Egalitarian Byzantine Fault Tolerance

Michael Eischer and Tobias Distler
Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Email: {eischer,distler}@cs.fau.de

***Abstract*—Minimizing end-to-end latency in geo-replicated systems usually makes it necessary to compromise on resilience, resource efficiency, or throughput performance, because existing approaches either tolerate only crashes, require additional replicas, or rely on a global leader for consensus. In this paper, we eliminate the need for such tradeoffs by presenting ISOS, a leaderless replication protocol that tolerates up to f Byzantine faults with a minimum of $3f + 1$ replicas. To reduce latency in wide-area environments, ISOS relies on an efficient consensus algorithm that allows all participating replicas to propose new requests and thereby enables clients to avoid delays by submitting requests to their nearest replica. In addition, ISOS minimizes overhead by limiting message ordering to requests that conflict with each other (e.g., due to accessing the same state parts) and by already committing them after three communication steps if at least $f + 1$ replicas report each conflict. Our experimental evaluation with a geo-replicated key-value store shows that these properties allow ISOS to provide lower end-to-end latency than existing protocols, especially for use-case scenarios in which the clients of a system are distributed across multiple locations.**

***Index Terms*—State-Machine Replication, Byzantine Fault Tolerance, Geo-Replication, Leaderless Consensus**

I. Introduction

Distributing a replicated service across several geographic sites offers the possibility to make the service resilient against a wide spectrum of faults, including failures of entire data centers. Unfortunately, traditional state-machine replication approaches [1], [2] in such environments incur high latency due to electing a leader replica which is then responsible for establishing a total order on all incoming client requests. Relying on a single global leader replica in wide-area environments comes with the major drawbacks of (1) creating a potential performance bottleneck, (2) disadvantaging clients that reside at a greater distance to the current leader, and (3) introducing response-time volatility, because overall latency can vary significantly depending on where the acting leader is located. Although it is possible to rotate the leader role among replicas [3], this technique only slightly mitigates the problem since the rotation process itself introduces coordination overhead in the form of (at least) an additional communication step.

Several existing works [4], [5], [6], [7], [8] address these issues by building on the insight that for guaranteeing linearizability [9] it is not actually necessary to totally order all client requests that are submitted to a service. Instead, the efficiency of message ordering in many cases can be improved by taking the semantics of requests into account [10] and only ordering those requests that conflict with each other, for example due to operating on the same application-state variables. In recent years, applications of this principle led to a variety of protocols that explore different points in the design space of replicated systems. Specifically, this includes protocols that have been designed to tolerate crashes [5], [6], Byzantine fault-tolerant (BFT) protocols achieving efficiency at the cost of additional replicas [4], [8], as well as protocols that rely on a global leader replica to ensure progress in case of disagreements between different replicas [7]. While on the one hand illustrating the effectiveness and flexibility of the underlying concept, this variety of protocols on the other hand also means that existing approaches require compromising on resilience, resource efficiency, or throughput performance.

To eliminate the need for such tradeoffs, our goal was to develop a protocol that combines all three desirable properties while still providing low latency. The result of our efforts is ISOS, a state-machine replication protocol that tolerates Byzantine faults, demands only the minimum group size necessary for BFT in asynchronous environments (i.e., $3f + 1$ replicas to tolerate f faults), and operates without global leader replica. To minimize end-to-end latency in geo-replicated settings, ISOS offers a fast path that enables replicas to execute client requests after three consensus communication steps if either (a) there currently are no conflicting requests or (b) each conflict is identified by at least $f+1$ replicas. In the (typically rare) case in which none of the two scenarios applies, ISOS switches to a fallback path that is then responsible for resolving the discrepancies between replicas. Since neither of the two paths in ISOS requires the election of a global leader, we refer to this concept as *egalitarian* Byzantine fault tolerance.

In summary, this paper makes the following contributions: (1) It presents ISOS's efficient BFT consensus algorithm that only orders conflicting requests and avoids a global leader during both normal-case operation as well as conflict-discrepancy resolution. (2) It shows how ISOS's request-execution stage is able to safely operate with a bounded state, and this despite the fact that faulty replicas possibly introduce request-dependency chains of infinite length. (3) It details ISOS's checkpointing mechanism that enables the protocol to garbage-collect consensus information about already ordered requests; garbage collection is a relevant problem in practice, but often not implemented in other protocols (e.g., EPaxos [5]). (4) It formally proves the correctness of both ISOS's agreement and execution stage. Notice that due to space limitations, we limit Section IV-H to the presentation of a proof sketch; the full proof (as well as a pseudocode summary of ISOS's agreement protocol) is available in the extended version of this paper [11]. (5) It experimentally evaluates ISOS with a key-value store in a geo-distributed setting deployed in Amazon's EC2 cloud.

II. System Model

In this work we focus on stateful applications that are replicated across multiple servers for fault tolerance. To remain available even in the presence of data-center outages, the replicas of a system are hosted at different geographic sites, as illustrated in Figure 1. Clients of the service typically reside in proximity to one of the replicas, often within the same data center. As a result of such a setting, overall response times in our target systems are dominated by the latency induced by the state-machine replication protocol executed between servers.

We assume that the replicated service must provide safety in the presence of Byzantine faults as well as an asynchronous network. To further be able to ensure liveness despite the FLP impossibility [12], there need to be synchronous phases during which the one-way network delay between all pairs of replicas is below a threshold Δ, which is known to replicas. Clients and replicas communicate over the network by exchanging messages that are signed with the sender's private key, denoted as $\langle ... \rangle_{\sigma_i}$ for a sender i. Recipients immediately discard messages in case they are unable to verify the signature.

Clients invoke operations in the application by submitting requests to the server side. With regard to the execution of requests, we define a predicate $conflict(a, b)$ which holds if there is an interdependency between two requests a and b. Specifically, two requests are in conflict with each other if their effects (i.e., changes to the application state) and outcomes (i.e., results) vary depending on the relative order in which they are executed by a replica. In addition, we define that $conflict(a, b)$ always holds for requests issued by the same client. Several previous works [5], [6], [8], [13], [14], [15] relied on similar predicates and concluded that for many applications determining request conflicts is straightforward. In key-value stores, for example, requests typically contain the key(s) of the data set(s) they access. Consequently, a write can be identified to conflict with another write or read to the same key. In contrast, two reads of the same data set are independent of each other due to not modifying application state and their results not being influenced by their relative execution order.

With our work presented in this paper we target use-case scenarios in which conflicting requests only constitute a small fraction of the application's overall workload (e.g., less than 5% [5]). In practice, this for example is the case for key-value stores with high read-to-write ratios [16] or coordination services for which the vast majority of requests access client-specific data structures (e.g., to renew session leases [17]).

III. Background & Problem Statement

Providing the agreement stage of a replicated system with information about request conflicts makes it possible to significantly increase consensus efficiency by limiting the ordering to requests that interfere with each other [10]. In this section, we analyze existing approaches that apply this general concept. Notice that (although tackling a related problem) our discussion does not include the recently proposed ezBFT [18], as since publication the protocol has been found to contain safety, liveness, and execution consistency violations [19].

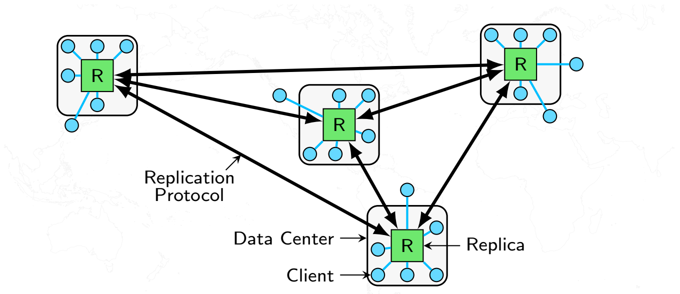

Figure 1. Geo-distributed state-machine replication

A. Existing Approaches

Based on their design goals and characteristics, existing protocols can be classified into the following three categories.

Crash Tolerance. One of the first leaderless consensus algorithms focusing on conflicting requests was EPaxos [5], which enables all replicas in the system to initiate the agreement process for new client requests. In geo-distributed deployments where clients are scattered across the globe (see Figure 1), this property often significantly improves latency as each client can directly submit requests to its local replica, instead of all clients having to contact the same central leader. If a quorum of replicas agrees on a proposed request's conflicts, EPaxos allows the proposing replica to immediately commit and process the request; otherwise, the replica is required to execute a sub-protocol responsible for resolving the conflict discrepancies. Building on the same general idea, the recently proposed Atlas [6] protocol offers several improvements over EPaxos, including for example the use of smaller quorums as well as the ability to commit requests early even if the conflict reports of different replicas do not match exactly (see Section VI for details). Both EPaxos and Atlas tolerate crashes.

BFT with Additional Replicas. The quorum-based Q/U [4] offers resilience against Byzantine faults without the need for a global leader, however to do so it requires $5f + 1$ replicas. Byblos [8], a BFT protocol tailored to permissioned ledgers, reduces the replication cost to $4f + 1$ servers by determining the execution order of transactions based on a leaderless non-skipping timestamp algorithm that is driven by clients.

Global Leader Replica. Byzantine Generalized Paxos [7] shows that it is possible for a BFT protocol to only order conflicting requests with a minimum of $3f+1$ replicas. However, to resolve request-conflict discrepancies between replicas the protocol resorts to a global leader which then sequentializes the affected requests. For this purpose, followers need to provide the leader with information about all requests they have previously voted for, making conflict resolution an expensive undertaking, as confirmed by our experiments in Section V.

B. Problem Statement

The analysis above has shown that existing approaches explore different tradeoffs with regard to fault model, replica-group size, and the existence of a global leader replica. In contrast, our goal in this paper is to integrate several desirable properties within the same state-machine replication protocol:

- **Byzantine Fault Tolerance:** The protocol should tolerate up to f replica faults as well as an unlimited number of faulty clients that possibly collude with faulty replicas.
- **Resource Efficiency:** To also support small deployments, the protocol must require a minimum of $3f+1$ replicas.
- **Leaderlessness:** To avoid a bottleneck and enable clients to submit requests to their nearest replica, the protocol must not rely on a single global leader replica. This should not only apply to normal-case operation, but also to the task of reconciling discrepancies between replicas.
- **Low Latency:** In the absence of discrepancies, the agreement process should complete within three communication steps, which is optimal for the targeted systems.
- **Bounded State:** To avoid an infinite accumulation of consensus state, in contrast to other leaderless protocols (e.g., EPaxos), the protocol should comprise a checkpointing mechanism for garbage-collecting such state. In addition, the protocol's execution stage should also be able to operate with a bounded amount of memory when determining the request execution order based on the conflict dependencies reported by the agreement stage.

In the following, we show that it is possible to unite these properties in a single state-machine replication protocol.

IV. Isos

Isos is a leaderless BFT protocol designed to minimize latency in wide-area settings. This section first gives an overview of Isos and then provides details on different protocol mechanisms; for pseudo code please refer to [11].

A. Overview

Isos requires a minimum of $N = 3f+1$ replicas to tolerate f faults and enables each of the replicas to order client requests without the involvement of a global leader. This allows clients to submit their requests to the nearest replica and thereby avoid lengthy detours. When a replica receives a request from a client, the replica acts as *request coordinator* and manages the replication of the request to all other replicas, which for this specific request serve as *followers*. That is, to prevent bottlenecks as well as disruptions due to costly election procedures, replica roles in Isos are not assigned globally as in many other BFT protocols [20], but instead on a per-request basis.

To order client requests as coordinator, each replica r_i maintains its own sequence of *agreement slots* which are uniquely identified by sequence numbers $s_i = \langle r_i, sc_i \rangle$, with sc_i representing a local counter. Apart from its own agreement slots, each replica also stores information about other replicas' agreement slots for which the local replica acts as follower.

Consensus Fast Path. Having received a new request, a coordinator allocates its next free agreement slot and creates a *dependency set* containing all conflicts the new request has to previous requests already known to the coordinator. As illustrated in Figure 2 for request A, the coordinator then initiates the consensus process by forwarding the request

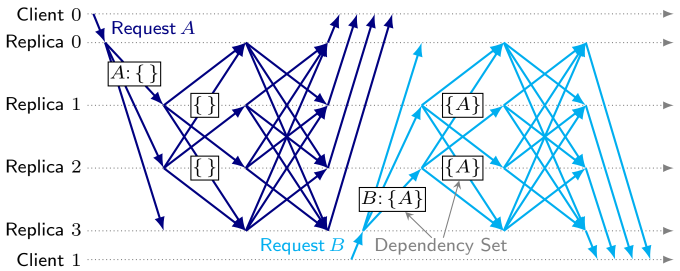

Figure 2. Fast-path ordering of two conflicting requests A and B in Isos

together with the dependency set to its followers. In a next step, a coordinator-selected quorum of $2f$ followers react by computing and broadcasting their own dependency set for the request. If all of these followers report the same dependencies as the coordinator, the consensus process completes at the end of another protocol phase, that is after three communication steps; we refer to this scenario as Isos's *fast path*. Notice that the fast path in Isos is not exclusive to non-conflicting requests, but as illustrated by the example of request B in Figure 2 can also be taken by conflicting client requests.

Reconciliation & View Change. If the coordinator determines that the fast-path quorum for a request is no longer possible, it triggers Isos's reconciliation mechanism which is responsible for resolving the request-conflict discrepancies between replicas by deciding on a consistent dependency set. In case of a faulty leader or faulty followers in the coordinator-selected quorum, the replicas initiate a view change for the affected agreement slot and continue to perform reconciliation.

Request Execution. Isos replicas rely on a deterministic algorithm to determine the execution order of requests based on the dependency sets they agreed on in the consensus process. Collecting dependency sets from a quorum of replicas ensures that conflicting requests, even when proposed by different coordinators at the same time, will pick up a dependency between them and thus guarantee a consistent execution order. For non-conflicting requests, there are no dependencies to consider, meaning that a replica is allowed to independently process such a request once it has been committed by the agreement stage. After executing a request, the replicas send a reply to the client which waits for $f+1$ matching replies to ensure that at least one of the replies originates from a correct replica.

Checkpointing. Isos relies on checkpointing to limit the amount of memory required by the agreement protocol and to allow replicas that have fallen behind to catch up. To create a consistent checkpoint, all replicas have to capture a copy of the application state after executing the exact same set of requests. As each replica can independently propose and execute requests, in contrast to traditional protocols such as PBFT [2], in Isos there are no predefined points in time (e.g., specific sequence numbers) at which all replicas have the same application state. To solve this problem, Isos introduces *checkpoint requests* which are agreed upon by the replicas and act as a barrier separating the requests that should be covered by a checkpoint from the ones that should not.

B. Fast Path

When a new request $r = \langle \textsc{Req}, x, t, o \rangle_{\sigma_x}$ for command o from client x arrives at a replica, the replica serves as coordinator for the request; t is a client-local timestamp that increases for each request and enables replicas to ignore duplicates.

DepPropose Phase. To start the fast path, the coordinator selects its agreement slot with the lowest unused sequence number and computes the dependency set containing sequence numbers of requests that conflict with request r. For this purpose, the coordinator takes all known requests from both its own and other replicas' agreement slots into account. Requests of the same client are automatically treated as conflicting with each other, independent of their content. This ensures that all correct replicas will later execute the requests of a client in the same order and therefore discard the same requests as duplicates. As a consequence, faulty clients cannot introduce inconsistencies between correct replicas by assigning the same timestamp to two non-conflicting requests. On correct clients, on the other hand, the client-specific request dependencies have no impact as correct clients commonly only submit a new request after having received a result for their previous one.

To limit the size of the set, the coordinator for each replica only includes the sequence number of the latest conflicting request, thereby treating the replica's earlier requests as implicit dependencies [5]. This approach potentially introduces (unnecessary) additional dependencies, however it offers two major benefits: (1) a compact dependency set in general is significantly smaller than a full set explicitly containing all conflicts would be, and (2) since correct replicas only accept and process compact dependency sets, a faulty replica cannot slow down the agreement process by distributing huge sets.

Having assembled the dependency set D for request r in agreement slot s_i, the coordinator co selects a quorum F containing the IDs of the $2f$ followers to which it has the lowest communication delay. As shown in Figure 3 (left), the coordinator then broadcasts a $\langle \textsc{DepPropose}, s_i, co, h(r), D, F \rangle_{\sigma_{co}}$ message together with the full request to all of its follower replicas; $h(r)$ is a hash that is computed over client request r.

DepVerify Phase. Follower replicas accept a DepPropose if the message originates from the proper coordinator and is accompanied by a client request with matching hash $h(r)$. A follower only sends a DepVerify in the next protocol phase if it is part of the quorum F. In such case, follower f_i calculates its own dependency set D_{f_i} for request r and broadcasts the set in a $\langle \textsc{DepVerify}, s_i, f_i, h(dp), D_{f_i} \rangle_{\sigma_{f_i}}$ message to all replicas, with dp referring to the corresponding DepPropose.

Followers strictly process the DepProposes of a coordinator in increasing order of their sequence numbers, thereby ensuring that a coordinator cannot skip any sequence numbers. Furthermore, they only compile and send the DepVerify for a DepPropose once they know that consensus processes have been initiated for all agreement slots listed in the DepPropose's dependency set. A follower has confirmation of the start of the consensus process if it fully processed a DepPropose, received $f + 1$ DepVerifys, or triggered a view change for a slot. Waiting for the conflicting slots to begin ensures that all dependencies in the dependency set will eventually complete agreement and thus guarantees that a faulty coordinator cannot block execution of a client request by including dependencies to non-existent requests.

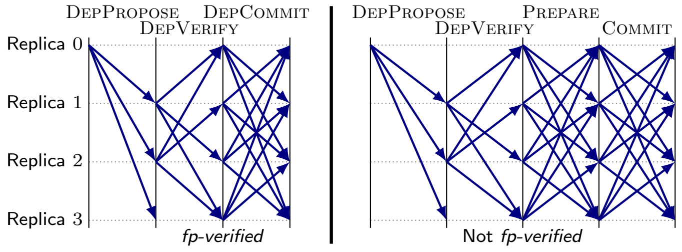

Figure 3. Fast path (left) and abandoned fast path + reconciliation (right)

DepCommit Phase. When a replica receives a DepVerify it checks that the included hash $h(dp)$ matches the slot's DepPropose and that the sender is part of the quorum F. As before for the DepPropose, the replica then waits until it knows that all agreement slots contained in the dependency set D_{f_i} will finish eventually. To continue with the fast path, a replica must complete the predicate *fp-verified*, which requires a valid DepPropose from the coordinator and matching DepVerifys from the $2f$ followers selected in the quorum F. The set of DepVerifys matches the DepPropose if either all DepVerifys have the same dependency set as in the DepPropose or if all additional dependencies are included in at least $f + 1$ DepVerifys. In the latter case, at least one correct replica has reported the additional dependencies, which ensures that these dependencies will be included in the fast path or the reconciliation path (see Section IV-C), independent of the behavior of faulty replicas. The DepPropose and DepVerifys yield a Byzantine majority quorum of $2f+1$ replicas, thereby guaranteeing that only a single proposal can complete, as correct replicas only accept the first valid DepPropose.

Once *fp-verified* holds, a replica broadcasts a corresponding $\langle \textsc{DepCommit}, s_i, r_i, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ message in which $\vec{dv}$ refers to the set of DepVerifys received from the followers in F. As each correct replica includes DepVerifys from the same followers, they all will use the same set $\vec{dv}$ to calculate $h(\vec{dv})$.

An agreement slot in Isos is *fp-committed* once a replica has obtained matching DepCommits from $2f + 1$ replicas (possibly including itself). At this point, the replica forwards the request to the execution (see Section IV-F), together with the union of the dependency sets of the DepPropose and DepVerifys. The quorum guarantees that if a request commits, then enough replicas have *fp-verified* it and consequently the request will be decided by (potential) later view changes.

C. Reconciliation Path

If a replica observes that completing *fp-verified* is not possible due to diverging dependency sets, the replica abandons the fast path and starts reconciliation (as illustrated on the right side of Figure 3). The main responsibility of Isos's reconciliation mechanism is to transform the diverging dependency sets from the fast path into a single dependency set that is agreed upon by all correct replicas. To ensure that fast path and reconciliation path cannot reach conflicting decisions regarding the dependency set, a correct replica that has reached *fp-verified* (and therefore already sent a DEPCOMMIT on the fast path) does not contribute to the reconciliation path.

Prepare Phase. Upon switching to the reconciliation path, a replica stops participating in the fast path and broadcasts a $\langle \text{PREPARE}, v_{s_i}, s_i, r_i, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ message in which $\vec{dv}$ is the set of previously received DEPVERIFYs; v_{s_i} denotes a view number, which in contrast to traditional BFT protocols [2] in ISOS is not global, but a variable specific to the individual agreement slot. That is, for each request that enters reconciliation the view number starts with its initial value of -1.

Commit Phase. After a replica has obtained $2f+1$ PREPARES matching the set of known DEPVERIFYs, the replica has *rp-prepared* the agreement slot and continues with broadcasting a $\langle \text{COMMIT}, v_{s_i}, s_i, r_i, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ message. Having collected $2f+1$ COMMITs from different replicas with matching hash $h(\vec{dv})$, the replica has *rp-committed* the request and forwards it to the execution, together with the union of the dependency sets of all DEPVERIFYs and the associated DEPPROPOSE.

Invariant. *An agreement slot in* ISOS *can either fp-commit or rp-prepare.* As sending a DEPCOMMIT and sending a PREPARE are mutually exclusive, correct replicas can either collect enough DEPCOMMITs from a quorum to *fp-commit* the fast path or enough PREPAREs to *rp-prepare* the reconciliation path, but never both, thus ensuring agreement among replicas.

D. View Change

In case the agreement for a slot fails to complete within a predefined amount of time (see Section IV-E), replicas in ISOS initiate a view change for the specific agreement slot affected.

ViewChange Phase. Once a replica decides to abort a view, the replica stops to process requests for the old view and broadcasts a $\langle \text{VIEWCHANGE}, v_{s_i}, s_i, r_i, \mathit{certificate} \rangle_{\sigma_{r_i}}$ message for the new view v_{s_i} to report the agreement-slot state in the form of a $\mathit{certificate}$ of one of the following types:

- A *fast-path certificate* (FPC) consists of a DEPPROPOSE message from the original coordinator and a set of $2f$ corresponding DEPVERIFY messages from different followers matching the DEPPROPOSE, thereby confirming that the agreement slot was *fp-verified*.
- A *reconciliation-path certificate* (RPC) consists of the original DEPPROPOSE, $2f$ matching DEPVERIFYs, and $2f+1$ matching PREPAREs from different followers. The PREPAREs must be from the same view. Together, these messages confirm the agreement slot to be *rp-prepared*.

If available, a replica includes an RPC for the highest view in its own VIEWCHANGE message, resorting to an FPC as alternative. If neither of the two certificates exists, the replica sends the VIEWCHANGE message without a certificate.

In case a replica receives $f+1$ VIEWCHANGEs for sequence number s_i with a view higher than its own, the replica switches to the $f+1$-highest view received for that agreement slot and broadcasts a corresponding VIEWCHANGE message.

NewView Phase. The view change for a request is managed by a coordinator that is specific to the request's agreement slot s_i. For a new view v_{s_i}, the coordinator is selected as $co = (s_i.r_i + max(0, v_{s_i})) \bmod N$. Having collected valid VIEWCHANGEs for its view from a quorum of $2f+1$ replicas, the coordinator determines the result of the view change. For this purpose, it deterministically selects a request based on the certificate with the highest priority: first RPC, then FPC.

If both a reconciliation-path certificate and a fast-path certificate exist at the same time, it is essential for the coordinator to determine the view-change result based on the reconciliation-path certificate. According to the reconciliation-path invariant, this path can only *rp-prepare* if the fast path does not *fp-commit*. Thus, the fast-path certificate stems from up to f replicas that tried to complete the DEPCOMMIT phase but did not finish it, meaning that the certificate can be ignored. The reconciliation path, on the other hand, might have completed and thus the view change must keep its result. If no certificate exists, the view-change result is a no-op request with empty dependencies, which later will be skipped during execution.

To install the new view, the coordinator broadcasts a $\langle \text{NEWVIEW}, v_{s_i}, s_i, co, dp, \vec{dv}, VCS \rangle_{\sigma_{co}}$ message in which dp is the DEPPROPOSE, $\vec{dv}$ are the accompanying DEPVERIFYs, and VCS is the set of $2f+1$ VIEWCHANGEs used to determine the result. If no certificate exists, dv is replaced by a no-op request and $\vec{dv}$ is empty. After having verified that the coordinator has correctly computed the NEWVIEW, the other replicas follow the coordinator into the new view. There, the NEWVIEW's DEPPROPOSE and DEPVERIFYs are used to resume with the reconciliation path at the corresponding step (see Section IV-C), just for a higher view. In case a request is replaced with a no-op during the view change, the request coordinator proposes the request for a new agreement slot.

E. Progress Guarantee

In the following, we discuss several liveness-related scenarios and explain how ISOS handles them to ensure that requests proposed by correct replicas eventually become executable.

Fast Path. Faulty replicas in ISOS may try to prevent correct replicas from making progress by not properly participating in the consensus process. For example, a faulty replica r_i may send a DEPPROPOSE for a sequence number s_i, but only to one correct replica r_j and not the others. Replica r_j thus must include sequence number s_i as dependency in its own future proposals, meaning that other replicas can only process r_j's proposals if they also know about s_i. To ensure that the system in such case eventually makes progress despite replica r_i's refusal to properly start the consensus for s_i, correct followers in ISOS start a *propose timer* with a timeout of 2Δ whenever they receive a DEPPROPOSE; Δ is the maximum one-way delay between replicas (see Section II). If the propose timer expires or a view change is triggered and the follower has not collected $2f$ matching DEPVERIFYs in the meantime, the follower broadcasts the affected DEPPROPOSE (which does not include the full client request, see Section IV-B) to all other follower replicas, thereby enabling them to move on.

Agreement. To monitor the agreement progress of a slot, replicas in ISOS start a *commit timer* with a timeout of 9Δ once they know that the consensus process for a slot has been initiated. This is the case if a replica has (1) sent its DEPPROPOSE, (2) (directly or indirectly) received a valid DEPPROPOSE and learned that its dependencies exist or (3) obtained $f+1$ DEPVERIFY messages proving that at least one correct replica has accepted a DEPPROPOSE for this slot. If the commit timer expires, a replica triggers a view change. Please refer to [11] for an explanation of the timeout value.

Forwarding the DEPPROPOSE after the propose timer expires (see above) and listening for DEPVERIFY messages ensures that every correct replica will eventually learn that a proposal for the agreement slot exists and thus start the commit timer. This in turn guarantees that either $f+1$ correct replicas commit a client request or trigger a view change.

Recovering the Fast-Path Quorum. If the quorum F proposed by a fast-path request coordinator includes faulty replicas, it is possible that these replicas do not send DEPVERIFY messages and thus prevent requests from being ordered in the agreement slot. In such case, after the agreement slot was completed with a no-op by a view change, the request coordinator selects a different set of $2f$ followers and proposes the request for a new agreement slot. This ensures that eventually all replicas in quorum F are correct which allows the agreement to complete.

Lagging Replicas. As the active involvement of $2f+1$ replicas is sufficient to commit a request in ISOS, there can be up to f correct but lagging replicas that do not directly learn the outcome of a completed agreement process. Furthermore, as the agreement processes of different coordinators advance largely independent of each other, different replicas may lag with respect to different coordinators. To resolve circular-waiting scenarios under such conditions, an ISOS replica can query others for committed requests. If $f+1$ replicas (i.e., at least one correct replica) report a request to have committed for an agreement slot, the lagging replica also regards the request as committed. Since $2f+1$ replicas are required to complete consensus, for each completed slot there are at least $f+1$ correct replicas that can assist lagging replicas in making progress.

Crashed Replicas. If a coordinator crashes, the effects of the crash are limited to the slots the coordinator has started prior to its failure. Once these slots have been completed (if necessary through view changes), there are no further impairments as the failed coordinator does no longer propose new requests, and thus there are no new dependencies on the coordinator.

F. Request Execution

Using committed requests and their dependency sets as input, the execution stage of a replica is responsible for determining the order in which the replica needs to process these requests. For correct replicas to remain consistent with each other, they all must execute conflicting requests in the same relative order. Non-conflicting requests on the other hand may be processed by different replicas at different points in time. In the following, we explain how ISOS ensures that these requirements are met even if faulty replicas manipulate dependencies.

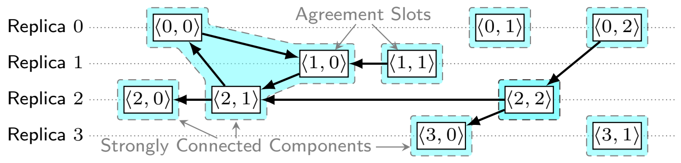

Figure 4. Strongly connected components in an execution dependency graph

Regular Request Execution. For each committed request, the execution builds a dependency graph whose nodes are not yet executed requests which are connected by directed edges as specified in the requests' dependency sets. This graph is constructed by recursively expanding the dependencies of the request. If a dependency refers to a not yet committed agreement slot, the graph expansion waits until the dependency is committed. The execution then calculates the strongly connected components in the dependency graph and executes them in inverse topological order. As illustrated in Figure 4, each strongly connected component represents either a single request or multiple requests connected by cyclic dependencies. The inverse topological order ensures that dependencies of all requests in a strongly connected component are executed first. For each such component, the requests are sorted and then executed according to their slot sequence number to ensure an identical execution order on all replicas. The execution uses the timestamp in a client request to filter out duplicates.

Handling Dependency Chains. As the dependency collection for a request is a two-step process (see Section IV-B), it is possible that DEPVERIFY messages include dependencies to agreement slots that were proposed after the request itself. These slots in turn can also collect dependencies to additional future slots resulting in a temporary execution livelock [5] that delays the execution of a request until all its dependencies are committed. Such dependency chains can either arise naturally when processing large amounts of conflicting requests [5] or due to faulty replicas manipulating dependency sets by including dependencies to future requests in their DEPVERIFYs.

To handle this kind of dependency chains with a bounded amount of memory, ISOS replicas limit how many requests are expanded. Specifically, for each coordinator the execution only processes a window of k agreement slots. The start of each window points to the oldest agreement slot of the coordinator with a not yet executed request. Dependencies to requests beyond this expansion limit are treated as missing and block the execution of a request. This bounds the effective size of dependency chains, while still allowing the out-of-order execution of non-conflicting requests within the window.

To unblock request execution, replicas use the following algorithm: First, a replica tries to normally execute all committed requests within the execution window. Then, for each coordinator the replica constructs the dependency graph of the oldest not yet executed request, called *root node*, and checks whether its execution is only blocked by missing requests beyond the execution limit. If this is the case, the replica ignores dependencies to the latter requests and starts execution.

However, it only processes the first strongly connected component and then switches back to regular request execution.

The intuition behind the algorithm is that the execution of root nodes occurs when only requests beyond the expansion limit are still missing (i.e., at a time when all replicas see the same dependency graph). For dependencies to other nodes ISOS's compact dependency representation (see Section IV-B) automatically includes a dependency on the root node for the associated coordinator, this ensures that dependent root nodes are executed in the same order on correct replicas.

G. Checkpointing

The checkpoints of correct replicas in BFT systems must cover the same requests in order to be safely verifiable by comparison [21]. Traditional BFT protocols [2], [22], [23] ensure this by requiring replicas to snapshot the application state in statically defined sequence-number intervals. In ISOS, this approach is not directly applicable because instead of one single global sequence of requests, there are multiple sequences (i.e., one per coordinator) that potentially advance at different speeds. To nevertheless guarantee consistent checkpoints, ISOS replicas rely on dedicated *checkpoint requests* to dynamically determine the points in time at which to create a snapshot. As illustrated in Figure 5, a checkpoint request conflicts with every other request and therefore acts as a barrier such that each regular client request on all correct replicas is either executed before or after the checkpoint request.

Basic Approach. A checkpoint request in ISOS is a special empty request that is known to all replicas and when processed by the execution triggers the creation of a checkpoint. Each correct replica is required to propose the checkpoint request for every own agreement slot with sequence number $sc_i \bmod cp\_interval = 0$; $cp\_interval$ is a configurable constant that also defines the minimum size of the agreement ordering window (i.e., $2 * cp\_interval$), that is the number of slots per coordinator for which a replica needs information.

Relying on a checkpoint request to determine when to create a snapshot in ISOS has the key benefit that replicas, as a by-product of the consensus process for this request, also automatically agree on the client requests the checkpoint must cover. Specifically, based on the checkpoint request's dependency set replicas know exactly which client requests they are required to execute prior to taking the application snapshot.

Having created the checkpoint, a replica broadcasts a $\langle \text{CHECKPOINT}, cp.seq, r_i, barrier, h(cp)\rangle_{\sigma_{r_i}}$ message to all other replicas; $cp.seq$ is a monotonically increasing checkpoint counter, $barrier$ refers to the requests included in the checkpoint (i.e., the dependency set plus the checkpoint request itself), and $h(cp)$ represents a hash of the checkpoint content.

Once a replica has collected $2f + 1$ matching checkpoint messages from different replicas, the messages form a checkpoint certificate that proves the stability of the checkpoint. After obtaining such a certificate, a replica can garbage-collect all earlier state covered by the checkpoint, including requests kept for conflict calculations. As a substitute, a replica from this point on uses $barrier$ as minimum dependency set.

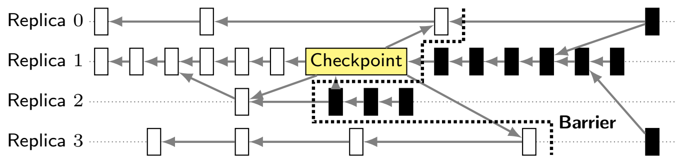

Figure 5. Checkpoint request serving as barrier for regular client requests

Checkpoint-specific View Change. While regular agreement slots may eventually result in a no-op being committed (see Section IV-D), ISOS guarantees that the proposal of a checkpoint request will eventually succeed within its original checkpoint slot. Our solution to achieve this relies on an auxiliary DEPVERIFY that a replica additionally includes in its VIEWCHANGE when starting a view change for a checkpoint slot. The auxiliary DEPVERIFY contains a placeholder hash as well as a dependency set for the checkpoint request. If the replica has previously participated in the fast path, the dependency set is identical with the one from the replica's own DEPPROPOSE or DEPVERIFY, otherwise the replica computes a new dependency set for the checkpoint request. Notice that due to the fact that the content and sequence numbers of checkpoint requests are known in advance, a replica is able to create such an auxiliary DEPVERIFY even if it has not received the actual DEPPROPOSE for the checkpoint slot.

Utilizing the auxiliary DEPVERIFYs, we are able to extend the certificate list of Section IV-D with a third option: a *checkpoint request certificate (CRC)* that is selected if neither of the two other certificates is available. The CRC consists of $2f + 1$ auxiliary DEPVERIFYs verified to only include known dependencies and can be used in the new view to agree on a common dependency set. Since for checkpoint slots, the CRC is always available as a fallback, there is no need for a view-change coordinator to introduce a no-op request.

Checkpoint-specific Execution. Being generally treated like regular client requests, a checkpoint request can be part of a dependency cycle in which some requests should be processed before the checkpoint, while others are to be executed afterwards. To handle such a scenario, ISOS's execution processes strongly connected components in a special way if they contain checkpoints. First, it merges the dependency sets of all checkpoint requests included in a strongly connected component, adding the checkpoint requests themselves to the merged set. Next, the merged set is bounded to not exceed the expansion limit described in Section IV-F, and to include all requests before the first not yet executed request of each replica. The resulting set now acts as a barrier defining which requests should be covered by the checkpoint and which should not. In the final step, the execution uses the barrier to only execute client requests before the barrier, followed by the merged checkpoint request. For the remaining requests after the barrier, a new dependency graph is constructed and used to order requests. Restarting the execution algorithm for these requests ensures that they are executed the same way as a lagging replica would do if it applied the checkpoint to catch up.

H. Correctness (Proof sketch; full proof is available in [11])

Safety. *All correct replicas that commit a slot must decide on the same request and dependencies.* A correct replica can only commit on the fast or reconciliation path if it has collected a quorum of DepVerifys or Prepares, which ensures that all replicas agree on the same request. As shown in Section IV-C, committing the fast or reconciliation path is mutually exclusive, meaning that within a view all replicas arrive at the same result. The final dependencies for a slot are defined by the DepPropose and the set $\vec{dv}$ of $2f$ DepVerifys whose hash $h(\vec{dv})$ is included in the DepCommit and Commit messages, respectively. This ensures that all replicas agree on the dependencies. After a successful commit, at least $f + 1$ correct replicas have collected a certificate for the fast or reconciliation path, and thus the certificate will be included in future view changes.

Execution Consistency (as used in EPaxos [5])***.*** *If two conflicting requests A and B are committed, all replicas will execute them in the same order.* This is achieved by ensuring that the two requests are connected by a dependency such that either A depends on B, or B depends on A, or both depend on each other. All three cases result in the execution consistently ordering the requests before processing them. The dependencies for a request are collected from a quorum of $2f + 1$ replicas using DepPropose and DepVerify messages. If requests A and B are proposed by different replicas at the same time, their dependency collection quorums will overlap in at least $f + 1$ replicas, of which at least one replica must be correct. This replica will either receive A or B first and thus add a dependency between them. Therefore, two conflicting requests are always connected by a dependency.

Note that a malicious coordinator proposing different DepProposes to its followers cannot cause missing dependencies. Either the same DepPropose is fully processed by at least $f+1$ correct replicas (which ensures dependency correctness), the faulty DepProposes are ignored, or none of the DepProposes gathers $2f$ DepVerifys, thus causing the slot to be filled with a no-op during the following view change. In the latter case, no dependencies from or to the slot are necessary, as a no-op command does not conflict with any other request.

The dependency cannot be lost when switching between protocol paths or during a view change. The reconciliation path carries over the dependency sets from the fast path and cannot introduce new dependencies in the agreement process. Replicas that learn about a client request in a view change have no influence on the dependency calculations for the request.

Invariant. *The view change either selects the (only) request that was fp-verified or rp-prepared, or a no-op.* We proof this by induction. Only a single DepPropose can collect $2f$ matching DepVerifys in a slot. Thus, no fast or reconciliation path certificate can exist for any other request, as constructing a certificate requires a matching set of DepVerifys from a quorum of replicas. The view change only selects a request with a certificate or a no-op, and hence all future reconciliation-path executions can only decide one of the two. This guarantees that a slot either commits the request initially sent to a majority of replicas or a (by definition) non-conflicting no-op. Requests that were not properly proposed to a quorum of replicas will therefore be replaced with a no-op. This ensures that all ordered requests have proper dependency sets.

V. Evaluation

In this section, we experimentally evaluate Isos together with other protocols in a geo-replicated setting. For a fair comparison, we focus on BFT protocols and implement them in a single codebase written in Java: (1) **PBFT** [2] represents a protocol that pursues the traditional concept of relying on a central global leader replica to manage consensus. (2) **CSP**, short for Centralized Slow Path, refers to a hybrid approach which, similar to Byzantine Generalized Paxos (BGP) [7], combines a leaderless fast path with a leader-based slow path for conflict resolution. We decided to create CSP because BGP requires its leader replica to share large sets of previously ordered requests to resolve conflicts, which in practical use-case scenarios results in unacceptable overhead. Since CSP's slow path does not suffer from this problem, we expect CSP's results to represent a best-case approximation of BGP's performance. (3) **Isos** in contrast to the other two protocols is entirely leaderless, in both the fast path as well as during reconciliation.

We conduct our experiments hosting the replicas in virtual machines (t3.small, 2 VCPUs, 2GB RAM, Ubuntu 18.04.5 LTS, OpenJDK 11) in the Amazon EC2 regions in Oregon, Ireland, Mumbai, and Sydney. Our clients run in a separate virtual machine in each region. CSP's slow-path leader resides in Oregon. All messages exchanged between replicas are signed with 1024-bit RSA signatures. As the communication times between replicas vary between 59 and 127 ms, we set Δ to 200 ms. Replicas use $cp\_interval$ = 2,000 to create new checkpoints and an expansion limit of 20 for the request execution. Each coordinator accumulates new client requests in batches of up to 5 requests before proposing them for ordering.

As application for our benchmarks, we use a key-value store for which clients issue read and write requests in a closed loop. Write requests modifying the same key conflict with each other. In contrast, read requests for a key only conflict with write requests but not with other read requests.

A. Latency

In our first experiment, we use a micro benchmark (200 bytes request payload, 10 clients per region) to compare the response times experienced by clients in the three systems. To control the rate of requests that can conflict with each other, we follow the setup of EPaxos [5] and ATLAS [6] and let clients issue write requests for a fixed key with a probability p, and for a unique key otherwise. We use conflict rates of 0%, 2%, and 5% to evaluate typical application scenarios of Isos; for comparison, EPaxos considers low conflict rates between 0% and 2% as most realistic [5]. In addition, to present the full picture we repeat our experiment with conflict rates of 10% and 100% for completeness. For PBFT, which is not affected by the conflict rate, we instead measure the latency for each possible

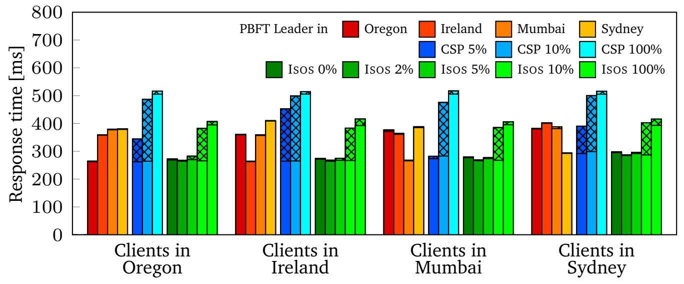

Figure 6. 50th (□) and 90th (▨) percentiles of response times for clients at different geographic locations, issuing requests with various conflict rates.

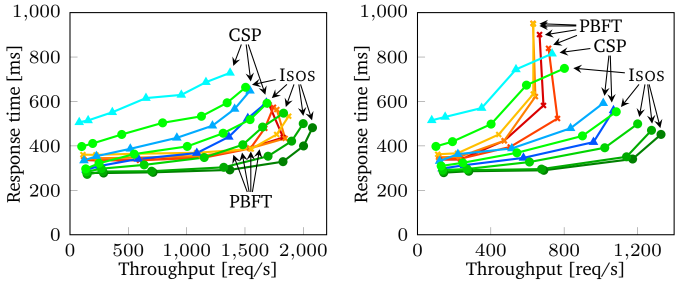

Figure 7. Relation between average throughput and response time for client requests with different payload sizes of 200 bytes (left) and 16 kB (right).

leader location. The results of this experiment are presented in Figure 6. For clarity, we omit the CSP numbers for low conflict rates of 0% and 2% as they are dominated by the fast path and thus similar to the corresponding results of Isos.

In PBFT, the median response times for clients in a region heavily depend on the current location of the leader replica. For clients in Ireland, for example, the response times can increase by up to 56% when the leader replica is not located in Ireland but in a different region. This puts all clients at a disadvantage whose location differs from that of the leader. In contrast, for typical low conflict rates of 2%, Isos in each region achieves median and 90th percentile response times similar to those of the best PBFT configuration for that region. However, PBFT due to its reliance on a single leader replica can only provide optimal response times for a single region at a time, whereas Isos's leaderless design enables clients to submit their requests to a nearby replica and thus provides optimal response times for clients in all regions at once.

For conflict rates of 5% and higher, the median and 90th percentile response times for CSP rise up to 517 ms, which is a result of the additional communication step required by the central leader to initiate the agreement on conflicting dependencies. For comparison, the response times of Isos are significantly lower even for a conflict rate of 100% where most requests are ordered via the reconciliation path. This illustrates the benefits of Isos's design choice to refrain from a global leader, not only on the fast path but also during reconciliation.

B. *Throughput*

In our second experiment, we assess the relation between throughput and response times for up to 1,000 evenly distributed clients and different request sizes (see Figure 7). For PBFT and requests with 200 bytes payload, the average response time stays below 369 ms for up to 400 clients and starts to rise afterwards. The throughput reaches nearly 1,875 requests per second at which point it is limited by the leader replica saturating its CPU. For low conflict rates of 0% and 2% Isos, on the other hand, achieves response times below 304 ms for up to 400 clients and reaches a throughput of up to 2,079 requests per second. This represents an improvement of 18% lower latency and 11% higher throughput over PBFT, showing the benefit of clients being able to submit their requests to a nearby replica instead of forwarding it to a central leader. Comparing CSP and Isos for conflict rates above 5% shows that CSP provides higher response times and thus lower throughput than Isos, which is a consequence of the additional communication step necessary to initiate CSP's slow path.

Issuing large requests with 16 kB payload from up to 600 clients, we observe that PBFT reaches a maximum throughput between 632 and 764 requests per second depending on the leader location. At this point, the network connection of the leader, which has to distribute the requests to all other replicas, is saturated and prevents further throughput increases. In contrast, Isos reaches a maximum throughput of 1,328 requests per second, outperforming PBFT by up to 110%. The throughput advantage even holds for conflict rates as high as 10%. Isos benefits from its leaderless design in which all replicas share the load of distributing requests, allowing it to handle larger requests than a protocol using a single leader.

C. *YCSB*

In our third experiment, we run the YCSB benchmark [24] with a total of 200 clients that are evenly distributed across all regions and issue a mix of reads and writes. The database is loaded with 1,000 entries of 1kB size. The key accessed by a client request is selected according to the Zipfian distribution which skews access towards a few frequently accessed elements and is parameterized using the standard YCSB settings.

Figure 8 shows the throughput achieved for different shares of read and write requests. For the write heavy 50/50 benchmark, Isos and PBFT achieve similar average throughputs of nearly 600 requests per second. Consistent with the previous benchmarks, the throughput of CSP stays below that of Isos. For the 95/5 and 100/0 workloads, Isos outperforms PBFT by 17% and 20%, respectively. Due to a high fraction of read requests, these workloads have a low conflict rate, thereby allowing Isos to take full advantage of its leaderless design.

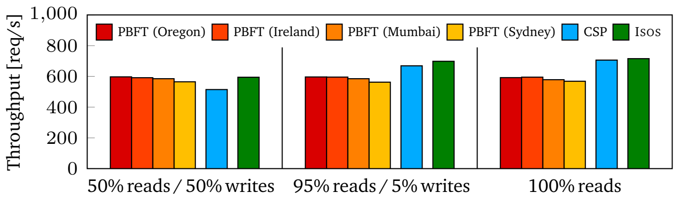

Figure 8. Average throughput for different read-write ratios in YCSB.

VI. Related Work

Optimized Leader Placement. One method to reduce the response time for systems with a central leader replica is to optimize its placement. Archer [25] uses clients to send probes through the agreement protocol to measure latency and thus enable the system to select a leader offering low latency. In AWARE [26], replicas measure the communication latency between themselves and use the outcome to adjust replica voting weights to prefer the fastest replicas. In ISOS, these approaches could be used to select optimal fast-path quorums.

Concurrent Consensus. To distribute the work of a leader, it is possible to partition a global sequence number space onto multiple leader replicas. Protocols like BFT-Mencius [27], Mir-BFT [28], Omada [23] and RCC [29] then run multiple ordering instances in parallel and merge them according to their sequence numbers. In comparison to ISOS these protocols primarily focus on throughput and either have to wait for ordered requests from all replicas or require additional coordination to handle imbalanced workloads.

Leaderless Consensus. DBFT [30] avoids using a central leader by letting replicas distribute their proposals using a reliable Byzantine broadcast and then reaching agreement on which replicas contributed proposals. This requires at least four communication steps compared to the three of ISOS's fast path, resulting in higher latency. The eventually consistent PnyxDB [31] uses conditional endorsements based on conflicts between requests. An endorsement for a request becomes invalid if a conflicting request could be committed before the request, causing some requests to be dropped eventually.

Crash Faults. PePaxos [32] is a recent variant of EPaxos [5] which during execution uses the agreement's dependency sets to schedule independent strongly-connected components for parallel execution. This approach can also be integrated in ISOS. Atlas [6] uses a fast path based on a preselected quorum of replicas, allowing it to optimize the reconciliation of differing dependency sets. Dependencies for an agreement slot proposed by at least f replicas can be agreed on via the fast path, allowing Atlas to always take the fast path for $f = 1$. ISOS uses a similar optimization for its fast path requiring $f+1$ replicas to report dependencies to handle Byzantine faults.

VII. Conclusion

ISOS is a fully leaderless BFT protocol for geo-replicated environments. It requires only $3f+1$ replicas and offers a fast path that orders client requests in three communication steps if request conflicts are reported by at least $f+1$ replicas.

Acknowledgments: This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 ("REFIT").

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. of OSDI '99*, 1999, pp. 173–186.

[3] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proc. of OSDI '08*, 2008.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proc. of SOSP '05*, 2005, pp. 59–74.

[5] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of SOSP '13*, 2013, pp. 358–372.

[6] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. Sutra, "State-machine replication for planet-scale systems," in *Proc. of EuroSys '20*, 2020.

[7] M. Pires, S. Ravi, and R. Rodrigues, "Generalized Paxos made Byzantine (and less complex)," *Algorithms*, vol. 11, no. 9, 2018.

[8] R. Bazzi and M. Herlihy, "Clairvoyant state machine replication," *Information and Computation*, 2021.

[9] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 3, p. 463–492, 1990.

[10] F. Pedone and A. Schiper, "Generic broadcast," in *Proc. of DICS '99*, 1999, pp. 94–106.

[11] M. Eischer and T. Distler, "Egalitarian Byzantine fault tolerance (extended version)," 2021. [Online]. Available: http://arxiv.org/abs/2109.06811

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[13] R. Kotla and M. Dahlin, "High throughput Byzantine fault tolerance," in *Proc. of DSN '04*, 2004, pp. 575–584.

[14] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency," in *Proc. of EuroSys '11*, 2011, pp. 91–105.

[15] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "SAREK: Optimistic parallel ordering in Byzantine fault tolerance," in *Proc. of EDCC '16*, 2016, pp. 77–88.

[16] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing Facebook's memcached workload," *IEEE Internet Computing*, vol. 18, no. 2, pp. 41–49, 2013.

[17] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. of OSDI '06*, 2006, pp. 335–350.

[18] B. Arun, S. Peluso, and B. Ravindran, "ezBFT: Decentralizing Byzantine fault-tolerant state machine replication," in *Proc. of ICDCS '19*, 2019.

[19] N. Shrestha and M. Kumar, "Revisiting ezBFT: A decentralized Byzantine fault tolerant protocol with speculation," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1909.03990

[20] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Computing Surveys*, vol. 54, no. 1, 2021.

[21] M. Eischer, M. Büttner, and T. Distler, "Deterministic fuzzy checkpoints," in *Proc. of SRDS '19*, 2019.

[22] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient Byzantine fault tolerance," *IEEE Trans. on Computers*, vol. 65, no. 9, pp. 2807–2819.

[23] M. Eischer and T. Distler, "Scalable Byzantine fault-tolerant state-machine replication on heterogeneous servers," *Computing*, vol. 101, no. 2, pp. 97–118, 2019.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. of SoCC '10*, 2010, p. 143–154.

[25] M. Eischer and T. Distler, "Latency-aware leader selection for geo-replicated Byzantine fault-tolerant systems," in *Proc. of BCRB '18*, 2018.

[26] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani, "Resilient wide-area Byzantine consensus using adaptive weighted replication," in *Proc. of SRDS '19*, 2019.

[27] Z. Milosevic, M. Biely, and A. Schiper, "Bounded delay in Byzantine-tolerant state machine replication," in *Proc. of SRDS '13*, 2013.

[28] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolić, "Mir-BFT: High-throughput robust BFT for decentralized networks," *CoRR*, 2021. [Online]. Available: http://arxiv.org/abs/1906.05552

[29] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: Resilient concurrent consensus for high-throughput secure transaction processing," in *Proc. of ICDE '21*, 2021.

[30] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "DBFT: Efficient leaderless Byzantine consensus and its application to blockchains," in *Proc. of NCA '18*, 2018.

[31] L. Bonniot, C. Neumann, and F. Taïani, "PnyxDB: a lightweight leaderless democratic byzantine fault tolerant replicated datastore," in *Proc. of SRDS '20*, 2020, pp. 155–164.

[32] T. Ceolin, F. Dotti, and F. Pedone, "Parallel state machine replication from generalized consensus," in *Proc. of SRDS '20*, 2020, pp. 133–142.