Egalitarian Byzantine Fault Tolerance (Extended Version)

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

arXiv:2109.06811v1 [cs.DC] 14 Sep 2021

***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 Appendix A. (5) It experimentally evaluates ISOS with a key-value store in a geo-distributed setting deployed in Amazon's EC2 cloud.

This is an extended version of the article "Michael Eischer and Tobias Distler. 2021. Egalitarian Byzantine Fault Tolerance. In *Proceedings of the 26th Pacific Rim International Symposium on Dependable Computing (PRDC '21), Perth, Australia, 1–4 December 2021.*"

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 [11], 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], [12], [13], [14] 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 [15] or coordination services for which the vast majority of requests access client-specific data structures (e.g., to renew session leases [16]).

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 [17], as since publication the protocol has been found to contain safety, liveness, and execution consistency violations [18].

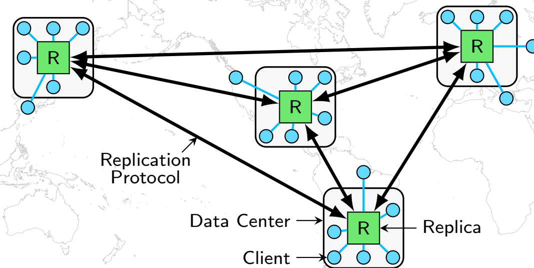

Figure 1. Geo-distributed state-machine replication

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 refer to Appendix A.

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 [19], 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 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.

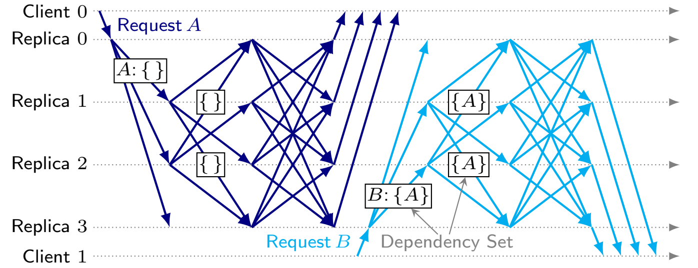

Figure 2. Fast-path ordering of two conflicting requests A and B in Isos

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 $r = \langle \text{REQ}, x, t, o \rangle_{\sigma_x}$ 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 $\langle \text{DEPPROPOSE}, s_i, co, h(r), D, F \rangle_{\sigma_{co}}$ 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 $\langle \text{DEPVERIFY}, s_i, f_i, h(dp), D_{f_i} \rangle_{\sigma_{f_i}}$ 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.

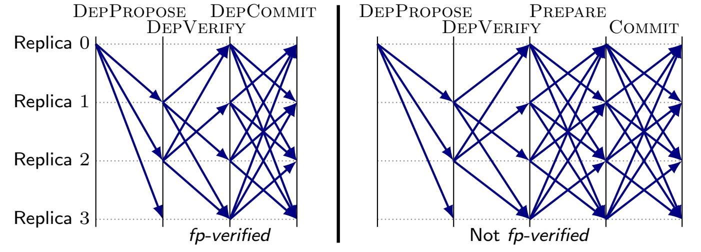

Figure 3. Fast path (left) and abandoned fast path + reconciliation (right)

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 $\langle \text{DEPCOMMIT}, s_i, r_i, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ 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 $\langle \text{VIEWCHANGE}, v_{s_i}, s_i, r_i, certificate \rangle_{\sigma_{r_i}}$ message for the new view v_{s_i} to report the agreement-slot state in the form of a $certificate$ 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. The value of the commit timeout is explained in Appendix A.

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.

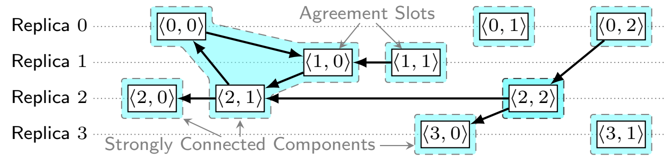

Figure 4. Strongly connected components in an execution dependency graph

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 [20]. Traditional BFT protocols [2], [21], [22] 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.

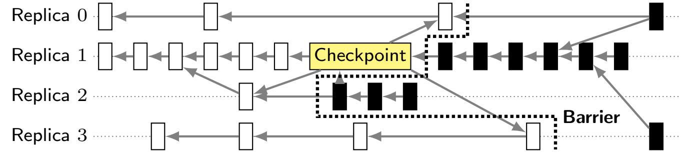

Figure 5. Checkpoint request serving as barrier for regular client requests

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 in Appendix A)

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

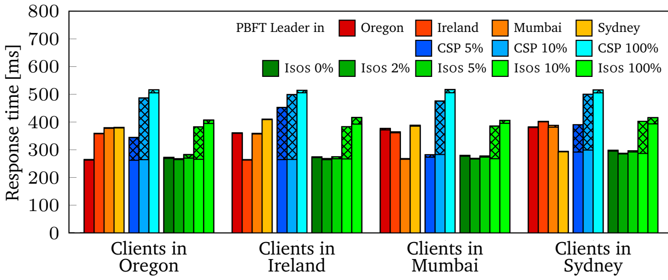

Figure 6. 50th (□) and 90th (⊠) percentiles of response times for clients at different geographic locations, issuing requests with various conflict rates.

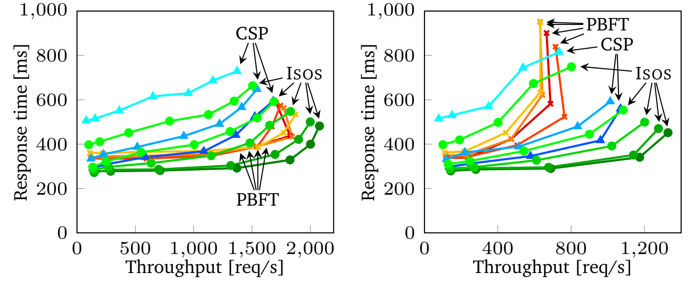

Figure 7. Relation between average throughput and response time for client requests with different payload sizes of 200 bytes (left) and 16 kB (right).

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 [23] 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.

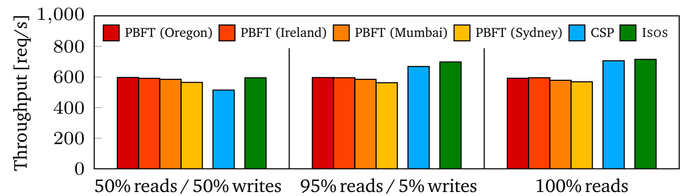

Figure 8. Average throughput for different read-write ratios in YCSB.

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 [24] 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 [25], 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 [26], Mir-BFT [27], Omada [22] and RCC [28] 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 [29] 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 [30] 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 [31] 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. 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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

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

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

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

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

[23] 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.

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

[25] 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.

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

[27] 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

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

[29] 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.

[30] 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.

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

APPENDIX

A. ISOS *Agreement Protocol*

1 **Variables at each replica**:
2 $p[s_j]$ /* DEPPROPOSE *for agreement slot* s_j *includes fast–path quorum* F */
3 $pr[s_j]$ /* REQ *for* DEPPROPOSE *of agreement slot* s_j */
4 $v[s_j][f_i]$ /* DEPVERIFY *for slot* s_j *from follower* f_i */
5 $step[s_j] \in \{$`init`,`proposed`,`fp−verified`,`fp−committed`, `rp−verified`,`rp−prepared`,`rp−committed`,`view−change`$\}$
6 $view[s_j]$ /* *View number for slot* s_j*, initially* $view[s_j] := -1$ */
7 $views[s_j][r_i]$ /* *Highest view number for slot* s_j *seen for replica* r_i */
8 $cert[s_j]$ /* *Latest own certificate for slot* s_j */
9 $\Delta_{propose} := 2\Delta;\ \Delta_{commit} := 9\Delta;\ \Delta_{vc} := 3\Delta;$ $\Delta_{vc-commit} := 3\Delta;\ \Delta_{query-exec} := 4\Delta$

Fast Path

10 **Request coordinator** co **receives new** $r := \langle \text{REQ}, x, t, o\rangle$:
11 `assert` r `correctly signed`
12 $s_j := \langle co, sc_j\rangle$ /* *Smallest free slot* */
13 $D :=$ `conflicts`(r)
14 $F :=$ Quorum of $2f$ followers
15 $dp := \langle\langle \text{DEPPROPOSE}, s_j, co, h(r), D, F\rangle_{\sigma_{co}}, r\rangle$
16 $\langle p[s_j], pr[s_j]\rangle := dp$
17 $step[s_j] :=$ `proposed`
18 Broadcast d to all replicas
19 Start commit timeout Δ_{commit} for slot s_j
20 **Follower** f_i **receives** $dp := \langle\langle \text{DEPPROPOSE}, s_j, co, h(r), D, F\rangle, r\rangle$:
21 **`pre:`** $step[s_j] =$ `init`
22 `assert` F `is a valid fast-path quorum`
23 `assert` $pr[s_j] = \emptyset \wedge s_j.co = co$ /* *First propose from coordinator* */
24 `wait`$(D \cup s_{j-1})$ /* s_{j-1} *is the previous slot from coordinator* co */
25 If $p[s_j] = \emptyset$:
26 Start commit timeout Δ_{commit} for slot s_j
27 Start propose timeout $\Delta_{propose}$ for slot s_j
28 $p[s_j] := dp.$DEPPROPOSE
29 If $r \neq \bot$:
30 `assert` r `correctly signed`
31 $D_{f_i} :=$ `conflicts`(r)
32 $pr[s_j] := r$
33 $step[s_j] :=$ `proposed`
34 If $f_i \in F$:
35 Broadcast $\langle \text{DEPVERIFY}, s_j, f_i, h(dp), D_{f_i}\rangle_{\sigma_{f_i}}$
36 **Replica** r_i **receives** $m := \langle \text{DEPVERIFY}, s_j, f_i, h(dp), D_{f_i}\rangle$:
37 **`pre:`** $step[s_j] =$ `proposed` $\wedge\ h(p[s_j]) = h(dp)$
38 `assert` $v[s_j][f_i] = \emptyset$ /* *First verify from follower* */
39 `assert` $f_i \in m[s_j].F$ /* *Follower is in fast–path quorum* */
40 `wait`(D_{f_i})
41 $v[s_j][f_i] := m$
42 $\vec{dv} := \{v[s_j][f_i] \mid \forall f_i \in p[s_j].F\}$
43 If $|\vec{dv}| = 2f$:
44 Stop propose timeout $\Delta_{propose}$ for slot s_j
45 $D := \cup D_{f_i} \in \vec{dv}$
46 /* *Every dependency is reported by at least* $f+1$ *followers* */
47 If $\{d \in D \mid |\{f_i \mid \forall f_i : d \in v[s_j][f_i].D\}| \geq f+1\} = D$:
48 /* *Slot* s_j *is now fp–verified at replica* r_i */
49 $step[s_j] :=$ `fp−verified`
50 Broadcast $\langle \text{DEPCOMMIT}, s_j, r_i, h(\vec{dv})\rangle_{\sigma_{r_i}}$
51 Else:
52 Enter reconciliation path, stop participating in fast path
53 **Replica** r_i **receives** $\langle \text{DEPVERIFY}, s_j, *, *, *\rangle$ **from** $f+1$ **replicas**:
54 Start commit timeout Δ_{commit} for slot s_j
55 **Replica** r_i **receives** $\langle \text{DEPCOMMIT}, s_j, *, h(\vec{dv})\rangle$ **with identical** $h(\vec{dv})$ **from** $2f+1$ **replicas**:
56 **`pre:`** $step[s_j] =$ `fp−verified` $\wedge\ h(\{v[s_j][f_i] \mid \forall f_i\}) = h(\vec{dv})$
57 Stop propose/commit timeout $\Delta_{propose}$ and Δ_{commit} for slot s_j
58 $exec[s_j] := \langle pr[s_j], \cup D_{f_i} \in \vec{dv}\rangle$
59 Forward $\langle pr[s_j], D, s_j\rangle$ to execution, with $D := \cup D_{f_i} \in \vec{dv}$
60 **`wait`**(D):
61 For $d \in D$:
62 Wait until either
63 $p[d] \neq \emptyset$ /* *received a valid* DEPPROPOSE */
64 received $f+1$ correctly signed DEPVERIFY
65 received $f+1$ correctly signed VIEWCHANGE
66 **`conflicts`**(r):
67 Return $\{s_i \mid \forall s_i, pr[s_i] \neq \emptyset : conflict(pr[s_i], r)\}$

Reconciliation Path

68 **Timeout** $\Delta_{propose}$ **for slot** s_j **expires:**
69 Broadcast $\langle p[s_j], \bot\rangle$ to all replicas
70 **Timeout** Δ_{commit} **for slot** s_j **expires:**
71 Move to new view $v_{s_j} + 1$
72 **Enter reconciliation path for slot** s_j **at replica** r_i:
73 $step[s_j] :=$ `rp−verified`
74 $\vec{dv} := \{v[s_j][f_i] \mid \forall f_i\}$
75 Broadcast $\langle \text{PREPARE}, view[s_j], s_j, r_i, h(\vec{dv})\rangle_{\sigma_{r_i}}$
76 **Replica** r_i **receives** $\langle \text{PREPARE}, v_{s_j}, s_j, *, h(\vec{dv})\rangle$ **with identical** $h(\vec{dv})$ **from** $2f+1$ **replicas**:
77 **`pre:`** $step[s_j] =$ `rp−verified` $\wedge\ view[s_j] = v_{s_j}$ $\wedge\ h(\{v[s_j][f_i] \mid \forall f_i\}) = h(\vec{dv})$
78 $step[s_j] :=$ `rp−prepared`
79 Broadcast $\langle \text{COMMIT}, v_{s_j}, s_j, r_i, h(\vec{dv})\rangle_{\sigma_{r_i}}$
80 **Replica** r_i **receives** $\langle \text{COMMIT}, v_{s_j}, s_j, *, h(\vec{dv})\rangle$ **with identical** $h(\vec{dv})$ **from** $2f+1$ **replicas**:
81 **`pre:`** $step[s_j] =$ `rp−prepared` $\wedge\ view[s_j] = v_{s_j}$ $\wedge\ h(\{v[s_j][f_i] \mid \forall f_i\}) = h(\vec{dv})$
82 $step[s_j] :=$ `rp−committed`
83 Stop commit timeout Δ_{commit} for slot s_j
84 $exec[s_j] := \langle pr[s_j], \cup D_{f_i} \in \vec{dv}\rangle$
85 Forward $\langle pr[s_j], D, s_j\rangle$ to execution, with $D := \cup D_{f_i} \in \vec{dv}$

View Change

86 **Move to new view** v_{s_j} **for slot** s_j **at replica** r_i:
87 If propose timeout $\Delta_{propose}$ for slot s_j is active, trigger its expiry
88 Stop commit/vc timeout Δ_{commit} and Δ_{vc} for slot s_j
89 $dp := \langle p[s_j], pr[s_j]\rangle;\ \vec{dv} := \{v[s_j][f_i] \mid \forall f_i\}$
90 /* *Update certificate if current view fp–verified / rp–prepared* */
91 If $step[s_j] \in \{$`fp−verified, fp−committed`$\}$:
92 $cert[s_j] := \langle \text{FPC}, dp, \vec{dv}, -1\rangle$
93 Else If $step[s_j] \in \{$`rp−prepared, rp−committed`$\}$:
94 $\vec{prep} :=$ set of $2f+1$ PREPARES with $h(\vec{dv})$
95 $cert[s_j] := \langle \text{RPC}, dp, \vec{dv}, \vec{prep}, view[s_j]\rangle$
96 $view[s_j] := v_{s_j}$
97 $views[s_j][r_i] := v_{s_j}$
98 $step[s_j] :=$ `view−change`
99 Start query execute timeout $\Delta_{query-exec}$ for slot s_j
100 Broadcast $\langle \text{VIEWCHANGE}, v_{s_j}, s_j, r_i, cert[s_j]\rangle_{\sigma_{r_i}}$
101 **Replica** r_i **receives** $\langle \text{VIEWCHANGE}, v_{s_j}, s_j, r_k, *\rangle$:
102 **`pre:`** $v_{s_j} > views[s_j][r_k]$ /* *View of a replica must only increase* */
103 $views[s_j][r_k] := v_{s_j}$
104 /* *Move to f+1–highest known view* */
105 $vn := f+1$-highest in $\{views[s_j][r_l] \mid \forall r_l\}$
106 If $vn > view[r_i]$:
107 Move to new view vn /* *Sends new* VIEWCHANGE *message* */
108 $co := (s_j.co + \texttt{max}(0, v_{s_j}))\ \texttt{mod}\ N$

109 **View-change coordinator** co **for view** v_{s_j} **receives valid** $VCS := \{\langle \text{VIEWCHANGE}, v_{s_j}, s_j, *, * \rangle\}$ **from** $2f+1$ **replicas**:
110 `pre:` $step[s_j] =$ `view-change` $\wedge$ $view[s_j] = v_{s_j}$
111 `assert` $\forall VC \in VCS : VC$ is valid $\wedge (VC.cert = \emptyset \vee VC.cert.view \leq VC.v_{s_j})$
112 Pick dp, $\vec{dv}$ from
113 reconciliation-path result for highest view if RPC certificate exists
114 fast-path result if FPC certificate exists
115 `null` otherwise
116 Broadcast $\langle \text{NEWVIEW}, v_{s_j}, s_j, co, dp, \vec{dv}, VCS \rangle_{\sigma_{co}}$
117 **Replica** r_i **receives valid** $VCS := \{\langle \text{VIEWCHANGE}, v_{s_j}, s_j, *, * \rangle\}$ **from** $2f+1$ **replicas**:
118 `pre:` $step[s_j] =$ `view-change` $\wedge$ $view[s_j] = v_{s_j}$
119 Start VC timeout Δ_{vc} for slot s_j
120 Stop query execute timeout $\Delta_{query-exec}$ for slot s_j
121 **Timeout** Δ_{vc} **for slot** s_j **expires:**
122 Move to new view $v_{s_j} + 1$
123 **Replica** r_i **receives** $\langle \text{NEWVIEW}, v_{s_j}, s_j, co, dp, \vec{dv}, VCS \rangle$:
124 `pre:` $step[s_j] =$ `view-change` $\wedge$ $view[s_j] = v_{s_j}$
125 `assert` co is view-change coordinator for view v_{s_j}
126 `assert` $\forall VC \in VCS : VC$ is valid
127 `assert` $dp, \vec{dv}$ correctly picked based on VCS
128 $\langle p[s_j], pr[s_j] \rangle := dp$
129 $v[s_j][*] := \emptyset$ /* *Cleanup* DEPVERIFY*s* */
130 $\forall dv \in \vec{dv} : v[s_j][dv.f_i] := dv$
131 If $s_j.i = r_i \wedge dp =$ `null`:
132 `permute-fast-quorum()`
133 Re-propose request in a new slot
134 Start commit timeout Δ_{commit} with reduced timeout $\Delta_{vc-commit}$ for slot s_j
135 Enter reconciliation path
136 **Timeout** $\Delta_{query-exec}$ **for slot** s_j **expires:**
137 Broadcast $\langle \text{QUERYEXEC}, s_j, r_j \rangle_{\sigma_{r_i}}$ to all replicas
138 **Replica** r_i **receives** $\langle \text{QUERYEXEC}, s_j, r_j \rangle$:
139 `pre:` $exec[s_j] \neq \emptyset$
140 $\langle dp, D \rangle := exec[s_j]$
141 Send $\langle \text{EXEC}, s_j, r_i, dp, D \rangle_{\sigma_{r_i}}$ to replica r_j
142 **Replica** r_i **receives** $\langle \text{EXEC}, s_j, *, dp, D \rangle$ **from** $f+1$ **replicas:**
143 `pre:` $exec[s_j] = \emptyset$
144 $exec[s_j] := \langle dp, D \rangle$
145 Forward $\langle dp, D, s_j \rangle$ to execution with dependencies D

B. Proof

We show that ISOS provides the following properties:

- **Validity**: Only correctly signed client requests are executed.
- **Consistency**: Two correct replicas commit the same request and dependencies for a slot. [5]
- **Execution Consistency**: Two interfering requests are executed in the same order on all correct replicas. [5]
- **Linearizability**: If two interfering requests are proposed one after another, such that the first request is executed at some correct replica before the second request is proposed, then all replicas will execute the requests in that order.
- **Agreement Liveness**: In synchronous phases a client request will eventually commit. [5]
- **Execution Liveness**: In synchronous phases a client will eventually receive a result.

We write p^i to refer to a variable p from the perspective of replica r_i.

We make the following standard cryptography assumptions: A message m signed by replica r_i is denoted as $\langle m \rangle_{r_i}$. All replicas are able to verify each other's signatures. A malicious replica is unable to forge signatures of correct replicas. All replicas drop messages without a valid signature.

By $h(m)$ we refer to the hash of a message m calculated using a collision resistant hash function, that is $h(m) \neq h(m') \Rightarrow m \neq m'$.

Messages for a slot are delivered eventually by retransmitting them unless the slot was garbage collected in the meantime. That is we assume reliable point-to-point connections between all replicas until slots are garbage collected. Once a replica has successfully completed a view change for a slot, then it is no longer necessary to retransmit messages for earlier views. It is also not necessary to retransmit DEPPROPOSE and DEPVERIFY messages once a new view was entered for the slot. In addition, for each message type only the message from the highest view in which the message type was sent has to be retransmitted.

We first show the properties for ISOS without checkpointing and extend the pseudocode and proofs to include checkpointing later on.

1) Validity:

Theorem A.1 (Validity)**.** *Only correctly signed client requests are executed.*

Proof: The execution algorithm only executes committed requests. For a client request to execute it must reach Line 59 or 85 in the Agreement Protocol in variable $p[s_j]$ or be received via EXEC messages in Line 145. $p[s_j]$ is set in

- Line 16 and 32: The validity of the client request was verified after receiving.
- Line 128: The value can be `null` or a value from a certificate. As each valid certificate contains $2f$ DEPVERIFYs from the initial view, one must be from a correct replica and as a correct replica only creates a valid DEPVERIFY once after verifying the client request (L. 35), the request must be correct. Otherwise, a correct replica must have created two DEPVERIFYs which yields a contradiction.

Requests received via EXEC are only forwarded to the execution if a replica receives $f+1$ matching EXECs. Thus, at least one EXEC is from a correct replica which must have processed the request according to one of the two previous cases.

The `null` request is skipped during execution and thus only correctly signed client requests are executed. ∎

2) Consistency: We first establish some notation:

Definition A.2. *A slot* s_j *is* verified *if a correct replica collects a valid* DEPPROPOSE dp, $2f$ *valid* DEPVERIFY*s from different replicas with matching* $h(dp)$ *and each* DEPVERIFY *is from a replica in the fast-path quorum* $dp.F$.

Definition A.3. *A slot* s_j *is* fp-verified *if a correct replica* verified *it and each dependency in the* DEPVERIFY*s occurs at least* $f+1$ *times.*

Definition A.4. *A slot s_j is* fp-committed *if a correct replica collects $2f+1$* DEPCOMMIT*s from different replicas with matching $h(\vec{dv})$.*

Remark A.5. *Note that* fp-committed *implies* fp-verified *as* DEPCOMMIT*s are only sent by replicas which* fp-verified *the slot.*

Definition A.6. *A slot s_j is* rp-verified *if a correct replica* verified *it and it is not* fp-verified*.*

Definition A.7. *A slot s_j is* rp-prepared *if a correct replica collects $2f+1$* PREPARE*s from different replicas with matching $h(\vec{dv})$.*

Definition A.8. *A slot s_j is* rp-committed *if a correct replica collects $2f+1$* COMMIT*s from different replicas with matching $h(\vec{dv})$.*

Remark A.9. *Similar to* fp-committed*,* rp-committed *implies* rp-prepared *and* rp-prepared *implies* rp-verified*.*

Definition A.10. *A slot s_j is* committed *if a correct replica* fp-committed *or* rp-committed *it.*

Theorem A.11 (Consistency)**.** *Two correct replicas commit the same request and dependencies for a slot.*

We first proof the following auxiliary lemmas.

Lemma A.12. *A slot s_j cannot both be* fp-committed *and* rp-prepared *in $view = -1$.*

Proof: By contradiction. Assume that a slot is both *fp-committed* and *rp-prepared* in $view = -1$. As the slot was *rp-prepared* a correct replica received $2f+1$ PREPAREs (L. 76). This requires $f+1$ correct replicas to have entered the reconciliation path (via L. 52 and 72). To be *fp-committed*, another replica must have received $2f+1$ DEPCOMMITs. As sending a DEPCOMMIT (L. 50) and entering the reconciliation path (L. 52) are mutually exclusive, a correct replica must have done both which yields a contradiction. ■

Lemma A.13. *The content of a reconciliation-path certificate (RPC) cannot be manipulated.*

Proof: As each protocol phase includes hashes of the previous phase, faulty replicas can only manipulate the last round of messages included in a certificate. For an RPC only the PREPAREs can be manipulated, however, these only confirm the values selected by the DEPPROPOSE and DEPVERIFYs. ■

Lemma A.14. *A faulty replica can only create a manipulated but valid fast-path certificate (FPC) if not* fp-committed*.*

Proof: A faulty replica could construct a faulty FPC using manipulated DEPVERIFYs, allowing the replica to include manipulated dependency sets. A replica only takes the fast path, if each dependency was reported in at least $f+1$ DEPVERIFYs (L. 47). This requirement is also necessary for an FPC to be valid.

Only a single DEPPROPOSE can be *verified*, which requires $2f$ matching DEPVERIFYs, as each correct replica only sends a DEPVERIFY for the first DEPPROPOSE for a slot. Thus correct replicas use the same fast-path quorum F to create an FPC and all valid FPCs must use the same F. To change the dependency sets faulty replicas only have the option to create manipulated DEPVERIFYs.

We now proof the Lemma by contradiction. Assume *fp-committed* holds. Then a correct replica has received $2f$ DEPVERIFYs in which each dependency is part of 0 (nonexistent dependency) or $f+1$ DEPVERIFYs.

- 0 occurrences: A manipulated FPC can either not include the dependency, in which case the FPC is unchanged. Or include a new dependency up to f times, which causes the FPC to become invalid.
- $f+1$ or more occurrences: A manipulated FPC can either include the existing dependency at least $f+1$ times, in which case the outcome of applying the FPC is unchanged. Or include a dependency only between 1 and f times, which causes the FPC to become invalid.

■

Lemma A.15. *A manipulated FPC can only be used if neither* fp-committed *nor* rp-committed*.*

Proof: If *fp-committed*, then no manipulated FPC can exist. If *rp-committed*, at least $f+1$ correct replicas have *rp-prepared* and thus at least one RPC is contained in one of the $2f+1$ VIEWCHANGEs required for the view change. Thus the FPC is ignored. As *fp-committed* and *rp-prepared* are mutually exclusive and *rp-committed* implies *rp-prepared*, no valid FPC can exist. ■

Now we proof Theorem A.11:

Proof: **Case 1:** A replica r_i commits $\langle c, D, s_j \rangle$ via the fast-path (L. 59). c is the request committed with dependencies D for slot s_j.

- Case 1.1: Another replica r_k commits $\langle c', D', s_j \rangle$ with $c \neq c' \lor D \neq D'$ via the fast-path.
  - Case $c \neq c'$:
    Proof: Then $p^i[s_j] \neq p^k[s_j]$, as $c \neq c'$. $h(p[s_j])$ is part of the DEPVERIFYs. Therefore, $h(\vec{dv})$ must differ. Then r_i and r_k each need $2f+1$ DEPCOMMITs with different $h(\vec{dv})$, which due to the properties of a Byzantine majority quorum would require a correct replica to send two DEPCOMMITs, which yields a contradiction. ■
  - Case $D \neq D'$:
    Proof: With $D := \cup D_{f_i} \in \vec{dv}$ it follows, that for a differing fast-path quorum F or dependency sets D_{f_i}, r_i and r_k must use different $h(\vec{dv})$. Now, the proof of the previous case applies. ■
- Case 1.2: r_k commits in view -1 via the reconciliation path.
  Proof: Then *rp-committed* holds which requires $2f+1$ COMMITs (L. 80) of which $f+1$ must originate from correct replicas. This implies *rp-prepared* which

according to Lemma A.12 conflicts with *fp-committed*. ■

- Case 1.3: r_k commits $\langle c', D', s_j \rangle$ with $c \neq c' \vee D \neq D'$ in $view \geq 0$ via the reconciliation path.
  Deferred to Case 3.

Case 2: A replica r_i commits $\langle c, D, s_j \rangle$ via the reconciliation path in $view = -1$ (L. 85).

- Case 2.1: r_k commits $\langle c', D', s_j \rangle$ with $c \neq c' \vee D \neq D'$ via the fast-path.
  Proof: See Case 1.2. ■
- Case 2.2: r_k commits in view -1 via the reconciliation path.
  Proof: This requires two sets of $2f+1$ PREPAREs with different $h(\vec{dv})$ which would require a correct replica to send two different PREPAREs. ■
- Case 2.3: r_k commits $\langle c', D', s_j \rangle$ with $c \neq c' \vee D \neq D'$ in $view \geq 0$ via the reconciliation path.
  Deferred to Case 3.

Case 3: A replica r_k commits $m' := \langle c', D', s_j \rangle$ via the reconciliation path in $view \geq 0$.

Proof: We proof this by induction: Once a replica commits $m := \langle c, D, s_j \rangle$, with $c \neq c' \vee D \neq D'$, in some $view$, then no replica can commit or prepare a different result m' in views $> view$.

Base case: $view' = view + 1$:

Assume that m committed in $view$ and that m' prepares / commits in $view'$. A correct replica only decides a result in view $view'$ after receiving a valid NEWVIEW. No manipulated RPC and FPC are used according to Lemma A.13 and A.15.

- Case $view = -1 \wedge$ *fp-committed*: No RPC can exist, as *fp-committed* and *rp-prepared* are mutually exclusive. As the fast-path committed, at least $f+1$ correct replicas have *fp-verified* the slot. These will include an FPC in their VIEWCHANGE. As the view-change coordinator has to wait for $2f+1$ VIEWCHANGEs, at least one VIEWCHANGE will include the FPC. The FPC would include m, which contradicts the assumption.
- Case $view = -1 \wedge$ *rp-committed* : $f+1$ correct replicas must be *rp-prepared* and thus provide the view-change coordinator with an RPC, which must be included in the NEWVIEW. No correct replica can be *fp-committed*, as it is mutually exclusive with *rp-prepared*. Therefore, if valid RPC and FPC exist, then the FPC is from a faulty replica and must be ignored. Thus, the selected RPC contains m which yields a contradiction.
- Case $view \geq 0$: The slot must have *rp-committed* and thus, similar to the previous case, the view change correctly selects the RPC. Therefore, the RPC contains m which yields a contradiction.

Induction step: $view' > view + 1$:

To commit a slot, $2f+1$ replicas have to send a DEPCOMMIT or COMMIT. One VIEWCHANGE with a corresponding certificate from a correct replica must be part of the $2f+1$ VIEWCHANGE messages. A correct replica always sends its newest certificate (L. 91-95), and therefore one of the VIEWCHANGEs used by the view-change coordinator includes a certificate from the highest view v_{max} in which a request has committed.

- Case $v_{max} \geq 0$: Thus the reconciliation path must have committed in v_{max}, and therefore the correct certificate is selected (L. 112-115).
- Case $v_{max} = -1$: The existence of an RPC shows that not *fp-committed* and thus the RPC must be selected. If only an FPC exists, then not *rp-committed* and therefore it is valid to select the FPC.

■

Case 4: A replica r_k commits $\langle dp, D, s_j \rangle$ after receiving $f+1$ valid $\langle \text{EXEC}, s_j, *, dp, D \rangle$ messages (Line 136-145). This case allows lagging replicas to catch up and learn the agreement result as described in Section IV-E.

Proof: At least one of the EXEC messages is from a correct replica, which either has committed the slot itself in which case the previous cases apply. Or the correct replica has learned from another correct replica that the slot was committed. ■

The cases are exhaustive. ■

3) Execution Consistency: Execution pseudo code

```
146 Variables at each replica:
147 k                                              /* Size of execution window */
148 committed, executed /* Sets containing all slots which have been
        commmitted or executed so far */
149 exp(r_i) := min{v_min | v_min ∉ executed ∨ v_min.i = r_i}
        /* First not executed request for replica r_i, defines the lower
        bound of the execution window */
150 exp_k := {v | v.s_i < exp(v.i) + k}
        /* Executed slots and slots in execution window */
151 rhist[*] := ⊥   /* History variable for dependency graph calculation */
```

Request Execution

```
152 /* Calculate dependency graph for slot v */
153 rdeps(v):
154   D' := {v}
155   While D ≠ D':
156     D := D'
157     For v ∈ D:
158       If v ∉ executed:
159         D' := D' ⋃_{d∈deps(v)} (v → d) ∪ {d}
160       Else:
161         D' := D' ∪ rhist[v]
162   Return D
163
164 /* Calculate dependency graph for slot v. Excludes slots outside the
        execution window */
165 rdeps_exp(v):
166   D' := {v} ∩ exp_k
167   While D ≠ D':
168     D := D'
169     For v ∈ D:
170       If v ∉ executed:
171         D' := D' ⋃_{d∈deps(v),d∈exp_k} (v → d) ∪ {d}
172       Else:
173         D' := D' ∪ rhist[v]
174   Return D
```

```
175 While True:
176   Update slots committed in the meantime
177   /* repeat loop until no further suitable v exists */
178   For all v ∈ (exp_k \ executed) ∧ rdeps(v) ⊆ (committed ∩ exp_k):
179     sc⃗ := find not executed strongly-connected components in
              rdeps(v) in inverse topological order
180     For sc ∈ sc⃗:
181       /* normal case execution */
182       execute(sc, rdeps(sc))
183   For all v ∈ (exp_k \ executed) ∧ rdeps_exp(v) ⊆ committed:
184     sc⃗ := find not executed strongly-connected components in
              rdeps_exp(v) in inverse topological order
185     /* unblock execution case */
186     execute(sc⃗[0], rdeps_exp(sc⃗[0]))
187
188 execute(v⃗, d):
189   For c ∈ sort(v⃗):
190     Execute request c
191     rhist[v] := d
192
193 sort(v⃗):
194   Return v⃗ sorted by sequence numbers v.seq and use replica id v.i as
          tie breaker
```

Similar to the Execution Consistency property in EPaxos [5], we show:

Theorem A.16 (Execution Consistency)**.** *All replicas execute all pairs of committed, conflicting requests in the same order.*

The relation $conflict(a, b)$ states that two requests a and b conflict with each other. The dependencies which were committed for a slot a are given by $deps(a)$. We first show that conflicting requests have dependencies on each other, before showing that conflicting requests are executed in the same order on all replicas.

Lemma A.17. *If $conflict(a, b)$ then, a has a dependency on b in $deps(a)$ or the other way around.*

Proof: For a request r one DepPropose and $2f$ DepVerify, that is in total $2f+1$ replicas, provide dependencies for the request.

For a and b at least one correct replica r_i receives both requests.

- r_i receives a before b: Then $a \in deps(b)$.
- r_i receives b before a: Then $b \in deps(a)$.

Therefore, the dependency is included when the slot is *committed* via the fast or reconciliation path in $view = -1$. When a replica *rp-prepares* the slot, then its RPC must by construction also include the dependency. As an FPC must include f DepVerifys and a DepPropose from correct replicas or $f+1$ DepVerifys from correct replicas, one of these messages includes the dependency. This is the case as a faulty replica cannot change the fast-path quorum F afterwards and thus cannot change which replicas contribute to an FPC.

In case no FPC or RPC is part of the view change, then a `null` request is selected. As that request does not conflict with any other request, no dependencies are required.

Note that the requirement for an FPC or RPC which include DepPropose and DepVerifys ensures that only the request coordinator can propose a request for the slot. ∎

Next, we show that conflicting requests are executed in the same order on all replicas. We start with several definitions used in the following:

Definition A.18 (SCC trace)**.** *A strongly-connected components (SCC) trace t is a 0-based vector consisting of executed SCCs in the order of their execution. We write t^i to refer to the trace belonging to a replica r_i.*

Definition A.19 (SSCC trace)**.** *A special-case strongly-connected component (SSCC) trace $\hat{t}$ is the subset of an SCC trace t containing only the SSCCs, that is all SCCs which were executed via the unblock execution case (Line 185).*

Definition A.20 (Regular SCC)**.** *A regular SCC is one which was executed via the normal case execution (Line 181).*

All slots executed as part of a trace are given by $flatten(t) = \{v \mid v \in s, s \in t\}$.

For a slot v we write $v.i$ to refer to replica i which is the request coordinator for that slot. And $v.seq$ to access the attached counter / sequence number. For two slots v_1 and v_2 with $v_1.i = v_2.i$, we use $v_1 < v_2$ as shorthand for $v_1.seq < v_2.seq$.

We write $a \to b$ if a directly depends on b, that is $b \in deps(a)$. (Logical implications are written as $P \Rightarrow Q$.) $a \rightsquigarrow b$ also includes transitive dependencies, that is $a \rightsquigarrow b \Leftrightarrow a \to b \lor a \to v_1 \to \ldots \to v_n \to b$, with $n \in \mathbb{N}$.

$rdeps(v)$ and $rdeps_{exp}(v)$ define dependency graphs starting from a slot v. They return a graph consisting of slots and edges $v_1 \to v_2$ between slots in these graphs. By construction all slots and edges in the graph are reachable from v.

Corollary A.21. *An SCC trace fully defines the order in which requests are executed.*

Proof: Slots can only be executed via $execute(\vec{v}, d)$, which groups requests by SCCs. As the execution algorithm filters out executed slots, each slot is only executed once and can thus only be part of one SCC. Note that the inverse topological sorting ensures that slots in an SCC can only depend on the SCC itself or earlier SCCs. Requests within an SCC are sorted before execution, which yields a stable order. ∎

Note that by definition $executed^i = flatten(t^i)$.

We first show the following supporting Lemma:

Lemma A.22. *Assume correct replicas r_i and r_j have two traces t^i and t^j where $\hat{t^i} = \hat{t^j}$. Then $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$, where v is a slot in an SCC.*

That is for all slots executed at both replica r_i and r_j ($\forall v \in flatten(t^i) \cap flatten(t^j)$), the dependency sets used for ordering are identical on these replicas for each of these slots.

We proof Lemma A.22 in multiple steps by induction. Base case of Lemma A.22: The Lemma applies for an SSCC trace $\hat{t}$ with length 0, that is trace t only contains regular SCCs.

Lemma A.23. *When* $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$ *before executing an SCC s via the normal case then* $\forall v' \in s : rdeps^i(v') = rdeps^j(v')$.

Proof: By construction an SCC is only executed after all its slots were committed, therefore $deps^i(v) = deps^j(v) = deps(v)$ are identical on all replicas for any slot v used during the calculation of $rdeps^i(v')$ and $rdeps^j(v')$.

We set v_1 to be an arbitrary slot of SCC s. As by assumption no slot of the SCC was executed before, $rdeps(v_1)$ must expand all slots v_s of the SCC s via $deps(v_s)$ (Lines 159 and 171). As all dependencies of an SCC are executed before the SCC, then for slot v_d in one of these dependencies, $rhist[v_d]$ is identical on all replicas. Therefore $rdeps^i(v_1) = rdeps^j(v_1)$.

This completes the proof for SCCs of size 1. In the following we consider SCCs consisting of at least two slots and show that if two SCCs at different replicas have at least one slot in common, then the SCCs are identical. By definition $\forall v_2 \in SCC, v_1 \neq v_2 : v_1 \rightsquigarrow v_2 \wedge v_2 \rightsquigarrow v_1$. Thus $rdeps^i(v_1) = rdeps^i(v_2)$.

Now assume that $s^i \neq s^j \wedge s^i \cap s^j \neq \emptyset$: W.l.o.g $v_1 \in s^i, \notin s^j$ and $v_2 \in s^i \cap s^j$. Then $v_1 \in rdeps^i(v_1) = rdeps^i(v_2) = rdeps^j(v_2)$ and therefore $v_1 \in s^j$ which yields a contradiction. Thus $s^i = s^j$. ■

In the following we use $rdeps(v)$ and $rdeps(s)$ for slot $v \in$ SCC s interchangeably.

Lemma A.24. *Assume replica* r_i *and* r_j *have two traces* t^i *and* t^j *which only contain SCCs executed via the normal case, that is* $|\hat{t^i}| = 0$ *and* $|\hat{t^j}| = 0$. *Then* $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$, *where* v *is a slot in an SCC.*

Proof: W.l.o.g. we assume $|t^i| = |t^j|$. A short trace can be padded with empty "SCCs" which are equivalent to a no-op.

Base case $|t^i| = 0$: $flatten(t^i) \cap flatten(t^j) = \emptyset$.

Induction step $|t'^i| = |t^i| + 1$: We define $s^i := t'^i[|t'^i| - 1]$ to be the last element in t'^i. We only discuss s^i, the same arguments apply for an s^j with swapped i and j.

Case 1: $s^i \in set(t'^i) \cap set(t'^j)$: Both t'^i and t'^j contain $s = s^i$. As the SCCs are sorted in inverse topological order, s is only executed after all SCCs on which s depends were executed first, that is $rdeps^i(s)$ can only contain slots $v \in s \cup executed$. Note that all slots in the SCC s must already be committed (and not yet executed) as the SCC would not be executed otherwise. Per (Agreement) Consistency property the set $\mathcal{D} := \bigcup_{v \in s} deps(v)$ is identical on all replicas, and thus s directly depends on the same slots on all replicas. These slots (and their SCCs) have been executed on both replicas r_i and r_j and thus per induction assumption $\forall v \in \mathcal{D} : rhist^i[v] = rhist^j[v]$. Thus $rdeps^i(s) = rdeps^j(s)$ and therefore $rhist^i[s] = rhist^j[s]$.

Case 2: $s^i \notin set(t'^i) \cap set(t'^j) \Leftrightarrow s^i \notin t'^j$: We show that for all $s^j \in t'^j$: if $s^i \neq s^j$, then $s^i \cap s^j = \emptyset$. Assume that s_j is the SCC with the lowest index in t'^j with $s^i \cap s^j \neq \emptyset$. With Lemma A.23 this results in a contradiction.

The cases are exhaustive.

Thus, we have shown that $\forall v \in \{v | s \in set(t^i) \cap set(t^j), v \in s\} : rhist^i[v] = rhist^j[v]$. We also know that for $s_1 \in t^i, s_2 \in t^j : s_1 \cap s_2 \neq \emptyset \Rightarrow s_1 = s_2$. To complete the proof of the Lemma we have to show that $LHS := flatten(t^i) \cap flatten(t^j)$ equals $RHS := \{v | s \in set(t^i) \cap set(t^j), v \in s\}$. By construction $LHS \supseteq RHS$, thus we have to show that $LHS \not\supset RHS$.

Assume that is not the case: Pick v such that $v \in LHS, v \notin RHS$. Then $v \in s_1 \in t^i, v \in s_2 \in t^j$.

Case $s_1 = s_2$: Contradiction.

Case $s_1 \neq s_2$: Thus $s_1 \cap s_2 = \emptyset$, which contradicts $v \in s_1, s_2$. ■

Corollary A.25. *The compact dependency representation implicitly includes dependencies on all earlier slots of a replica. That is a dependency from slot a to slot b ensures that* $b \in deps(a) \Rightarrow deps(a) \supseteq deps(a, b) = \{v | v.i = b.i \wedge v.seq \leq b.seq\}$.

Lemma A.26. *A regular SCC is only executed by the normal case execution, whereas an SSCC is only executed by the unblock execution case.*

Proof: To reach a contradiction, assume that a regular SCC is executed via the unblock execution case. For this, we must find a slot $v \in exp_k \cap committed, v \notin executed$ which satisfies the following condition: $rdeps_{exp}(v) \subseteq committed \wedge \neg(rdeps(v) \subseteq committed \cap exp_k)$. The first part of the condition ensures that the unblock execution case can execute (Line 183) and the second part ensures that the normal case execution does not apply (Line 178). $\Leftrightarrow rdeps_{exp}(v) \subseteq committed \wedge (rdeps(v) \not\subseteq committed \vee rdeps(v) \not\subseteq exp_k)$ $\Leftrightarrow rdeps_{exp}(v) \subseteq committed \wedge (rdeps(v) \setminus rdeps_{exp}(v) \not\subseteq committed \vee rdeps(v) \not\subseteq exp_k)$.

We also make the following observation: $rdeps(v) \subseteq exp_k \Rightarrow rdeps(v) = rdeps_{exp}(v)$. If $rdeps(v) \subseteq exp_k$ then the check using exp_k in $rdeps_{exp}$ never skips dependencies and therefore $rdeps(v) = rdeps_{exp}(v)$.

Assume that the unblock case would execute while $rdeps(v) \subseteq exp_k$, then $rdeps(v) \setminus rdeps_{exp}(v) = \emptyset$ which prevents the unblock case from executing. Thus, the unblock case only executes if $rdeps(v) \not\subseteq exp_k$.

Due to the inverse topological sort order the SSCC $sc[0]$ in the unblock execution case must have $rdeps_{exp}(sc[0]) \setminus sc[0] \subseteq executed$ that is all dependencies except $sc[0]$ must be executed and $sc[0] \subseteq rdeps_{exp}(sc[0])$. As $sc[0] \subseteq exp_k$, $rdeps(sc[0]) \supset rdeps_{exp}(sc[0])$ and therefore $\exists x_a \in rdeps(sc[0]) : (x_a \to x_e) \in rdeps(sc[0]) \wedge x_e \notin exp_k$. Due to Corollary A.25 $(x_a \to exp(x_e.i)) = (x_a \to x_r) \in rdeps_{exp}(sc[0]), x_r \in rdeps(sc[0])$. By definition $x_e.seq - x_r.seq \geq k$ and therefore always $rdeps(sc[0]) \not\subset exp_k$. Thus, $sc[0]$ can never be executed via the normal case, which contradicts the assumption that $sc[0]$ is a regular SCC.

For $v \in SSCC$, as $rdeps(v) \not\subset exp_k$ before executing v, it can never execute via the normal execution case. ■

Lemma A.26 allows strengthening Lemma A.24 to:

Lemma A.27. *Assume replica r_i and r_j have two traces t^i and t^j which only contain regular SCCs, that is $|\hat{t^i}| = 0$ and $|\hat{t^j}| = 0$. Then $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$, where v is a slot in a (regular) SCC.*

Lemma A.28. *Assume replica r_i and r_j have two traces t^i and t^j which only contain regular SCCs, that is $|\hat{t^i}| = 0$ and $|\hat{t^j}| = 0$. If $s^i \neq s^j$, then $s^i \cap s^j = \emptyset$.*

Proof: Follows from the proof of Lemma A.24 and A.26. ■

Induction step 1 of Lemma A.22: The lemma applies for a fixed SSCC trace $\hat{t}'$ with length $|\hat{t}|+1$ where the SSCC is the last element of trace t'. We refer to it as $\hat{s} := t'[|t'|-1] = \hat{t}'[|\hat{t}'|-1]$.

Lemma A.29. *$rhist^i[\hat{s}] = rhist^j[\hat{s}]$ for a fixed SSCC trace $\hat{t}'$ with $\hat{s} := t'[|t'|-1]$.*

Proof: For an SSCC to execute via the unblock case, the following must hold: $rdeps_{exp}(\hat{s}) \subseteq committed \wedge \neg(rdeps(\hat{s}) \subseteq committed \cap exp_k)$. As shown in the proof of Lemma A.26 this requires $rdeps(\hat{s}) \not\subseteq exp_k$.

$rdeps_{exp}(\hat{s})$ depends on $deps(v)$, $rhist[v]$ and exp_k. $deps(v)$ is identical across replicas by the (Agreement) Consistency property and $rhist[v]$ is identical across replicas as SCC dependencies are executed first and thus this follows from the induction assumption. In the SSCC there must exist slots $v_m \in exp_k$ with a dependency on a slot $v_e \in deps(v_m), v_e \notin exp_k, \notin rdeps_{exp}(\hat{s})$. We now show that $\forall v_m \in exp_k : \forall v_e \in deps(v_m), v_e \notin exp_k : exp^i(v_e.i) = exp^j(v_e.i)$. Assume that this is not the case.

By Corollary A.25, $deps(v_m) \supseteq deps(v_m, v_e)$. We set $v_r = min(\{d | d \in deps(v_m, v_e) \wedge d \notin executed\})$. By definition $exp(v_e.i) = v_r$. Note that v_r must be part of SSCC as it would have to be $\in executed$ otherwise, which contradicts the definition of v_r. That is for all replicas r_l to which dependencies are removed in $rdeps_{exp}$, $exp(r_l)$ is defined by the slots in the SSCC. Thus $rdeps^i_{exp}(\hat{s}) = rdeps^j_{exp}(\hat{s})$.

Due to Lemma A.26 the SSCC can only be executed via the unblock execution case. ■

Lemma A.30. *Assume replica r_i and r_j have two traces t^i and t^j where the SSCC $\hat{s}$ is the last element and $\hat{t^i} = \hat{t^j}$. Then $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$, where v is a slot in an SCC.*

Proof: By construction slots are only executed once, thus either $v \in flatten(\hat{t})$ or $v \in flatten(t^i \setminus \hat{t}) \cap flatten(t^j \setminus \hat{t})$. Then $rhist^i[v] = rhist^j[v]$ follows from Lemmas A.27 and A.29. ■

Induction step 2 of Lemma A.22: The fixed SSCC trace $\hat{t}'$ has length $|\hat{t}|$ and the corresponding SCC trace t ends with a sequence of regular SCCs.

Lemma A.31. *At replica r_i and r_j for two traces t^i and t^j with identical SSCCs, that is $\hat{t^i} = \hat{t^j}$: $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$, where s is an SCC.*

Proof: W.l.o.g. we assume $|t^i| = |t^j|$. A short trace can be padded with empty "SCCs" which correspond to a no-op. In addition, we assume that there is a position x such that $t^i[x] = \hat{t}[|\hat{t}-1|] = t^j[x]$, that is the last SSCC is at the same index in both traces. By assumption, all SCCs after the SSCC are executed regularly.

Base case: $|t| = x+1$: Follows from Lemma A.30.

Induction step: $|t'| = |t|+1$: We define $s^i := t'^i[|t'^i|-1]$ to be the last element in t'^i. We only discuss s^i, the same arguments apply for an s^j with swapped i and j.

Case 1: $s^i \in set(t'^i) \cap set(t'^j)$: The same as Case 1 in the proof of Lemma A.24, except that the induction assumption is strengthened with Lemma A.30.

Case 2: $s^i \notin set(t'^i) \cap set(t'^j) \Leftrightarrow s^i \notin t'^j$: This never applies as regular SCCs and SSCCs are disjoint, according to Lemma A.24 (Case 2) and A.29. ■

We now finish the proof of Lemma A.22.

Lemma A.22 (repetition). *Assume replica r_i and r_j have two traces t^i and t^j:$\forall v \in flatten(t^i) \cap flatten(t^j)$. Then $rhist^i[v] = rhist^j[v]$, where v is a slot in an SCC.*

Proof: We assume w.l.o.g. that $|\hat{t^i}| = |\hat{t^j}|$. A short SSCC trace can be padded with empty no-op SSCCs.

Base case: $|\hat{t^i}| = |\hat{t^j}| = 0$: See Lemma A.27.

Induction step: $|\hat{t^i}'| = |\hat{t^i}|+1$: We only show this for $\hat{s^i} := \hat{t^i}'[|\hat{t^i}'|-1]$, a symmetrical argument applies for s^j.

Case 1: $\hat{s^i} \in set(\hat{t^i}') \cap set(\hat{t^j}')$: All SCCs on which $\hat{s^i}$ depends have already been executed, thus $rdeps^i_{exp}(\hat{s^i}) = rdeps^j_{exp}(\hat{s^i})$ as shown in the proof of Lemma A.29.

Case 2: $\hat{s^i} \notin set(\hat{t^i}') \cap set(\hat{t^j}')$: We show that if $\hat{s^i} \neq \hat{s^j}$, for a $\hat{s^j} \in \hat{t^j}$ then $\hat{s^i} \cap \hat{s^j} = \emptyset$. For that we show that an SSCC can be identified by a single slot v. An SSCC depends on $\mathcal{D} := \bigcup_{v \in \hat{s^i}} exp(v.i)$. $v \in \mathcal{D}$ must be part of the SSCC as otherwise they would have been executed before. Thus, $exp(...)$ is defined by the SSCC. And therefore $\forall v \in \hat{s^i} : rdeps^i_{exp}(v) = rdeps^j_{exp}(v)$.

The cases are exhaustive. ■

Theorem A.16 (repetition) (Execution Consistency)**.** *All replicas execute all pairs of committed, conflicting requests in the same order.*

Now we proof the theorem:

Proof: The agreement guarantees that for two conflicting requests a and b in slots v_1 and v_2, at least one will depend on the other. W.l.o.g. assume that $v_2 \in deps(v_1)$ and that v_1 and v_2 were already executed.

Case 1: v_1 and v_2 are part of the same regular SCC or SSCC: An SCC is sorted before executing, thus ensuring a stable order.

Case 2: $v_2 \in rhist[v_1]$: Then v_2 was executed before v_1. Assume this is not the case: This is only possible if v_1 and v_2 are part of a single SCC, which contradicts the assumption.

Case 3: $v_2 \notin rhist[v_1]$: v_1 must be part of an SSCC, as only $rdeps_{exp}$ can exclude dependencies from $deps(v_1)$.

We first show that the execution behaves as if the following additional dependencies for each slot exist: $\tilde{deps}(v) \hat{=} deps(v) \cup \{x \,|\, x.i = v.i \wedge x.seq \leq v.seq - k\}$. We

adapt $r\tilde{deps}$ and $r\tilde{deps}_{exp}$ to use $\tilde{deps}(v)$. When v is executed, $v \in exp_k$ thus $x.seq \leq v.seq - k < exp(v.i).seq$ and thus the additional dependencies only point to already executed slots. Therefore, they do not affect the execution.

When the SSCC was executed $v_1 \in exp_k, v_2 \notin exp_k$. Then we get: $v_r = exp(v_2.i) \in \tilde{deps}(v_1)$ due to Corollary A.25, $v_r \in \tilde{deps}(v_1)$, $v_r \in \tilde{deps}(v_2)$ and $(v_1 \rightsquigarrow v_r \wedge v_r \rightsquigarrow v_1) \vee v_1 = v_r$ as v_1 and v_r are part of the SSCC. Therefore, v_2 can only execute after the SSCC as $v_r \in SSCC$ and $v_2 \rightarrow v_r$.

The cases are exhaustive.

In contrast to the SCC trace, the execution pseudocode starts from individual slots and tests whether a slot and its dependency graph are executable. Only then the SCCs are calculated and executed. When the tested slots are part of the SCC to execute next, then it is trivial to see that both representations are equivalent. Now suppose slot v_b of SCC B which depends on SCC A is tested first. If both SCCs are regular SCCs, then SCC A will be executed before B. As $rhist[v_a] := rdeps(A)$ for $v_a \in A$ it makes no difference whether $rdeps(B)$ is calculated before or after executing SCC A.

If only A is an SSCC, then A is executed first and afterwards the execution is restarted, which includes a recalculation of $rdeps(v_b)$. If only B is an SSCC, then we arrive at a contradiction, as A must already have been executed. If both are SSCCs, then one of both is executed and afterwards the execution is restarted. In all these cases the behavior is equivalent to that assumed when working with SCC traces. This generalizes to dependency graphs which contain more than two SCCs. ∎

Remark A.32. *It is sufficient for the unblock execution case to only check slots in $exp(*)$, that is the root nodes. As shown in Lemma A.26 at least one slot in every SSCC is $\in exp(*)$.*

Remark A.33. *$rhist$ can be ignored for an implementation, as by construction it only contains executed slots. An already executed slot cannot have dependencies on not yet executed slots. Therefore, slots in $rdeps(v)$ and $rdeps_{exp}(v)$ can be split into two sets $\mathcal{A} \subseteq executed$ and $\mathcal{B} \cap executed = \emptyset$ with executed and not executed slots, respectively. Only slots in $\mathcal{B}$ can depend on slots in $\mathcal{A}$. A similar structure applies for the SCCs in $rdeps(v)$ or $rdeps_{exp}(v)$. As these SCCs are skipped if they were executed before, it is equivalent to remove executed slots from $rdeps$ or $rdeps_{exp}$ as well. The simplest way to achieve that is to drop $rhist$ completely.*

Remark A.34. *An implementation can handle $rdeps$ and $rdeps_{exp}$ using a single graph and immediately remove executed slots. This is easy to see for $rdeps$ alone, the combination with $rdeps_{exp}$ requires small modifications: Only slots $\in exp_k$ should be processed, all other slots can be regarded as not yet committed. Then $rdeps(v) \not\subseteq committed \Leftrightarrow rdeps(v) \not\subseteq exp_k$. $rdeps_{exp}(v)$ can be emulated by ignoring dependencies on slots $\notin exp_k$ on the fly.*

4) Linearizability:

Theorem A.35 (Linearizability)**.** *If two interfering reqeusts are proposed one after another, such that the first request is executed at some correct replica before the second request is proposed, then all replicas will execute the requests in that order.*

Proof: Once a request a was executed then all later conflicting requests b will depend on a and are thus ordered after a. To prevent the duplicate execution of client requests, the requests of a client always conflict with each other, which guarantees a total order for the requests of each client. ∎

5) Agreement Liveness: Similar to the Liveness property in EPaxos [5], we show:

Theorem A.36 (Agreement Liveness)**.** *In synchronous phases a client request will commit eventually.*

We first show that dependencies proposed by correct replicas will be accepted eventually, then show that a slot will commit and finish by showing that this also holds for a client request.

Definition A.37. *We say that `wait` (Line 60) accepts a slot as dependency, if the function does not block permanently, that is it returns eventually.*

Lemma A.38. *If a correct replica r_i has accepted a* DEPPROPOSE *from replica r_j, then all other correct replicas will accept it as a dependency eventually.*

Proof: We show this by induction: For the base case assume that the DEPPROPOSE contains no dependencies. The propose timeout stays active at replica r_i until it has accepted $2f$ DEPVERIFYs for the DEPPROPOSE (L. 44).

- **Case 1:** Coordinator r_j is correct.
  All replicas will receive the DEPPROPOSE and thus `wait` accepts the slot as dependency.
- **Case 2:** Coordinator r_j is faulty.
  Assume that replica r_i has accepted $2f$ DEPVERIFYs. Only $f - 1$ faulty DEPVERIFYs are possible. Thus, at least $f+1$ of $2f$ DEPVERIFYs are from correct replicas. And therefore all replicas will receive $f+1$ DEPVERIFYs causing wait to accept the dependency. Otherwise, the propose timeout will expire, causing a correct replica r_k to broadcast the request. This allows all other replicas to learn about the slot corresponding to the DEPPROPOSE as the message was signed by the request coordinator.
  Alternatively, if replica r_i receives the DEPPROPOSE, then it will broadcast the message if it fails to collect $2f$ DEPVERIFYs within the propose timeout.
- **Case 3:** A view-change triggers at replica r_i.
  The replica r_i broadcasts the DEPPROPOSE to all replicas if the propose timeout was still active (Line. 87).
- The cases are exhaustive.

For the induction step we look at a later DEPPROPOSE for which the correct replica r_i must also have accepted all dependencies. Thus, if it is necessary to broadcast the DEPPROPOSE it will be accepted eventually as all dependencies will be accepted due to the induction assumption. ∎

Remark A.39. *Note that the view-change special case to broadcast the* DEPPROPOSE *(Line. 87) only exists for completeness, but is not strictly necessary during synchronous phases. For a view change at least one correct replica r_k must have sent a* VIEWCHANGE*. This in turn requires that the replica r_k has either received the* DEPPROPOSE *in which case it will broadcast the* DEPPROPOSE *itself if necessary. Or the replica r_k has received $f + 1$ valid* DEPVERIFY*s in which case at least one of these was sent by a correct replica, which must have received the* DEPPROPOSE *and therefore also ensures its distribution. Together with the commit timeout 9Δ which is much larger than the propose timeout of 2Δ, the special case can only trigger if another correct replica has received and possibly distributed the* DEPPROPOSE *before.*

Lemma A.40. *A dependency included in a request proposed by a correct replica r_i will be accepted by all correct replicas eventually.*

Proof: By construction, correct replicas only propose dependencies, for which they accepted the corresponding DEPPROPOSE. Then according to Lemma A.38 the corresponding messages will be accepted (by `wait`). ∎

Lemma A.41. `wait` *only accepts slots as dependencies which will commit eventually.*

Proof: The `wait` function waits for each dependency until one of the following cases holds (L. 62-65):

- **Case 1:** $f + 1$ DEPVERIFYs received.
  These include at least one DEPVERIFY from a correct replica, which must have received a valid DEPPROPOSE and which will broadcast it if necessary.
- **Case 2:** $f + 1$ VIEWCHANGEs received.
  At least one VIEWCHANGE is from a correct replica, which also ensures that a correct replica has received a valid DEPPROPOSE, see Remark A.39.
- **Case 3:** DEPPROPOSE accepted.
  This enables a replica to broadcast the DEPPROPOSE itself if necessary.
- The cases are exhaustive.

Together with Lemma A.38 and A.40 eventually the commit timeout is active at all correct replicas which forces the slot to commit. ∎

Assume for now that the used timeout values are large enough to ensure progress.

Lemma A.42. *A slot (not request) of a correct coordinator will commit eventually.*

Proof: **Case 1:** The fast-path quorum F only contains correct replicas.

Then one of the following can happen:

- Case 1.1: The slot commits without view change.
  Proof: Correct replicas enforce that a coordinator does not leave gaps in its sequence number space (L. 24). During a synchronous phase, the `wait` calls in lines 24 and 40 do not block permanently according to Lemma A.40. The coordinator and the fast-path quorum make up a total of $2f + 1$ correct replicas which allows the slot to commit. In an asynchronous phase the coordinator will retransmit its DEPPROPOSE until all correct replicas have received it. Then either the slot will commit or $\geq f + 1$ replicas trigger a view change.
  The replicas start the commit timeout after receiving the DEPPROPOSE or in the case of the request coordinator after sending the DEPPROPOSE and thus either commit or request a view change. Once $f + 1$ correct replicas have committed, then the remaining f correct replicas can only trigger a view change with the help of faulty replicas. When the view-change does not start within timeout $\Delta_{query-exec}$ after sending the own VIEWCHANGE, then a replica issues QUERYEXEC requests to all other replicas (L. 137). These up to f replicas then receive the result via EXEC messages from at least $f + 1$ correct replicas. ∎
- Case 1.2: A view change is necessary for at least one replica.
  Proof: As soon as $f + 1$ correct replicas have issued a VIEWCHANGE for view $v+1$ then eventually all correct replicas will issue a VIEWCHANGE (L. 105-107). In a synchronous phase eventually all correct replicas will enter the view change in the same view $v + 1$.
  The timeout for view $v + 2$ is only started after ensuring that at least $f+1$ correct replicas have reached view $v+1$ and sent a VIEWCHANGE. This in turn ensures that all correct replicas will reach view $v+1$ at the same time if the network in synchronous, see also Lemma A.46. Then all correct replicas will start their view change timeout, as enough VIEWCHANGEs exist to ensure that a NEWVIEW can be created eventually. Then either at least $f+1$ correct replicas accept the NEWVIEW or $f + 1$ correct replicas switch to view $v + 2$.
  After a replica r_i accepts a NEWVIEW, then a different replica r_j will either eventually also receive and accept the NEWVIEW or switch to a higher view. As $2f + 1$ VIEWCHANGEs are necessary to compute a NEWVIEW, at least $f + 1$ must be from correct replicas, thus eventually all replicas start a view change, will receive $2f+1$ VIEWCHANGE and start their view-change timeouts. Then either the replica accepts the NEWVIEW or switches to a higher view. After accepting a NEWVIEW a replica restarts the commit timeout which again ensures that the reconciliation path completes or another view-change is started.
  The view-change coordinator is rotated in each view such that eventually a correct coordinator is used, which allows the slot to commit. ∎
- Case 1.3: DEPPROPOSE and DEPVERIFY (from correct replicas) contain dependencies not accepted by `wait`.
  Proof: Using Lemma A.40 we immediately arrive at a contradiction. ∎
- The cases are exhaustive.

Case 2: Fast-path quorum F contains faulty replicas.

We show that faulty replicas in the fast-path quorum F cannot prevent committing a slot (only its request) and cannot add dependencies to non-existing slots. The faulty replicas can exhibit one of the following behaviors:

- Case 2.1: A faulty replica sends multiple DEPVERIFYs. *Proof:* The replica can prevent the fast or reconciliation path from completing when replicas collect diverging or no $\vec{dv}$. If the faulty replica prevents the slots from committing then the commit timeout enforces a view change. This will result in filling the slot with `null` after the view change. ■
- Case 2.2: A faulty replica does not send a DEPVERIFY. *Proof:* Same as the previous case. ■
- Case 2.3: A faulty replica proposes non-existing dependencies. *Proof:* According to Lemma A.41, correct replicas, which have received the DEPPROPOSE, will time out while waiting for the dependencies to commit. This will trigger a view change which will fill the slot with `null`. Thus, non-existing dependencies for a slot cannot commit. ■
- The cases are exhaustive.

The cases are exhaustive. ■

Lemma A.43. *The fast-path quorum F will eventually contain only correct replicas.*

Proof: After filling a slot with `null` during the view change, the fast-path quorum is rotated. This will eventually result in the fast-path quorum F to only contain correct replicas. ■

Remark A.44. *Note that a faulty replica cannot prevent slots of correct replicas from committing by proposing manipulated* DEPPROPOSE*s. Assume this were the case. Then a correct replica has to accept a* DEPPROPOSE *from the faulty coordinator. Then Lemma A.38 applies, which yields a contradiction. Thus, a faulty coordinator can only cause its* DEPPROPOSE *to block permanently in* `wait` *which will also prevent all further slots of the faulty replica from committing (L. 24) until the faulty* DEPPROPOSE *is finally accepted.*

We now show that the timeout values are sufficient to ensure progress.

Lemma A.45. *A* DEPPROPOSE *or* DEPVERIFY *of a correct replica r_i will be accepted after at most 3Δ after sending.*

Proof: Replica r_i has received the DEPPROPOSE of a dependency as otherwise it would not include the dependency. The propose broadcast timeout is 2Δ. Thus, after 2Δ replica r_i has either received $2f$ DEPVERIFYs and thus after an additional Δ all replicas have received $f+1$ DEPVERIFYs after which `wait` accepts the dependency. Or replica r_i broadcasts the DEPPROPOSE which will reach all replicas within Δ. ■

Lemma A.46. *The calculation of a* NEWVIEW *can complete within at most 3Δ in synchronous phases.*

Proof: Once a correct replicas has received $2f+1$ VIEWCHANGEs then within 2Δ every correct replica will receive $2f+1$ VIEWCHANGE. This allows the view-change coordinator to calculate the NEWVIEW which after Δ arrives at all replicas. That is, in total a timeout of 3Δ is sufficient. ■

Lemma A.47. *A commit timeout of at least 8Δ allows correct coordinators to commit in synchronous phases.*

Proof: It can take 3Δ each until a DEPPROPOSE and DEPVERIFY are accepted. The fast path takes another Δ until DEPCOMMIT reaches all replicas. On the reconciliation path PREPARE and COMMIT require up to 2Δ. This yields a total timeout of 8Δ. As the timeout cannot start before the DEPPROPOSE was sent, this is sufficient in all cases.

After a view change $\Delta_{vc-commit} = 3\Delta$ is sufficient as the reconciliation path only requires up to 2Δ and the receipt time of a correct NEWVIEW can only vary by Δ between replicas. ■

Theorem A.36 (repetition) (Agreement Liveness)**.** *In synchronous phases a client request will commit eventually.*

Now, we show Theorem A.36:

Proof: For slots in which the request was replaced by `null` the request coordinator will propose the request again (L. 133). Together with Lemmas A.42 and A.43 this ensures that a slot / slots and also eventually a request will commit. The client also broadcasts its request after a timeout to all replicas. This ensures that a correct coordinator receives the request and commits it. ■

Lemma A.48. *The compact dependency representation does not break Liveness.*

Proof: The additional dependencies to replica r_j have sequence numbers s_d which are lower than the maximum sequence number max_{s_j} to which an explicit dependency exists. That is $s_d < max_{s_j} = \max_{r_j}\{d \in D_i \,|\, d.i = r_j\}$. A correct replica accepts a DEPPROPOSE for max_{s_j} only if it has seen all earlier sequence numbers, that is `wait` must already have accepted these (L. 24). Thus the guarantees provided by `wait` also include the earlier additional sequence numbers s_d.

The dependency representation does not affect execution consistency as it can only add, but not remove, dependencies. ■

6) Execution Liveness:

Theorem A.49 (Execution Liveness)**.** *In synchronous phases a client will eventually receive a result.*

Lemma A.50. *Any slot included as dependency of a committed slot will commit eventually.*

Proof: The `wait` calls in Lines 24 and 40 together with Lemma A.41 ensure that all dependencies of any committed slot will commit eventually. ■

Lemma A.51. *A committed request will be executed eventually.*

Proof: Lemma A.50 shows that all slots on which a committed slot depends will commit eventually. In order to avoid the execution livelock problem discussed in EPaxos [5], we now show that there is a finite upper bound for the number of slots which have to commit before a slot can be executed.

A slot s can be executed via the normal case if all slots in $rdeps(s) \subseteq exp_k$. As exp_k by construction only includes up to k not executed slots per replica, the number of dependee slots is bounded.

In addition, the unblock execution case executes slots in $rdeps_{exp}(s)$ which by construction always is $\subseteq exp_k$ and thus all slots executed via the unblock case also only have to wait for a bounded number of dependencies.

A slot s can only depend on a bounded number of slots (as all dependencies must have been proposed before). Thus, if any dependency d among these dependencies is not yet executed and therefore can prevent execution of s, then it serves as an upper bound for $exp(d.i).seq \leq d.seq$. Other dependee slots can add further upper bounds on $exp(*)$ which restrict the size of the dependency set even further. As the lowest upper bound per replica is relevant, a dependency chain can only include additional requests by depending on another replica which is not yet part of $rdeps(s)$ or $rdeps_{exp}(s)$. As the number of replicas is fixed, this can only add dependencies to a bounded number of slots. ■

The theorem follows by combining Theorem A.36 and Lemma A.51.

7) Checkpoint Correctness: We now extend the proof and pseudocode to also include the checkpointing mechanism of ISOS.

We only show the modified parts of the agreement and execution pseudocode below. Grey lines are unchanged.

195 **Variables at each replica**:
196 $\Delta_{vc} := 5\Delta$

Fast Path

197 Propose checkpoint request CPREQ if $s_j.seq$ mod $cp_interval = 0$
198 **Follower** f_i **receives** $dp := \langle\langle \text{DEPPROPOSE}, s_j, co, h(r), D, F\rangle, r\rangle$:
199 pre: $step[s_j] =$ init
200 assert $(s_j.seq \text{ mod } cp_interval = 0) \oplus (r = \text{CPREQ})$ /* *Each replica must propose a checkpoint request exactly every cp_interval slots* */
201 [...]
202 **conflicts**(r):
203 /* *A checkpoint request* CPREQ *conflicts with all other requests* */
204 Return $\{s_i | \forall s_i, pr[s_i] \neq \emptyset : conflict(pr[s_i], r)\}$ ∪ barrier of latest stable checkpoint

View Change

205 **Move to new view** v_{s_j} **for slot** s_j **at replica** r_i:
206 [...]
207 Else If $step[s_j] \in \{$rp-prepared, rp-committed$\}$:
208 [...]
209 Else If $s_j.seq$ mod $cp_interval = 0$:
210 $D_{f_i} := D_{f_i}$ used by r_i for own DEPPROPOSE / DEPVERIFY or as fallback $conflicts(\text{CPREQ}) \setminus s_j$
211 $dv := \langle \text{DEPVERIFY}, s_j, f_i, h(\text{CPREQ}), D_{f_i}\rangle_{\sigma_{f_i}}$
212 $cert[s_j] := \langle \text{CRC-PART}, \text{CPREQ}, dv, -1\rangle$ /* *for view −1* */
213 $step[s_j] =$ view-change
214 [...]
215 **VC-Coordinator** co **for view** v_{s_j} **receives valid** $VCS := \{\langle \text{VIEWCHANGE}, v_{s_j}, s_j, *, *\rangle\}$ **from** $2f+1$ **replicas**:
216 /* *A* $VC \in VCS$ *containing a* CRC-PART *is only considered valid after* $wait(VC.dv.D_{f_i})$ *has returned* */
217 [...]
218 Pick dp, $\vec{dv}$ from [...]
219 If $s_j.seq$ mod $cp_interval = 0 \wedge dp = null$:
220 $dp :=$ CPREQ
221 $\vec{dv} := \{VC.dv \,|\, VC \in VCS\}$ /* *Each VC must contain a* DEPVERIFY */
222 Broadcast $\langle \text{NEWVIEW}, v_{s_j}, s_j, co, dp, \vec{dv}, VCS\rangle_{\sigma_{co}}$

The execution pseudocode is adapted as follows:

223 execute($\vec{v}$, d):
224 $barrier := \emptyset$
225 If CPREQ $\in \vec{v}$:
226 $barrier := (\{x \,|\, \forall r_i : x < exp(r_i)\} \bigcup_{v \in \vec{v}, v.req = \text{CPREQ}} deps(v) \cup v) \cap exp_k$
227 For $c \in sort(\vec{v})$:
228 If $c \neq \text{CPREQ} \wedge (barrier = \emptyset \vee c \in barrier)$:
229 Execute request c
230 $rhist[v] := d$
231 If $barrier \neq \emptyset$:
232 Create execution checkpoint with $barrier$
233 Restart request execution

As described in Section IV-G a replica broadcast a CHECKPOINT message after creating a checkpoint. Once a valid checkpoint is backed by at least $2f+1$ replicas, it becomes *stable*. This guarantees that the checkpoint is correct. To apply a checkpoint, a replica requests the set of $2f+1$ checkpoint messages along with the checkpoint content and applies the checkpoint after verifying the correctness of all messages.

The Validity and Consistency properties are not affected by applying a checkpoint as this does not affect agreement slots except by garbage collecting old ones. The Execution Consistency is also maintained as the execution state of a correct replica is applied. As soon as a correct replica has a stable checkpoint, all other replicas will eventually be able to learn about the checkpoint. This in turn allows all correct replicas to update their state if necessary.

The following Lemma adapts the proof of Theorem A.11 to handle checkpoint requests.

Lemma A.52. *For a checkpoint slot, if a checkpoint request certificate (CRC) is selected during a view-change then the slot did not commit previously.*

Proof: As shown in the proof of Theorem A.11 the new-view calculation always includes an FPC or RPC if either of both committed. Thus the CRC cannot be selected. ■

Lemma A.53. *All correct replicas create identical checkpoints for each checkpoint request.*

Proof: A checkpoint request conflicts with all other requests. This ensures that each request is either executed before or after the checkpoint request at all replicas due to the Execution Consistency property. In addition, this guarantees

that all replicas execute a checkpoint request as part of the same SCC. Thus, all correct replicas execute the same part of the SCC before creating a checkpoint. As all replicas execute the same set of requests before a checkpoint, exp_k is identical across replicas, and therefore all replicas bound the checkpoint barrier to the same slots (Line 226).

We now show that the checkpoint barrier is tight. Assume that a slot x before the checkpoint barrier was not executed. $exp(*)$ which is added as lower bound to the checkpoint barrier cannot add unexecuted slots. For a slot x to be covered by the checkpoint barrier, the checkpoint request must include a dependency on x or a slot $x' > x$. Then by Corollary A.25 the checkpoint request depends on x which therefore must be executed first.

Assume that a slot x not covered by the checkpoint barrier was already executed. That slot must have been executed as part of a regular SCC or an SSCC.

Assume that slot x was executed as part of an SSCC. The SSCC consists of at least 2 slots and therefore includes a dependency on the slot x. Therefore, the SSCC also depends all slots between $exp(x.i)$ and the slot x. Thus, after execution of the SSCC $exp(x.i) > x$. This yields a contradiction as the lower bound of the checkpoint barrier covers $\{x' \mid x' < exp(*)\}$.

Assume slot x was executed as part of a regular SCC. x must either depend on the checkpoint request or vice versa. When the checkpoint request depends on x, it also depends on all slot between $exp(x.i)$ and x. Therefore $exp(x.i) > x$ when the checkpoint is executed, which yields a contradiction. Now, assume x depends on the checkpoint request. Then x must be executed after the checkpoint or as part of an SSCC, which both yields a contradiction.

Thus all replicas create a checkpoint after executing the exact same set of requests. Applying the checkpoint yields the same state as a replica has after executing all requests up to the checkpoint. ■

We now show that applying a checkpoint or garbage collecting slots after a checkpoint is stable does not affect Execution Consistency.

Proof: Once a checkpoint is stable, all later requests will include dependencies on all slots included in the checkpoint. Compared to an execution without checkpointing this can only introduce additional dependencies. However, as all slots included in the checkpoint are already executed, these have no influence on the request execution. ■

The following Lemma adapts the proof of Theorem A.36 to handle checkpoint requests.

Lemma A.54. *For a checkpoint slot, if the slot did not commit previously then at least a checkpoint request certificate (CRC) is selected during a view change.*

Proof: The new-view calculation requires $2f + 1$ VIEWCHANGEs which are sufficient to generate a CRC (Line 219-221). As shown in the proof of Lemma A.42, eventually all correct replicas will send a VIEWCHANGE. These messages and their dependencies will eventually be accepted by `wait` allowing the view change to complete (Line 216). Once a CRC has committed via the reconciliation path, then it is handled as any other request. ■

We modify Lemma A.46 as follows:

Lemma A.55. *The calculation of a* NEWVIEW *completes for a timeout of* 5Δ *in synchronous phases.*

Proof: Once a correct replica has received $2f+1$ VIEWCHANGEs then within 2Δ every correct replica will receive $2f + 1$ VIEWCHANGEs. All VIEWCHANGEs are sent after Δ and are accepted at most 3Δ later, similar to Lemma A.45. This allows the view-change coordinator to calculate the NEWVIEW which after Δ arrives at all replicas. That is, in total a timeout of 5Δ is sufficient. ■

8) Spatial Complexity: The primary source of memory usage in ISOS are the agreement slots. If each coordinator maintains $2 * cp_interval$ slots, this yields a total of $N * cp_interval * 2$ slots. For each slot, ISOS has to store the attached request and the messages necessary to create a fast-path or reconciliation-path certificate. The dependency tracking in the agreement only has to track dependencies to slots explicitly contained in the compact dependency set, as that implicitly ensures that all earlier slots will also commit eventually. Thus, only up to N dependencies are tracked for a slot in the agreement. That is the memory usage is roughly determined by $O(N * cp_interval * (|r| + N * signature_size))$. This is similar to other protocols which forward certificates during the view-change.

As the execution only processes a window of k agreement slots for each coordinator and only a single dependency graph is necessary as described in Remark A.34, the dependency graph in the execution only contains nodes for up to $N*k$ slots. For each slot the request itself is still stored by the agreement, as otherwise it would have been garbage-collected from both the agreement and execution, and thus requires no additional memory. The execution also has to unroll the compact dependency sets and add explicit edges between requests to the dependency graph. Dependencies on already executed slots are not necessary for determining the execution order and thus are not stored. Dependencies to slots beyond the window of currently executed requests, can be expanded once these slots enter the window. This yields an upper bound of $(N * k)^2$ for the number of edges. The algorithm to calculate strongly-connected components uses a stack which in the worst case can contain all slots in the graph, that is up to $N * k$ elements. In our evaluation, we have used $N = 4$ and $k = 20$, which limits the number of edges to a few thousands. For these parameters, the memory usage for the execution is negligible compared to that for the agreement slots.