shows an example of a slow path. Two clients c_0 and c_1 send signed REQUEST messages to replicas R_0 and R_3, respectively, to execute commands L_1 and L_2, respectively, on the replicated service. Assume that L_1 and L_2 conflict. Replica R_0 assigns the lowest available instance number of $\langle R_0, 0 \rangle$ to L_1. Thus, R_0 collects a dependency set $\mathcal{D}_{L_1} = \{\}$ and assigns a sequence number $\mathcal{S}_{L_1} = 1$ to L_1. Meanwhile, R_3 assigns the instance number $\langle R_3, 0 \rangle$ to L_1; the dependency set is $\mathcal{D}_{L_2} = \{\}$, and sequence number is $\mathcal{S}_{L_2} = 1$. Replicas R_0 and R_3 send SPECORDER messages with their respective commands and their metadata to other replicas. Let's assume that R_0 and R_1 receive L_1 before L_2, while R_2 and R_3 receive L_2 before L_1. The dependency set and sequence number will remain unchanged for L_1 at R_0 and R_1, because the dependency set in the SPECORDER message received is the latest. However, the dependency set and sequence number for L_2 will update to $\mathcal{D}'_{L_2} = \{L_1\}$ and $\mathcal{S}'_{L_2} = 2$, respectively. Similarly, the dependency set and sequence number will remain unchanged for L_2 at R_0 and R_1, but for L_1, $\mathcal{D}'_{L_1} = \{L_2\}$ and $\mathcal{S}'_{L_1} = 2$, respectively. Replicas send SPECREPLY messages for both L_1 and L_2 with the new metadata to the respective clients c_0 and c_1.

Let's assume that the slow quorum replicas are R_0, R_1, and R_2 for R_0, and R_1, R_2, and R_3 for R_3. Since client c_0 observes unequal responses, it combines the dependencies for L_1 from the slow quorum and selects the highest sequence number to produce the final dependency set $\mathcal{D}_{L_1} = \{L_2\}$ and sequence number $\mathcal{S}_{L_1} = 2$. This metadata is sent to the replicas in a COMMIT message. Similarly,
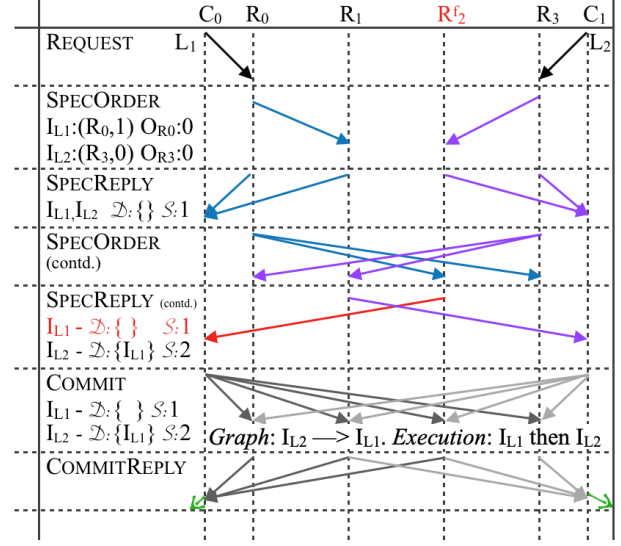
c_1 produces the final dependency set $\mathcal{D}_{L_2} = \{L_1\}$ and sequence number $\mathcal{S}_{L_2} = 2$ for L_2, and sends a COMMIT message to the replicas. The replicas update the dependency set and sequence number of the commands upon receipt of the respective COMMIT messages, and the commands are queued for execution. The replicas wait for the receipt of the COMMIT messages of all commands in the dependency set before processing them.

After the construction of the graph and the inverse topological sorting, there will exist commands L_1 and L_2 with a cyclic dependency between them. Since the sequence numbers for both the commands are the same and thus cannot break the dependency, the replica IDs are used in this case. Thus, L_1 gets precedence over L_2. L_1 is executed first, followed by L_2. The result of the executions are sent back to the clients. The clients collect $2f+1$ reply messages and return the result to the application.

Example with a faulty replica: Figure 3 shows an example of the slow path that is very similar to that of Figure 2, but with a faulty replica R_2 that misbehaves. Notice that the REQUEST and SPECORDER steps (first four rows) remain the same. Upon receipt of the SPECORDER message from R_0 and R_3 for L_1 and L_2, respectively, the replicas collect the dependency set and update the sequence number, and send back SPECREPLY messages to the client. For L_1, R_0 and R_1 send $\mathcal{D}'_{L_1} = \{\}$ and $\mathcal{S}'_{L_1} = 1$; however, R_2 misbehaves and sends $\mathcal{D}'_{L_1} = \{\}$ and $\mathcal{S}'_{L_1} = 1$, even though it received L_2 before L_1. For L_2, R_2 and R_3 send $\mathcal{D}'_{L_2} = \{\}$ and $\mathcal{S}'_{L_2} = 1$; R_1 sends $\mathcal{D}'_{L_2} = \{L_1\}$ and $\mathcal{S}'_{L_2} = 2$. c_0 receives a quorum of $2f + 1$ SPECREPLY messages, and sends a COMMIT message with an empty dependency set and a sequence number 1. On the other hand, the correct R_1 that participated in command L_1's quorum sends back the correct

dependency set and sequence number. Therefore, the final commit message for L_2 will have L_1 in its dependency set. Thus, even though replicas immediately execute L_1 since L_1's final dependency set is empty, they cannot do so for L_2. Correct replicas must wait until L_1 is *committed* before constructing the graph, at which point L_1 will be executed first due to the smallest sequence number, followed by L_2.

D. Triggering Owner Changes

EZBFT employs a mechanism at the clients to monitor the replicas and take actions to restore the service when progress is not being made. Although the slow path helps overcome the effects of a participant byzantine replica, it does ensure progress of a command when its command-leader, the replica that proposed that command, is byzantine. From the client-side, two events can be observed to identify misbehaving command-leaders.

> 4.3 The client times-out waiting for reply from the replicas.

After the client sends a request with command L, it starts another timer, in addition to the one for slow-path, waiting for responses. If the client receives zero or fewer than $2f + 1$ responses within the timeout, it sends the $\langle \text{REQUEST}, L, t, c, R_i \rangle_{\sigma_c}$ message to all the replicas, where R_i is the original recipient of the message.

When replica R_j receives the message, it takes one of the following two actions. If the request matches or has a lower timestamp t than the currently cached timestamp for c, then the cached response is returned to c. Otherwise, the replica sends a $\langle \text{RESENDREQ}, m, R_j \rangle$ message where $m = \langle \text{REQUEST}, L, t, c, R_i \rangle_{\sigma_c}$ to R_i and starts a timer. If the timer expires before the receipt of a SPECORDER message, R_j initiates an ownership change.

> 4.4 The client receives responses indicating inconsistent ordering by the command-leader and sends a proof of misbehavior to the replicas to initiate an ownership change for the command-leader's instance space.

Even though a client may receive prompt replies from the replicas, it must check for inconsistencies leading to a *proof of misbehavior* against the command-leader. The $\langle \langle \text{SPECREPLY}, O_{Ri}, I_L, \mathcal{D}', \mathcal{S}', d, c, t \rangle_{\sigma_{Ri}}, R_k, rep, SO \rangle$ messages from different replicas are said to match if they have identical O_{Ri}, I, $\mathcal{D}$, $\mathcal{S}$, c, t, and rep fields, but the contention may affect the equality of the dependency set and sequence number fields. Thus, the command-leader is said to misbehave if it sends SPECORDER messages with different instance numbers to different replicas (i.e., if the I field varies between the replicas). The client c can identify by inspecting SPECORDER SO message embedded in the SPECREPLY message received from the replicas. In this case, the client collects a pair of such messages to

construct a $\langle \text{POM}, O_{Ri}, POM \rangle$ message, where POM is a pair of SPECREPLY messages, proving misbehavior by the command-leader R_i of L.

E. The Owner Change Protocol

An ownership change is triggered for an instance space if its original owner is *faulty*. However, to initiate an ownership change, there must exist either a proof of misbehavior against the owner, or enough replicas must have timed out waiting for a reply from the owner.

A replica R_j commits to an ownership change by sending a $\langle \text{STARTOWNERCHANGE}, R_i, O_{Ri} \rangle$ message to other replicas, where R_i is the suspected replica and O_{Ri} is its owner number.

When another replica R_k receives at least $f + 1$ START-OWNERCHANGE messages for R_i, it commits to an ownership change. From this point forward, R_k will not participate in R_i's instance space. The new owner number is calculated as $O'_{Ri} = O_{Ri} + 1$, and the new command-leader is identified using $O'_{Ri} \bmod N$ (henceforth R_l). Replicas that have committed to an owner-change send $\langle \text{OWNERCHANGE} \rangle$ messages to the new leader. Once the new command-leader R_l receives $f + 1$ OWNERCHANGE messages, it becomes the new owner of R_i's instance space and finalizes its history.

Each replica sends an OWNERCHANGE message containing its view of R_i's instance space, i.e., the instances (speculative) executed or committed since the last checkpoint, and the commit-certificate with the highest owner number that it had previously responded to with a commit message, if any. The new owner collects a set P of OWNERCHANGE messages and selects only the one that satisfies one of the following conditions. For clarity, we label the sequence of instances in each OWNERCHANGE message as P_i, P_j, and so on.

There exists a sequence P_i that is the longest and satisfies one of the following conditions.

Condition 1 P_i has COMMIT messages with the highest owner number to prove its entries.

Condition 2 P_i has at least $f + 1$ SPECREPLY messages with the highest owner number to prove its entries.

If there exists a sequence P_j that extends a P_i satisfying any of the above conditions, then P_j is a valid extension of P_i if one of the following conditions hold:

1) P_i satisfies Condition 1, and for every command L in P_j not in P_i, L has at least $f + 1$ SPECREPLY messages with the same highest order number as P_i.
2) P_i satisfies Condition 2, and for every command L in P_j not in P_i, L has a signed COMMIT message with the same highest order number as P_i.

The new owner sends a NEWOWNER message to all the replicas. The message includes the new owner number O'_{R0}, the set P of OWNERCHANGE messages that the owner collected as a proof, and the set of safe instances G produced using Condition 1 and Condition 2. A replica accepts a

NEWOWNER message if it is valid, and applies the instances from G in R_i's instance space. If necessary, it rolls-back the speculatively executed requests and re-executes them again.

At this point, R_i's instance space is frozen. No new commands are ordered in the instance space, because each replica has its own instance space that it can use to propose its command. The owner change is used to ensure the safety of commands proposed by the faulty replicas.

F. Correctness

We formally specified EZBFT in TLA+ and model-checked using the TLC model checker. The actions of Byzantine replicas were modelled after inspirations from the BFT Paxos specification in [27]. The TLA+ specification is provided in the technical report [28]. In this section, we provide an intuition of how EZBFT achieves its properties.

Nontriviality. Since clients must sign the requests they send, a malicious primary replica cannot modify them without being suspected. Thus, replicas only execute requests proposed by clients.

Consistency. To prove consistency, we need to show that if a replica R_j commits L at instance I, then for any replica R_k that commits L' at I, L and L' must be the same command.

To prove this, consider the following. If R_j commits L at $I = \langle R_i, - \rangle$, then an order change should have been executed for replica R_i's instance space. If R_j is correct, then it would have determined that L was executed at I using the commit certificate in the form of SPECORDER or COMMIT messages embedded within the CHANGEOWNER messages. Thus, L and L' must be the same. If R_j is malicious, then the correct replicas will detect it using the invalid progress-certificate received. This will cause an ownership change.

In addition, we need to also show that conflicting requests L and L' are committed and executed in the same order across all correct replicas. Assume that L commits with $\mathcal{D}$ and $\mathcal{S}$, while L' commits with $\mathcal{D}'$ and $\mathcal{S}'$. If L and L' conflict, then at least one correct replica must have responded to each other in the dependency set among a quorum of $2f + 1$ replies received by the client. Thus, L will be in L''s dependency set and/or viceversa. The execution algorithm is deterministic, and it uses the sequence number to break ties. Thus, conflicting requests will be executed in the same order across all correct replicas.

Stability. Since only f replicas can be byzantine, there must exist at least $2f + 1$ replicas with the correct history. During an ownership change, $2f + 1$ replicas should send their history to the new owner which then validates it. Thus, if a request is committed at some instance, it will be extracted from history after any subsequent owner changes and committed at same instance at all correct replicas.

Liveness. Liveness is guaranteed as long as fewer than f replicas crash. Each primary replica attempts to take the fast

path with a quorum of $3f + 1$ replicas. When faults occur and a quorum of $3f + 1$ replicas is not possible, the client pursues the slow path with $2f + 1$ replicas and terminates in two additional communication steps.

Consistency Guarantees. EZBFT, by default, provides per-object linearizability [30] for transitive operations, or specifically those that target a single object. For multi-object operations, the equivalent property is *strict-serializability*, which EZBFT provides since, each command is enqueued for final execution only after the commands in its dependency set are committed [24].

V. EVALUATION

We implemented EZBFT, and its state-of-the-art competitors PBFT, FaB, and Zyzzyva in Go, version 1.10. In order to evaluate all systems in a common framework, we used gRPC [31] for communication and protobuf [32] for message serialization. We used the HMAC [33] and ECDSA [29] algorithms in Go's *crypto* package to authenticate the messages exchanged by the clients and the replicas. The systems were deployed in different Amazon Web Service (AWS) regions using the EC2 infrastructure [22]. The VM instance used was m4.2xlarge with 8 vCPUs and 32GB of memory, running Ubuntu 16.04. We implemented a replicated key-value store to evaluate the protocols. Note that, for Zyzzyva and EZBFT, the client process implements the logic for the client portion of the respective protocols.

The key-value store is in-memory, mapping string keys to byte array values. In our experiment, the keys and values were 8 bytes and 16 bytes, respectively. The store was initialized with 10,000 keys and random values.

Among the protocols evaluated, only EZBFT is affected by contention. Contention, in the context of a replicated key-value store, is defined as the percentage of requests that concurrently access the same key. Prior work [24] has shown that, in practice, contention is usually between 0% and 2%. Thus, a 2% contention means that roughly 2% of the requests issued by clients target the same key, and the remaining requests target clients' own (non-overlapping) set of keys. However, we evaluate EZBFT at higher contention levels for completeness.

A. Client-side Latency

To understand EZBFT's effectiveness in achieving optimal latency at each geographical region, we devised two experiments to measure the latency experienced by clients located at each region for each of the protocols.

Experiment 1: We deployed the protocols with four replica nodes in the AWS regions: US-East-1 (Virginia), India, Australia, and Japan. At each node, we also co-located a client that sends requests to the replicas. For single primary-based protocols (PBFT, FaB, Zyzzyva), the primary was set to US-East replica; thus, clients in other replicas send their requests to the primary. For EZBFT, the client
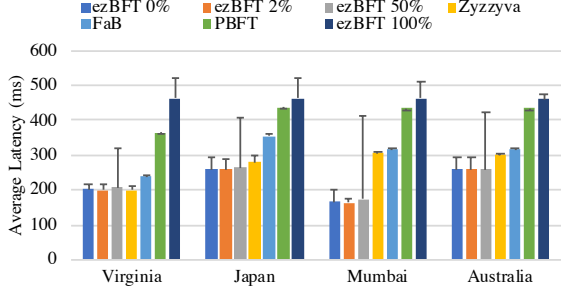
Figure 4: Median latencies for Experiment 1. All primaries are in US-East-1 region. The latency is shown per region as recorded by the clients in that region. Error bars indicate 99%ile.

sends its requests to the nearest replica (which is in the same region). The clients send requests in closed-loop, meaning that a client will wait for a reply to its previous request before sending another one.

Figure 4 shows the median latency (in milliseconds) observed by the clients located in their respective regions (shown on x-axis) for each of the four protocols. For EZBFT, the latency was measured at different contention levels: 0%, 2%, 50%, and 100%; the suffix in the legend indicates the contention. Among primary-based protocols, PBFT suffers the highest latency, because it takes five communication steps to deliver a request. FaB performs better than PBFT with four communication steps, but Zyzzyva performs the best among primary-based protocols using only three communication steps. Overall, EZBFT performs as good as or better than Zyzzyva, for up to 50% contention. In the US-East-1 region, both Zyzzyva and EZBFT have about the same latency because they have the same number of communication steps and their primaries are located in the same region. However, for the remaining regions, Zyzzva clients must forward their requests to US-East-1, while EZBFT clients simply send their requests to their local replica, which orders them. At 100% contention, five communication steps required for total-order increases EZBFT's latency close to that of PBFT's.

Experiment 2: To better understand Zyzzyva's best and worst-case performances and how they fare against EZBFT, we identified another set of AWS regions: US-East-2 (Ohio), Ireland, Frankfurt, and India. This experiment was run similar to that of Figure 4. The primary was placed in Ireland. The results are shown in Figure 5. Figure 5a shows the median latencies as observed by the clients in each deployed region for each of the four protocols. The choice of Ireland as the primary region represents the best case for Zyzzyva. Hence, EZBFT performs very similar to Zyzzyva.

In Experiment 1, the regions mostly had non-overlapping paths between them, and thus the first communication step of sending the request to the leader can be seen clearly (notice

Mumbai in Figure 4. On the other hand, in Experiment 2, connections between the regions have overlapping paths. For example, sending a request from Ohio to Mumbai for EZBFT will take about the same time as sending a request from Ohio to Mumbai via the primary at Ireland for Zyzzyva.

Figure 5b shows the effect of moving the primary to different regions. We disregard PBFT and FaB in this case, as their performance do not improve. For Zyzzyva, moving the primary to US-East-2 or India substantially increases its overall latency. In such cases, EZBFT's latency is up to 45% lower than Zyzzyva's. This data-point is particularly important as it reveals how the primary's placement affects the latency.

To curb the negative effects of byzantine primary replicas, in [34], the authors propose to frequently move the primary (this strategy is adopted by other protocols including Zyzzyva). From Figure 5b, we can extrapolate that such frequent movements can negatively impact latencies over time. Given these challenges, we argue that EZBFT's leaderless nature is a better fit for geo-scale deployments.

Effect of Byzantine Replicas. While a byzantine primary, in primary-based approaches, can cease progress, EZBFT enables correct replicas to continue to make progress using the slow path, effectively isolating the impact of byzantine replicas. Therefore, the client-side latency observed for a four-replica system will be equivalent to that observed at 100% contention with a small change: the requests at the affected node will timeout until before the view change. After the view change, the new owner will complete any pending requests. The new requests, however, will be sent to the closest node(s), so the latency recedes quickly. In our evaluation, with a timeout period of 3 seconds, the view change procedure completed in less than 4 seconds. For more details, refer to technical report [28].

B. Client Scalability

Another important aspect of EZBFT is its ability to maintain low client-side latency even as the number of connected clients increases. For this experiment, we deployed the protocols in Virginia, Japan, Mumbai, and Australia, and measured client-side latency per region by varying the number of connected clients. Figure 7 shows the results. Notice that as Zyzzyva approaches 100 connected clients per region, it suffers from an exponential increase in latency. However, EZBFT, even at 50% contention, is able to scale better with the number of clients. Particularly, in Mumbai, EZBFT maintains a stable latency even at 100 clients per region, while Zyzzyva's latency shoots up.

C. Server-side Throughput

We also measured the peak throughput of the protocols. For this experiment, we deployed the protocols in five AWS regions: US-East-1 (Virginia), India, Australia, and Japan.
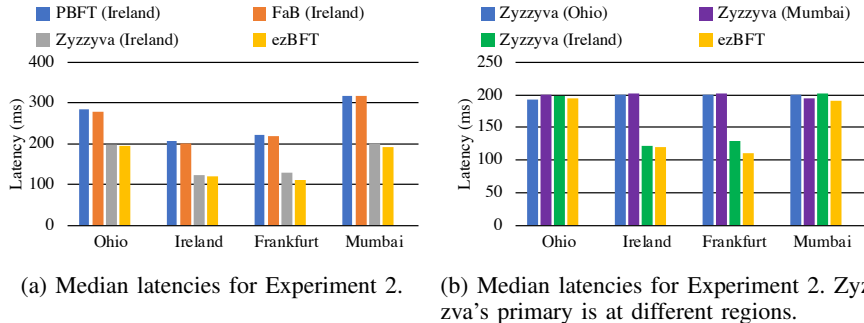
(a) Median latencies for Experiment 2.

(b) Median latencies for Experiment 2. Zyzzva's primary is at different regions.

Figure 5: Best case for Zyzzyva and the effect of moving primary to different regions. Experiments reveal EZBFT's effectiveness. Legend entries show primary's location in parenthesis.
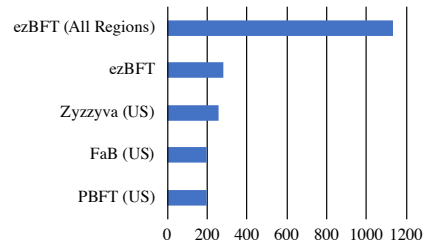

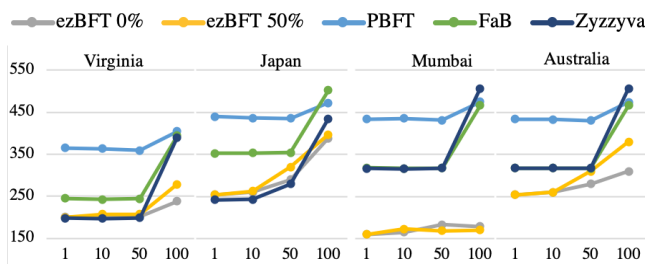Figure 6: Throughput of EZBFT and competitor BFT protocols.

Figure 7: Latency per location while varying the number of connected clients (1 – 100) per region.

We co-located ten clients with the primary replica in US-East-1. Unlike the experiments so far, here the clients send requests in an open-loop, meaning that they continuously and asynchronously send requests before receiving replies.

Figure 6 shows the results. For EZBFT, we carried out two experiments: i) clients are placed only at US-East-1 (labelled EZBFT in the figure); and ii) clients are placed at every region (labelled "EZBFT (All Regions)" in the figure). Due to the leaderless characteristic, each of the replicas can feed requests into the system, increasing the overall throughput. The contention was set to 0%, and no batching was done.

Observe that when clients are placed only at US-East-1, EZBFT performs at par or slightly better than others. On the other hand, when clients are placed in all the other regions, which does not yield any benefit for other protocols, EZBFT's throughput increases by as much as four times, as all EZBFT replicas are able to process and deliver requests concurrently.

VI. RELATED WORK

BFT consensus was first introduced in [35]. However, PBFT [11], [34] was the first protocol to provide a practical implementation of BFT consensus in the context of state machine replication. PBFT solved consensus in three communication steps (excluding two steps for client communication) using $3f+1$ nodes and requiring responses from at

least $2f+1$ nodes. f is the maximum number of byzantine faulty nodes that the system can tolerate and make progress.

FaB [13] reduced the number of communication steps required to reach agreement in the common case to two steps, called two-step consensus (excluding two steps for client communication). However, node and quorum requirements are substantially larger with $5f+1$ and $4f+1$ nodes, respectively, to tolerate f faults and still achieve two-step consensus. FaB falls back to a slower path when $4f+1$ responses cannot be acquired, and requires an extra communication step and at least $3f+1$ nodes to reach agreement.

FaB was the first BFT protocol to not require any digital signatures in the common case. Parameterized-FaB [13] requires $3f+2t+1$ nodes, where f is the number of tolerated faults and preserve safety and t is the number of tolerated faults and solve two-step consensus. This minimized the node requirements in the common case to $3f+1$ (by setting $t = 0$), but the quorum requirement is still more than that of PBFT.

The Q/U [12] protocol was the first to achieve consensus in two communications steps when there are no faults and update requests do not concurrently access the same object. Q/U defines a simplified version of conflicts. Requests are classified as either reads or writes. Reads do not conflict with reads, while write conflicts with reads and writes. This is more restrictive than the commutative property used by EZBFT. In EZBFT, for instance, mutative operations (such as incrementing a variable) are commutative.

HQ [15] is similar to PBFT with a special optimization to execute read-only requests in two communication steps and update requests in four communication steps under no conflicts. HQ's definition of conflict is the same as Q/U's.

Zyzzyva [14], [36] uses $3f+1$ nodes to solve consensus in three steps (including client communication), requiring a quorum of $3f+1$ responses. The protocol tolerates f faults, so it takes an additional two steps when nodes are faulty. Zyzzyva uses the minimum number of nodes, com-

munication steps, and one-to-one messages to achieve fast consensus. It is cheaper than the aforementioned protocols, but also more complex. Zyzzyva's performance boost is due to speculative execution, active participation of the clients in the agreement process, and tolerating temporary inconsistencies among replicas.

EZBFT has the same node and quorum requirements as well as the number of communication steps as Zyzzyva. However, by minimizing the latency of the first communication step and alleviating the specialized role of the primary, EZBFT reduces the request processing latency.

Aliph [18] builds a BFT protocol by composing three different sub-protocols, each handling a specific system factor such as contention, slow links, and byzantine faults. Under zero contention, the sub-protocol *Quorum* can deliver agreement in two steps with $3f + 1$ nodes by allowing clients to send the requests directly to the nodes. However, as contention or link latency increases, or as faults occur, Aliph switches to the sub-protocol *Chain* whose additional steps is equal to the number of nodes in the system, or to the sub-protocol *Backup* which takes at least three steps. Although the idea of composing simpler protocols is appealing in terms of reduced design and implementation complexities, the performance penalty is simply too high, especially in geo-scale settings.

In contrast, EZBFT exploits the trade-off between the slow and fast path steps. EZBFT uses three steps compared to Aliph's two steps in the common case, and in return, provides slow path in only two extra communication steps unlike Aliph. Moreover, EZBFT's leaderless approach reduces the latency of the first communication step to near zero, yielding client-side latency comparable to Aliph's.

EBAWA [37] uses the spinning primary approach [23] to minimize the client-side latency in geo-scale deployments. However, a byzantine replica can delay its commands without detection reducing the overall server-side throughput. EZBFT's dependency collection mechanism enables correct replicas to only depend on commands that arrive in time, and execute without waiting otherwise. Furthermore, EBAWA uses a trusted component in each replica making it a hybrid protocol [38] than a byzantine fault-tolerant one.

Table II summarizes the comparison of existing work with EZBFT. Note that EZBFT and Zyzzyva have the same best-case communication steps. However, for EZBFT, the latency for the first-step communication is minuscule (tending towards zero) compared to that of Zyzzyva's.

Leaderless and multi-leader protocols [25], [26], [24], [2] have been proposed for the CFT model. Among these, EPaxos [24] and Caesar [25] collect dependencies and find a total order among all the dependent requests. Both protocols work in two phases: a fast phase that is reached under no contention and an additional slow phase that is required un-

Table II: Comparison of existing BFT protocols and EZBFT.

Protocol	PBFT	Zyzzyva	Aliph	ezBFT
Resilience	$f < n/3$	$f < n/3$	$f < n/3$	$f < n/3$
Best-case comm. steps	5	3	2	3
Best-case comm. steps in absence of ...	Byz. Slow links	Byz. Slow links Contention	Byz. Slow links Contention	Byz. Slow links Contention
Slow-path steps	-	2	n + 3[1]	2
Leader	Single	Single	Single	Leaderless

der contention. In EPaxos, the collected dependencies form a graph, which is linearized before executed. EZBFT borrows the EPaxos' dependency collection technique and enhances it with *speculative execution* and *client-side validation* in order to decrease the number of communication steps as well as the number of messages exchanged in each path.

Caesar can deliver fast phase consensus even under non-trivial contention by having a replica wait until some conditions are satisfied before replying to the primary. Such wait conditions are harmful in BFT protocols, because a malicious replica can use this as an opportunity to cease progress.

In M^2Paxos [26], a replica can order a request if it *owns* the object that the request accesses. Otherwise, it forwards the request to the right owner or acquires ownership. Acquiring ownership means becoming the primary for some subset of objects, and in CFT-based protocols, any replica can propose to be a owner of any subset of objects at any point in time. However, in BFT-based protocols, electing a primary is a more involved process requiring consent from other replicas. In addition, view numbers are pre-assigned to replicas; therefore, randomly choosing primaries is not a trivial process.

VII. Conclusions

State-of-the-art BFT protocols are not able to provide optimal request processing latencies in geo-scale deployments – an increasingly ubiquitous scenario for many distributed applications, particularly blockchain-based applications.

We presented EZBFT, a leaderless BFT protocol that provides three-step consensus in the common case, while essentially nullifying the latency of the first communication step. EZBFT provides the classic properties of BFT protocols including nontriviality, consistency, stability, and liveness. Our experimental evaluation reveals that EZBFT reduces latency by up to 40% compared to Zyzzyva.

[1] n additional steps are required only if *Chain* sub-protocol is used

REFERENCES

[1] A. Bessani, J. a. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 355–362. [Online]. Available: https://doi.org/10.1109/DSN.2014.43

[2] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 369–384.

[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 2014, pp. 305–319.

[4] L. Lamport *et al.*, "Paxos made simple," Tech. Rep., 2001.

[5] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, ser. DSN '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 245–256. [Online]. Available: http://dx.doi.org/10.1109/DSN.2011.5958223

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[7] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Lon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2011.

[8] "CockroachDB," https://github.com/cockroachdb/cockroach, accessed: 2018-11-06.

[9] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukoli, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. ACM, pp. 30:1–30:15.

[10] A. Bessani, J. a. Sousa, and M. Vukolić, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, ser. SERIAL '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:2. [Online]. Available: http://doi.acm.org/10.1145/3152824.3152830

[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296824

[12] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 59–74. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095817

[13] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," vol. 3, no. 3, pp. 202–215.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," pp. 45–58, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294267

[15] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USENIX Association, 2006, pp. 177–190.

[16] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, vol. 00, June 2008, pp. 197–206. [Online]. Available: doi.ieeecomputersociety.org/10.1109/DSN.2008.4630088

[17] J. Li and D. Mazires, "Beyond one-third faulty replicas in byzantine fault tolerant systems," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07. USENIX Association, 2007, pp. 10–10.

[18] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 363–376. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755950

[19] F. Tian, "An agri-food supply chain traceability system for china based on rfid amp;amp; blockchain technology," in *2016 13th International Conference on Service Systems and Service Management (ICSSSM)*, June 2016, pp. 1–6.

[20] A. Spielman, "Blockchain: digitally rebuilding the real estate industry," Ph.D. dissertation, Massachusetts Institute of Technology, 2016.

[21] I. Eyal, "Blockchain technology: Transforming libertarian cryptocurrency dreams to finance and banking realities," *Computer*, vol. 50, no. 9, pp. 38–49, 2017.

[22] Amazon Inc., "Elastic Compute Cloud," http://aws.amazon.com/ec2.

[23] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '09. IEEE Computer Society, pp. 135–144.

[24] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13.   ACM, 2013, pp. 358–372.

[25] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, "Speeding up consensus by chasing fast decisions," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 49–60.

[26] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 156–167.

[27] L. Lamport, "Byzantizing Paxos by refinement," in *Proceedings of the 25th International Conference on Distributed Computing*, ser. DISC'11.   Springer-Verlag, 2011, pp. 211–224.

[28] B. Arun, S. Peluso, and B. Ravindran, "ezBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication," April 2019. [Online]. Available: http://arxiv.org/abs/1904.06023

[29] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *Int. J. Inf. Secur.*, vol. 1, no. 1, pp. 36–63, Aug. 2001. [Online]. Available: http://dx.doi.org/10.1007/s102070100002

[30] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[31] "gRPC," https://github.com/grpc/.

[32] Google, "Protocol buffers," https://developers.google.com/protocol-buffers/.

[33] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication."

[34] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/571637.571640

[35] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," vol. 4, no. 3, pp. 382–401, 1982.

[36] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, 2010.

[37] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "EBAWA: Efficient byzantine agreement for wide-area networks," in *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, Nov 2010, pp. 10–19.

[38] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: SGX-based high performance BFT," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17.   ACM, 2017, pp. 222–237.