Geo-Replicated Byzantine Fault-Tolerant State-Machine Replication with Low Latency

—

Georeplizierte byzantinisch fehlertolerante Zustandsmaschinenreplikation mit niedriger Latenz

Der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg
zur Erlangung des Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Michael Eischer

Als Dissertation genehmigt
von der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	20. Dezember 2023
Gutachter:	PD Dr.-Ing. Tobias Distler
	Prof. Dr. Hans P. Reiser

Abstract

Protocols tolerating Byzantine faults allow a service to maintain correctness even if some of its parts misbehave. However, they significantly increase the overhead for processing client requests. Especially for geo-replicated systems, response times can grow to several hundred milliseconds. As the communication latency between remote locations is limited by the speed of light, minimizing these delays requires optimized protocols. This thesis investigates different approaches to reduce the response times of strongly consistent Byzantine fault-tolerant state-machine replication protocols that only require a low number of replicas in a geo-replicated setting. It first reviews the steps necessary to process client requests with strong consistency and derives approaches to minimize the delay introduced by the client communication, the agreement and the execution.

To minimize the latency for submitting a client request to the system, each replica is enabled to immediately initiate the ordering for a request, thus allowing a client to communicate with the nearest replica. The proposed Byzantine fault-tolerant egalitarian protocol, which only uses the minimum number of $3f + 1$ replicas to tolerate f faults, accordingly removes the need for a central leader replica by instead agreeing on conflicts between requests. The agreement can complete on a fast path if the involved replicas propose matching conflicts. Before execution, requests are sorted according to their conflicts to ensure a consistent order across replicas.

To minimize the latency of the agreement protocol, this thesis introduces an approach based on the architecture of modern clouds. The replicas are split into a central agreement group, which determines the execution order for all client requests, with at least $3f + 1$ replicas and multiple execution groups with only $2f + 1$ replicas. Each group of replicas is located in one region, with its replicas distributed across multiple availability zones to minimize the risk of correlated failures. Thus, replicas in a group can communicate with each other with low latency, thereby allowing the agreement to work with low latency. The groups use an abstraction called inter-regional message channel, which allows them to exchange client requests and the agreement results reliably.

To minimize the execution latency, this thesis reduces response-time spikes caused by the periodic creation of checkpoints during which the request execution must be paused. Instead, it introduces a concurrent state-capture phase, which starts before reaching the point at which to create the checkpoint. The resulting fuzzy snapshot together with a list of state modifications made by requests executed in the meantime, can be combined into a regular checkpoint that is identical on all replicas. The application interface offers two variants providing different trade-offs regarding simplicity and efficiency.

These approaches successfully reduce the client-perceived response times while also reducing the performance variation caused by different configurations and periodic tasks.

Kurzzusammenfassung

Protokolle, die byzantinische Fehler tolerieren, ermöglichen es einem Dienst sich korrekt zu verhalten, selbst wenn Teile von ihm Fehlverhalten zeigen. Allerdings erhöht dies den Aufwand zur Verarbeitung von Nutzeranfragen deutlich. Insbesondere bei georeplizierten Systemen, können die Antwortzeiten auf mehrere hundert Millisekunden steigen. Da die Kommunikationslatenz zwischen entfernten Orten durch die Lichtgeschwindigkeit beschränkt wird, sind optimierte Protokolle nötig, um diese Verzögerungen zu minimieren. Diese Dissertation untersucht Ansätze, um die Antwortzeiten von stark konsistenten, byzantinisch fehlertoleranten Zustandsmaschinenreplikationsprotokollen im Kontext von Georeplikation zu reduzieren, die nur eine geringe Anzahl an Replikaten benötigen. Dazu werden zuerst die für die stark konsistente Verarbeitung von Nutzeranfragen nötigen Schritte betrachtet und daraus Ansätze abgeleitet, die Verzögerungen durch die Kommunikation mit dem Nutzer, die Einigung und die Ausführung minimieren.

Zum Minimieren der Latenz für den Versand einer Nutzeranfrage an das System wird jedem Replikat ermöglicht, die Einigung einer Anfrage sofort zu starten, sodass der Nutzer mit dem nächstgelegenen Replikat kommunizieren kann. Das vorgeschlagene byzantinisch fehlertolerante egalitäre Protokoll, das nur das Minimum von $3f+1$ Replikaten zum Tolerieren vom f Fehlern benötigt, ermöglicht es dementsprechend ohne ein zentrales Anführerreplikat auszukommen, indem sich auf Abhängigkeiten zwischen Anfragen geeinigt wird. Falls die beteiligten Replikate die gleichen Abhängigkeiten vorschlagen, kann die Einigung im Schnelldurchlauf abgeschlossen werden. Vor der Ausführung werden Anfragen dann entsprechend ihrer Abhängigkeiten sortiert, um eine konsistente Ausführungsreihenfolge über alle Replikate zu garantieren.

Zum Minimieren der Latenz des Einigungsprotokolls stellt diese Dissertation einen Ansatz vor, der die Architektur moderner Cloud-Plattformen ausnutzt. Die Replikate werden in eine zentrale Einigungsgruppe, die die Ausführungsreihenfolge der Nutzeranfragen festlegt und aus $3f+1$ Replikaten besteht, und mehrere Ausführungsgruppen mit jeweils nur $2f+1$ Replikaten aufgeteilt. Jede Gruppe läuft innerhalb einer Region, wobei die einzelnen Replikate auf verschiedene Verfügbarkeitszonen verteilt sind, um das Risiko gleichzeitiger Ausfälle zu minimieren. Dadurch können Replikate einer Gruppe miteinander mit niedriger Latenz kommunizieren, was wiederum die Einigung mit niedriger Latenz ermöglicht. Die Gruppen nutzen eine als interregionaler Nachrichtenkanal bezeichnete Abstraktion, die den zuverlässigen Austausch von Nutzeranfragen und Einigungsergebnissen ermöglicht.

Zum Minimieren der Ausführungslatenz reduziert diese Dissertation Antwortzeitspitzen, die durch das periodische Erstellen von Sicherungspunkten hervorgerufen werden, währenddessen die Anfrageausführung pausiert werden muss. Stattdessen wird eine nebenläufige Zustandssicherungsphase eingeführt, die bereits startet, bevor der nächste

Sicherungspunkt erstellt werden muss. Der hierbei entstehende unscharfe Sicherungspunkt kann zusammen mit einer Liste von in der Zwischenzeit erfolgten Zustandsmodifikationen wieder in einen normalen Sicherungspunkt umgewandelt werden, der auf allen Replikaten übereinstimmt. Die Anwendungsschnittstelle bietet dabei zwei Varianten an, die sich hinsichtlich Einfachheit und Effizienz unterscheiden.

Diese Ansätze ermöglichen es die Antwortzeiten aus Sicht des Nutzers zu optimieren, sowie auch Leistungsschwankungen durch unterschiedliche Systemkonfigurationen oder periodische Aufgaben zu reduzieren.

Contents

1 Introduction

1.1. Motivation

Our world increasingly relies on computers to function, as more and more of our everyday life interacts with some online service. And correspondingly each year the number of people world-wide having access to the internet increases [200]. Be it communicating with others, shopping online, managing your finances or maybe filing taxes, nearly every part of our life has an online component.

As a consequence, we are becoming more and more dependent on these systems, which must offer their services reliably. However, at the same time as their importance increases, the incentives for attacks increase too. By now, nearly every day there are reports of another successful attack on yet another company [23]. Despite decades of efforts to secure computer systems, the increasing complexity has led to more and more chances for bugs to sneak in and thus a continuous stream of new security issues[1]. Some recent attempts to combat this problem rely on reducing the trust assumptions between different parts of a system, by requiring every participant to authenticate itself to the others and thereby allowing for checks whether a client or some service of the system is allowed to perform a certain action or not [205].

Actually, it is possible to reduce the necessary trust in a service even further by only allowing actions if *multiple* instances of a service support them. That is, an action initiated only by an individual, possibly faulty service instance is ignored until a sufficient number of other instances vouch for it. Such a service can then tolerate arbitrary (mis-)behavior of a limited subset of its replicas and is said to be Byzantine fault-tolerant [138]. Thereby, a service can continue to work despite intrusions that affect some of its instances. However, this level of fault tolerance comes at a cost: all requests to a service have to be replicated to at least $3f+1$ servers to tolerate up to f faults. To ensure that the replication protocol works in all cases, it also requires multiple rounds of communication between all replicas.

[1] https://www.cvedetails.com/browse-by-date.php

Before we take a closer look at these communication costs, we first present additional fault scenarios that should be tolerated.

Hardware Faults In rare cases, hardware faults can lead to wrong computation results due to bit flips in memory or the processor. The possible consequences range from program crashes [91, 107, 166, 172], data corruption [54, 77, 119, 168] to far-reaching outages where a single flipped bit propagates and causes the failure of a whole datacenter [11]. If a system stores cryptographically secured data like an audit log, a bit-flip could even permanently corrupt the data structures and require the system to be replaced [25]. While such hardware faults are usually rare, reports [77, 119, 168] from companies operating very large datacenters indicate that "Memory corruption is common at scale" [168]. Although the problem can be partially mitigated in hardware or software, Byzantine fault tolerance offers a more comprehensive solution as it is general enough to also handle these faults.

Datacenter Failures Faults at the level of individual components are not the only problem that threatens the availability of a service. Assume for a moment that all components of a service run in a single datacenter. Then the failure of that single datacenter, for example due to infrastructure issues like failed power supplies or damaged network connections, natural disasters like flooding or thunderstorms [35], or a fire in a datacenter [176], can render the whole service inaccessible. The solution is to replicate a service geographically so that it is replicated across multiple datacenters that are located sufficiently far apart to not be affected by the same disaster. That way, a local disaster only affects one of the datacenters and allows the service to continue working.

Response-Time Expectations

For services whose users are distributed across the whole globe, there is another reason to geographically distribute the service. Instead of requiring users to communicate with a service running somewhere far away, which consequently results in high latency, it would be preferable to bring the service instances closer to the users by distributing the instances. That way, users can contact the service instance offering the lowest latency, which is likely one that is also geographically located nearby.

Such response time improvements of services are considered as important especially in the context of e-commerce. Large companies, for example Amazon, have observed that a 100 millisecond increase in page load time, can lead to a 1% decrease of sales [104]. Similar observations were made in a more recent report by Akamai [197] that indicates even higher losses or by Google where a half second increase in the load time of the search results lost 20% visitors [105]. Especially in the wide-area context these response time constraints become problematic as individual requests to a service can already take a few hundred milliseconds which adds up in case multiple requests become necessary. As Byzantine fault-tolerant services typically even require multiple communication rounds, this poses the challenge of improving the response time for requests.

Network Capacity

With the rise of cloud computing it became relatively easy to run a service with instances spread across the globe by renting virtual machines [19, 101, 159]. The cloud providers take care of managing the hardware and in particular the underlying network infrastructure which connects the datacenters in the different regions. The underlying network connections are increasingly owned or rented by the providers themselves [8, 100], which gives them control over the communication backbone and enables them to optimize the communication between regions [120, 124]. This helps with satisfying the ever-increasing demand for more bandwidth between the datacenters [120]. In respect to Byzantine fault-tolerance and its larger amount of necessary communication, the growth of the available bandwidth partially mitigates the higher communication overhead.

In contrast, the communication latency presents a much harder challenge as it is fundamentally limited by the speed of light, which sets a strict lower bound for how fast information can be transferred between two locations [186]. Making matters worse, inefficiencies in hardware, software and protocols increase the gap between the physical lower limit and the actual latency. On the hardware level there is still room left for the providers to optimize latency by adding more direct connections between datacenters, by using fiber links with better cables that allow light to travel slightly faster and by optimizing every hop the data has to pass through [106, 186]. But unlike network throughput bottlenecks, which can be solved by the cloud providers by adding more capacity, latency remains ultimately bounded by the distance data has to travel. Thus, further improvements of the protocols managing the data replication are necessary by optimizing the overall system structure or the used message patterns.

1.2. Purpose of this Thesis

As previously discussed, users expect services to react quickly, which requires protocols that can process requests with low response times. The goal of this thesis is to investigate different approaches to reduce the response times for strongly consistent Byzantine fault-tolerant replication protocols that run in a geo-replicated setting while also using low numbers of replicas. The latter is beneficial, as using more replicas directly translates to higher costs for running the service in the cloud. Firstly, increasing the number of servers increases the costs for those. And secondly, more servers also mean a higher amount of replication traffic, as all requests and state changes have to be shared with every replica. Thus, we want to keep the number of replicas close to the required minimum. Additionally, the service should provide strong consistency, such that from the client perspective it behaves like a single central server.

To keep the replicas of a service in sync, we use a state-machine replication protocol, which is a generic approach to replicate applications. A state machine has a state which is modified by processing requests [179]. The protocol roughly speaking proceeds in three main steps: the client communication over which the clients' requests arrive at the replicas, then the core of the replication protocol, which distributes the requests among the replicas ensuring that all requests arrive at all correct replicas in the same order, and

finally the actual request execution using the state machine. Each of these steps can significantly contribute to the overall latency.

Client Communication

State-machine replication protocols often choose one of the participating replicas as leader [22, 24, 39, 43, 47, 57, 62, 73, 110, 122, 130, 141, 189, 210, 211]. That replica plays a central role in the protocol as all client requests are sent to the leader, which is then responsible for suggesting a request order that must be confirmed by the follower replicas in multiple rounds of communication. Consequently, the geographical location of the leader has a major influence on the latency until the client request arrives at the leader and on the duration of the actual replication. A leader with high communication latency to each client will impact latency even if it can quickly order a request. And conversely a leader with low communication latency to the clients might be unsuitable to quickly order requests when this would result in high communication latencies with the other replicas. Besides the leader location, latency is also determined based on the individual clients' location in relation to the leader. That means different clients might benefit from different leader replicas [82].

Several protocols try to decrease the time it takes for a client request to reach the leader by rotating the leader replica across different locations [161, 201, 202]. This allows a client to submit its request to a nearby leader replica. However, sharing the leader role requires the replicas to coordinate and wait for each other. This can result in higher latencies than a carefully selected leader replica [189].

Agreement Latency

A large part of the response time is determined by the communication steps between replicas to agree on a request order. Many protocols sketch this message flow with evenly spaced protocol phases [22, 39, 47, 57, 73, 130, 141, 189], which works well for local-area networks as these provide largely uniform latencies between replicas. However, as shown by Archer [82] and WHEAT [189] this does not match wide-area environments with their highly non-uniform latencies between different regions, which can vary between a few milliseconds to hundreds of milliseconds [6]. Several approaches try to take advantage of this latency structure to reduce the latency for the ordering step by determining an optimal leader location either statically [189] or dynamically [43, 82] based on feedback provided by clients or the measured latency between replicas. Other protocol structures go one step further and place groups of replicas at each location [20, 24], allowing communication over low-latency links within a group. However, these still require multiple wide-area communication steps and thereby leave room for further optimizations.

Execution without Latency Spikes

After ordering requests they have to be executed. This is often done sequentially, that means, one request is executed after the other, but it is also possible to speed up the process by parallelizing the execution [60, 131, 142]. In any case the depiction of a replication protocol often does not present an extra phase to execute requests [22, 43, 47, 57, 73, 130, 141, 189]. For most requests this is a good approximation, as the request execution usually only takes very little time, but this is not always the case. Each

protocol has to create checkpoints periodically, which allow the replicas to forget all earlier requests [46, 57, 59, 62] and can be used by lagging replicas to catch up. While creating a checkpoint, the request execution must be paused to allow the replica to capture a consistent snapshot of the application state. Depending on the size of the application state, this can result in significant delays during request processing and thus drastically increase the amount of time a client has to wait for a reply. Although it is possible to reduce the amount of work done during snapshot creation by only copying changed data [57, 59], this can still lead to significant delays.

Approach of this Thesis

In this thesis we propose to optimize the response time of a service by individually reducing the latency of each one of the three basic protocol steps, that is client communication, agreement and execution as close as possible to zero. Depending on the optimized step the resulting protocols offer drastically different trade-offs. This general idea leads to the three following approaches.

- Replication protocols often involve the usage of a leader replica, which is responsible for proposing a request order that has to be confirmed by the other replicas. For clients that are not located at the leader's location the request submission requires wide-area communication resulting in increased latency. We propose to design a protocol that allows each replica to independently initiate the ordering of new requests. By relying on commutativity between non-conflicting requests, replicas can propose an order without requiring coordination with all other replicas. This enables clients to submit the request to a nearby replica and thereby avoid the wide-area communication costs.

- Determining an order for requests usually requires three communication steps for Byzantine-fault tolerant protocols like PBFT [57]. For a geo-distributed system with replicas spread across the globe this results in a high latency due to the necessary wide-area communication. We propose to use a feature commonly offered by modern cloud infrastructures: availability zones [14, 101, 159], which are designed to be as fault independent of each other as much as possible. Each region consists of multiple availability zones, therefore allowing the ordering step to run in a single region without compromising the availability of the service. Thus, the system is able to avoid wide-area communication while ordering requests.

- A replication protocol has to periodically discard already executed requests to limit the size of its state. This is commonly done by creating a checkpoint which can be used to continue executing at the corresponding point in time. For larger application states this checkpoint creation can lead to significant delays during request processing. We propose to capture the application snapshot concurrently while the application is actively processing requests to reduce execution stalls. As a consequence we need a mechanism to handle such a fuzzy checkpoint and convert it into a consistent one. In total, this reduces the processing delays incurred by creating a checkpoint.

1.3. Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2 introduces the system model used throughout this thesis, along with the special characteristics of wide-area networks and the necessary background on distributed state-machine replication that serve as a basis for the remainder of this thesis. It then reviews the state of the art relevant for our approach.

Chapter 3 analyzes existing approaches and highlights their problems with providing services with low latency. Afterwards it presents our suggested approach in further detail and formulates the associated central questions and design goals guiding the system design.

Chapter 4 introduces ISOS, which allows all clients to submit their requests to the nearest replica. Each replica can propose requests concurrently, which are ordered based on conflicts between requests. Non-conflicting requests can use a fast path resulting in response times similar to submitting the request to a local leader replica.

Chapter 5 shows how to use the structure of modern cloud infrastructures to maintain resilience while providing low response times. The SPIDER architecture consists of multiple loosely coupled replica groups which can consistently execute requests. It also allows clients to bypass the replication protocol to read slightly outdated data from nearby replica groups in exchange for much lower latency. We describe an abstraction that is used to transmit requests between the replica groups and enhance the approach with several optimizations.

Chapter 6 presents Deterministic Fuzzy Checkpointing (DFC), a mechanism which collects a fuzzy application snapshot without having to pause request execution and thereby prevents latency spikes. It presents the required application interface and details how a collected fuzzy checkpoint is made deterministic again. This is complemented by a method to reduce the amount of data that has to be copied to collect a checkpoint.

Chapter 7 summarizes this thesis and discusses possible directions for future work.

1.4. Related Publications

Parts of the results of this thesis are based on the following publications:

[81] Michael Eischer, Markus Büttner, and Tobias Distler. "Deterministic Fuzzy Checkpoints." In: *Proceedings of the 38th International Symposium on Reliable Distributed Systems*. SRDS '19. 2019, pages 153–162. DOI: 10.1109/SRDS47363.2019.00026.

[82] Michael Eischer and Tobias Distler. "Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems." In: *Proceedings of the 1st Workshop on Byzantine Consensus and Resilient Blockchains*. BCRB '18. 2018, pages 140–145. DOI: 10.1109/DSN-W.2018.00053.

[83] Michael Eischer and Tobias Distler. "Efficient Checkpointing in Byzantine Fault-Tolerant Systems." In: *Tagungsband des FB-SYS Herbsttreffens 2019*. 2019. DOI: 10.18420/fbsys2019-01.

[85] Michael Eischer and Tobias Distler. "Resilient Cloud-Based Replication with Low Latency." In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. 2020, pages 14–28. DOI: 10.1145/3423211.3425689. (Best student paper).

[86] Michael Eischer and Tobias Distler. *Resilient Cloud-based Replication with Low Latency (Extended Version)*. arXiv. 2020. DOI: 10.48550/ARXIV.2009.10043.

[87] Michael Eischer and Tobias Distler. "Egalitarian Byzantine Fault Tolerance." In: *Proceedings of the 26th Pacific Rim International Symposium on Dependable Computing*. PRDC '21. 2021, pages 1–10. DOI: 10.1109/PRDC53464.2021.00019.

[88] Michael Eischer and Tobias Distler. *Egalitarian Byzantine Fault Tolerance (Extended Version)*. arXiv. 2021. DOI: 10.48550/arXiv.2109.06811.

[89] Michael Eischer, Benedikt Straßner, and Tobias Distler. "Low-Latency Geo-Replicated State Machines with Guaranteed Writes." In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '20. 2020. DOI: 10.1145/3380787.3393686.

Chapter 4 is in parts based on [87] of which I was the leading author, developed the main ideas and was the major contributor to its implementation, evaluation and writing. Appendix A presents a partially revised version of the proof included in the extended version [88] whose author I was.

Chapter 5 is in parts based on [85] which I was the leading author of, and the major contributor to its design, implementation and evaluation. Its proof, which was presented in the extended version [86], of which I was the main author, is included in partially revised form in Appendix B.

Chapter 6 is based on the results of [81, 83] whose leading author I was and the major contributor to its design, implementation and evaluation.

Ideas from [82, 89] have influenced various parts of this thesis in particular Sections 2.4.1, 4.7.5 and 5.7.4. I was the major contributor to the design, implementation and evaluation of [82]. And I was one of the two main authors contributing to the writing in [89] and developed the main ideas along with their implementation and evaluation.

2
System Model, Background and State of the Art

Clients expect a service to work reliably. To tolerate faults of the servers that provide the service, it has to be replicated. For clients ideally the service appears to behave just like a central implementation running on a single server would. The resulting system is based on the following basic design. The service is replicated onto multiple replicas with which clients interact via network by sending a request and waiting for replies from a sufficient number of replicas. Then the replicas use a replication protocol to distribute and agree on an order for all client requests. Finally, all replicas deterministically execute the requests in the agreed order to provide the clients with a consistent reply [179].

We start with providing more details on the expected environment in Section 2.1 and on the characteristics of wide-area networks in Section 2.2. Afterwards Section 2.3 presents a slightly simplified variant of such a replication protocol. Section 2.4 reviews relevant state of the art and Section 2.5 concludes the chapter.

2.1. System Model

In the following we introduce the basic assumptions used throughout this thesis regarding the behavior of replicas and clients in Section 2.1.1. These are connected via the internet for which we discuss the expected behavior in Section 2.1.2. Afterwards we review the system properties in Section 2.1.3 and the possible types of faults along with their theoretical limits in Section 2.1.4. Next, Section 2.1.5 presents the cryptographic primitives used for this thesis. Section 2.1.6 concludes with the application model.

2.1.1. Nodes

The service is replicated onto multiple servers, the so-called *replicas*, which are hosted at one or multiple locations either as virtual or physical machines. We also refer to locations

as *sites*; both terms are used interchangeably. The set $R = \{r_1, r_2, ..., r_N\}$ contains all replicas with N being the total number of replicas. All replicas know their identities and can communicate with each other via a local network or the internet.

The *clients*, which issue requests to the service, are typically collocated at the same sites as the replicas or somewhere nearby. Each client must be able to communicate with every replica. When referring to both clients and servers, we call them *nodes*.

Nodes that properly follow the protocol are *correct*, whereas those that deviate or crash are called *faulty*. We review the expected fault types in more detail in Section 2.1.4.

To allow all replicas to keep up with each other, the processing speed of replicas should only differ by a small factor. The timeouts used in the replication protocols also expect the clocks of correct replicas to advance at similar rates. Modern cloud infrastructures offer time servers in each location that use atomic clocks and the globally synchronized time signal from global positioning system (GPS) satellites to provide accurate timestamps [13, 103, 156]. Replicas can synchronize their clocks using those servers and thereby fulfill the previous requirement. We only rely on synchronized clock speeds for performance, but not for correctness.

2.1.2. Network

The communication between all nodes runs over the internet, which can be modelled using *partial synchrony* [57, 80]. In that model the network alternates between synchronous and asynchronous phases. During synchronous phases messages sent by a node arrive at the destination node within a bounded amount of time Δ. In contrast, during asynchronous phases there is no upper limit for the transmission delay. The network can switch between these phases at arbitrary points in time that are not known to the nodes. This is a good match for the behavior of the internet, which normally delivers packets after a short delay except in case of network interruptions, which either get repaired or routed around after a short time [57]. We assume that the replicas know an approximation of the *maximum one-way communication delay* Δ. It can be used to derive appropriate protocol timeouts. To allow the protocols to make progress, the network must provide sufficiently long synchronous phases.

Message Loss

The network may also drop, delay, reorder or damage messages with the restriction that a message that is sent repeatedly must eventually arrive at the receiver. To transparently compensate for these transmission problems, reliable communication channels are used between nodes. These channels automatically retransmit lost messages until they arrive. A commonly used protocol to handle the majority of retransmissions is transmission control protocol (TCP) [78], which provides a reliable first-in-first-out (FIFO) communication channel between nodes. Only in case the TCP connection is interrupted, for example due to a too long period of message loss, then it becomes necessary to explicitly retransmit potentially lost messages over a new connection. Section 5.6.2 describes a possible approach to handle these retransmissions.

2.1.3. Safety and Liveness

A replicated service should provide the *safety* and *liveness* properties. The following definitions are based on those used by PBFT [56].

Safety For a correct client the service behavior must be indistinguishable from a central service implementation that executes requests one after another.

Liveness A correct client will eventually receive a reply to its request once the system is in a sufficiently long synchronous phase.

The safety property requires the service to be *strongly consistent*, or more formally to guarantee *linearizability* [118], with the modification that this guarantee can only be provided for correct clients, as faulty clients can deviate arbitrarily from the protocol. From the client perspective, this property provides the guarantee that like a centralized service implementation once it has received a reply to a request, all later requests will work on an application state that includes the effects of that previous request. This guarantee also extends to processed requests from all other clients.

The liveness property ensures that the service stays available while the system is in a synchronous phase. Fischer et al. [93] have shown that if at least one replica can fail, then any deterministic agreement protocol can only guarantee progress during sufficiently long synchronous phases. That is, a deterministic protocol can only provide both properties while the network is synchronous, otherwise we temporarily give up liveness in favor of safety [97]. We do not consider randomized algorithms in this thesis, as for these an ordering step only completes with a certain probability and thus can require multiple retries resulting in an increased latency.

2.1.4. Fault Assumptions

Clients and replicas may be subject to *Byzantine faults*. This class of faults, defined by Lamport et al. [138], allows clients and replicas to arbitrarily deviate from a given protocol or even behave maliciously. In the simplest case a node just crashes and stops reacting to messages and no longer sends anything. But it is also possible for a Byzantine node to send invalid messages, manipulated ones, or to be duplicitous and send different, contradictory messages to different replicas. Other possible misbehavior includes sending specific messages only to some replicas or omitting them altogether. A faulty node can also adhere correctly to the protocol most of the time and only deviate from it at specific points in time, for example, when the node could reap some benefit from doing so without being detected [147]. Faulty replicas can collude with each other allowing them to coordinate their attacks. This resembles an attacker that has compromised multiple nodes. Therefore, nodes cannot trust each other and instead have to verify information using messages from a sufficient number of nodes.

Limits on Faults

A fault-tolerant *agreement protocol* enables the replicas to reliably agree on a totally ordered log of requests. That is, the result at each replica contains the same requests

in the same order. However, such a protocol is only able to mask the misbehavior of a limited number f of faulty replicas. As shown by Bracha et al. [49], at least $3f+1$ replicas are required to tolerate f Byzantine faults. The systems we consider in this thesis are focused on tolerating a small number of faults, typically at most 3. To prevent conflicting ordering decisions, a replica proposing a request ordering has to receive confirmations from a *quorum* of replicas Q that is large enough to intersect with any another quorum of replicas Q' that could be used to order other messages [57, 148]. *Disseminating quorums* [148] require this intersection to always contain at least one correct replica, such that this replica can prevent conflicting decisions. That is, the size of two quorums must be at least as large as the number of replicas plus the intersection: $2 \cdot |Q| \geq N + (f+1)$. To be able to guarantee liveness, a quorum must also not require the participation of faulty replicas, that is, $|Q| \leq N - f$. Combined these conditions result in a minimum quorum size of $|Q| = \lceil \frac{N+f+1}{2} \rceil$ using $N \geq 3f+1$ replicas.

Faulty Clients

Different from the replicas we assume no limit on the number of faulty clients. Nevertheless, the service must guarantee that correct clients receive correct replies. That is, it must be impossible for a faulty client to prevent correct clients from using the service or to cause them to receive a wrong result.

Note that the replication protocol cannot prevent a faulty client from issuing syntactically valid requests, which misuse the application semantics, for example to delete data. Such requests must be handled by application-specific means like access control mechanisms, which can limit operations based on the client's identity [57].

Causes of Faults

In order to guarantee limits on faults, replicas must fail independently. Hardware problems causing various kinds of data corruption [77, 119, 168] can in rare cases result in individual replicas becoming faulty, consequently it is even less likely that multiple replicas are affected at the same time.

Another source of faults are bugs in the replica implementation or the operating system which allow an attacker to gain control of the replica. Several approaches have been suggested to address this problem which include n-version programming [33] to create multiple independent implementations that are expected to fail in different ways, the usage of diverse commercial of-the-shelf (COTS) software at the replicas like different database implementations or operating systems [59, 95, 96], automatically generating variants of a software [140] or using a verified implementation [115, 208]. Thus, we consider this problem orthogonal to this thesis.

For simplicity, we assume that a faulty replica stays faulty permanently. In order to allow the service to tolerate a higher number of failures during the lifetime of the system, it would be necessary to recover faulty replicas such that the fault threshold is never exceeded at any point in time. We expect faults to be rare, thus giving a recovery mechanism sufficient time. Existing approaches like proactive recovery [58, 190], which periodically rebuilds replicas, could be used for this task.

Fault Detection
While we usually talk about a specific replica being faulty when discussing fault scenarios, there is generally no externally visible indicator available that other replicas can use. Another replica r_a might in some cases learn that a replica r_b is faulty by examining messages received from that replica. But even then, it might not be possible to convince another replica r_c that this is the case. After all, from the perspective of replica r_c it could just as well be replica r_a that is faulty and tries to spread rumors. If a replica r_a does not react to a message, it is impossible to determine whether the message already arrived and was just ignored by the replica r_a, or whether the message was delayed or lost on the network. Thus, Byzantine faults of a specific replica cannot be detected reliably [111, 112, 129]. This must be kept in mind when designing the subprotocols to recover from faults.

2.1.5. Cryptography

Messages exchanged between nodes must be authenticated to allow the receiving node to verify that the message content has not been tampered with and that it was sent by a certain node. A node may only make permanent changes based on a message after verifying that it is correctly authenticated, otherwise the message must be discarded. For a message $\langle \ldots \rangle_{auth}$ we use the suffix $auth$ to indicate how it is authenticated. The authentication data also includes the sender's identity. Clients and replicas must be provided with the necessary keys to create and verify authenticated messages either during setup or have the means to retrieve them using a key exchange mechanism [44, 58]. In this thesis we use the following authentication types:

MAC A message $\langle \ldots \rangle_{\mu_{i,j}}$ is authenticated using a message authentication code (MAC) from replica r_i for replica r_j. Computing and verifying a MAC for a message uses a symmetric key that is only known to sender and receiver.

MAC Authenticator $\langle \ldots \rangle_{\alpha_{i,\mathcal{A}}}$ indicates a MAC authenticator [57] containing a list of MACs from replica r_i for each replica $r_k \in \mathcal{A}$ that allows them to verify the message.

Signature $\langle \ldots \rangle_{\sigma_i}$ is a signature from replica r_i, which all other replicas can verify using the public key of replica r_i.

Each authentication type offers different guarantees and has different computational costs. MACs can be computed much faster than signatures [57]. However, unlike the latter they cannot provide non-repudiation, that is, a receiver r_b cannot prove to some other replica r_c that a certain message was created by sender r_a, as creating and verifying MACs requires a symmetric key. Being able to verify a MAC also allows computing it; therefore, the symmetric key must only be known to sender and receiver, which prevents any third party from verifying the correctness of a MAC.

In contrast, signatures use keys consisting of a public and a private key [152]. The private key is only known to the sender, which ensures that only this node can create the signature. The corresponding public key can be distributed to all other nodes and

allows them to verify the correctness of the signature. Therefore, signatures provide non-repudiation, as a replica r_a that receives a valid, signed message m from replica r_b can forward it in order to prove to another replica r_c that message m was sent by replica r_b.

Additionally, we require a cryptographic hash function $h(m)$ that for a message m computes a *digest* d [153] to which we also refer to as *hash*. It must provide collision resistance, that is, it must be virtually impossible to find two arbitrary messages m and m' with identical digest [153]. This allows referring to a message m using only its hash $h(m)$, as an attacker cannot create a fake replacement message m', which is useful to reduce the size of some messages.

Furthermore, we make the standard assumption that an adversary is computationally bounded and cannot break the used cryptographic primitives.

2.1.6. Application

Like other replication protocols we expect the application to behave like a *deterministic state machine* [179], which processes requests, modifies its state accordingly and generates a reply. The execution of each request must be deterministic. This ensures that executing the same totally ordered list of requests, on all replicas results in the same changes to the application state and provides the same replies to all clients. The model is generic enough to represent most applications.

The *application state* consists of a set $\mathcal{O} = \{o_1, o_2, ..., o_n\}$ of small *state objects* which each are uniquely identifiable using an arbitrary object identifier. Each object must be readable and writable by the replication protocol. For example, for a key-value store each entry could be represented by an individual object using the key as object identifier. The set of objects is dynamic and can change over time.

We differentiate between read requests, which do not modify anything, and write requests, which do. This can allow read requests to be executed without prior coordination between replicas. To prevent faulty clients from issuing write requests marked as read requests, the application must offer a method to check whether a request is guaranteed to only read data.

2.2. Wide-Area Environment

Compared to a system consisting only of replicas running in the same local network, the geo-replicated systems considered in this thesis are affected by the special properties of wide-area networks. In the following we discuss properties of modern cloud networks in Section 2.2.1, trade-offs like differing latencies between pairs of replicas in Section 2.2.2 and considerations for geo-distribution in Section 2.2.3. We conclude by reviewing attacks at the network level in Section 2.2.4.

2.2.1. Cloud Networks

With the rise of cloud computing, several providers became large enough to provide datacenters in dozens of regions in which users can rent resources like virtual machines [19,

101]. The providers not only manage the servers in the datacenters, but also the network between the regions [8, 50, 92, 100]. Data transmissions across the internet are often routed via transit networks [28] with all the effects of changing communication links, unstable performance and congestion. In contrast, the cloud networks are optimized for data transmission between datacenters [120, 124], and offer the chance for more reliable communication and lower latency than transmissions routed via the public internet.

With control over the network, the providers can adequately scale the available wide-area capacity to meet the demand. As shown by Lai et al. [136], the network bandwidth available to customers is rather determined by limits enforced for each virtual machine instance than by limitations of the cloud provider's wide-area network.

This level of control also helps to reduce the amount of packet loss. The latter is particularly important for wide-area communication as connections with high round-trip times are more sensitive to packet loss, because it takes much more time until the receiver notices the problem and can inform the sender. Although transmission protocols like TCP optimize for this case by only selectively retransmitting lost packets [78], packet loss could still significantly delay network transmissions. Measurements of cloud networks have shown a much lower amount of packet loss than normal connections over the internet [114], which drastically reduces the impact of packet loss on wide-area communication within the cloud. Link failures between cloud regions are typically also rare and tend to be resolved within minutes [90, 145].

That is, the cloud providers offer reliable network connections between virtual machines with high network bandwidth. However, this leaves the problem of communication latency for which there is only limited room for optimization by the providers.

2.2.2. Trade-Offs

The most obvious trait of wide-area networks is that communication can take a long time. Round-trip times between servers at distant geographic regions now range from dozens to hundreds of milliseconds as opposed to sub-millisecond latencies within a datacenter. Due to the much higher latency than in a local-area setting several trade-offs change. We briefly discuss the three most influential effects of the higher latency.

For local use cases, cryptographic operations that take several hundred microseconds may be prohibitively expensive, but compared to the communication latency in wide-area networks such operations are barely noticeable. That is, more expensive but also more powerful cryptographic operations like signatures instead of the cheaper MACs have a much smaller relative impact on response times.

The other way around, in a local-area network it might be enough to process requests one after another to keep the whole system busy, whereas in wide-area networks when the communication latency far exceeds the per-request computation, *latency hiding* by concurrently processing multiple requests becomes necessary.

Another differentiating factor is that the round-trip times between various regions are no longer uniform, as shown in Figure 2.1, but can differ by more than one hundred milliseconds. In contrast, in a local environment all replicas might be connected to the same switch, providing uniform communication latencies. But for wide-area networks

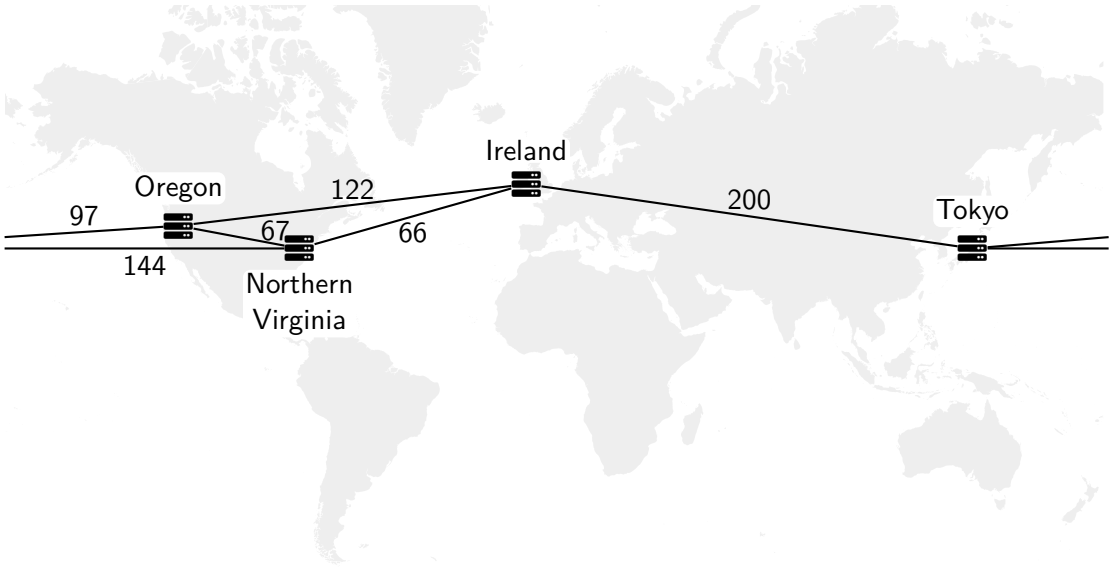


Figure 2.1: Round-trip communication latency in milliseconds between replicas in selected Amazon EC2 regions

the communication latency is partially dictated by the distance between replicas. In particular, this make it essentially impossible to provide uniform latencies between four or more replicas[1]. Depending on the latency structure, these non-uniform round-trip times result in varying progress among replicas. A replica may be lagging behind the other replicas by several requests or always be the first one to process a request. Thus, a replication protocol has to take the non-uniform round-trip times into consideration.

2.2.3. Geo-Distribution

We expect clients to be either located at or near the regions in which the replicas are hosted, thus the clients are also distributed worldwide. This can result in workload changes over time due to changing activity, for example, during the course of a day.

If a service just ran at a central datacenter, then all clients would have to direct their requests there, essentially resulting in the datacenter becoming a bottleneck. The main benefit of this approach is that the centrally located replicas are able to communicate with each other with very low latency. However, this comes at the price that a failure of the central datacenter will make the service unavailable. Power failures, software failures or natural disaster like thunderstorms, flooding or fires are just a few of the reasons why a datacenter can fail [12, 15, 35, 70, 176]. Such disasters are usually limited to a single datacenter or in rare cases an entire region, but (normally) do not affect whole continents

[1]With four replicas, it would be necessary for them to form a tetrahedron, which essentially requires one of the replicas to reside at the North or South Pole. Five or more equidistant replicas cannot be represented in three-dimensional space.

at once. Consequently, a system replicated across multiple regions has the potential to tolerate datacenter failures.

A geo-distributed variant of the service would place replicas in a set of datacenters that roughly matches the clients' distribution. This can significantly reduce the maximum communication latency to the client by providing the clients with a nearby datacenter they can use to access the service. Especially for service operations that only require interaction with a single nearby region, this can drastically improve latency. However, there is a trade-off: moving the replicas closer to the clients inevitably increases the costs of keeping the replicas synchronized, which now have to communicate over wide-area links with high latency, resulting in an increase of the overall latency. Thus, it becomes necessary to balance these competing requirements.

We assume that all replicas contain a full copy of the data and do not consider a setting where regulations require data to reside in specific geographic regions.

2.2.4. Network Attacks

We do not consider denial-of-service (DoS) attacks, which flood the system with useless requests. The world-wide connectivity of cloud providers provides a first line of defense to mitigate large floods of unsolicited traffic. On the replica side, filtering out faulty messages as fast as possible by applying cheaper checks first before running more expensive ones like signature verifications [63] can help to partially mitigate such attacks. The network communication must also differentiate between replicas and clients, which through appropriate scheduling can get a fair share of the processing time [63].

Network Trustworthiness

The network within a cloud datacenter is fully controlled by the cloud provider, whereas the communication between datacenters can run over provider-owned fibers or use rented fiber connections [92]. To protect this inter-datacenter communication all traffic between cloud regions is encrypted [19, 99], thus as long as the provider can be trusted we can assume that no tampering with network communication between replicas takes place. Therefore, we assume that no targeted manipulation takes place during transmission. Messages between nodes must nevertheless be authenticated to verify their senders' identity. As additional protection, the replicas could set up reliable and authenticated channels between each other, for example, by using transport layer security (TLS) [174], to reduce the trust required in the cloud provider.

2.3. State-Machine Replication

In the following we introduce the architecture of an exemplary Byzantine fault-tolerant state-machine replication protocol and its components. It serves as a starting point for our approaches to optimize the client communication, agreement, or execution, which each redesign different parts of the state-machine replication protocol. The description here presents a derivative version of the PBFT protocol using signatures [57] and serves as a basic structure for the later chapters.

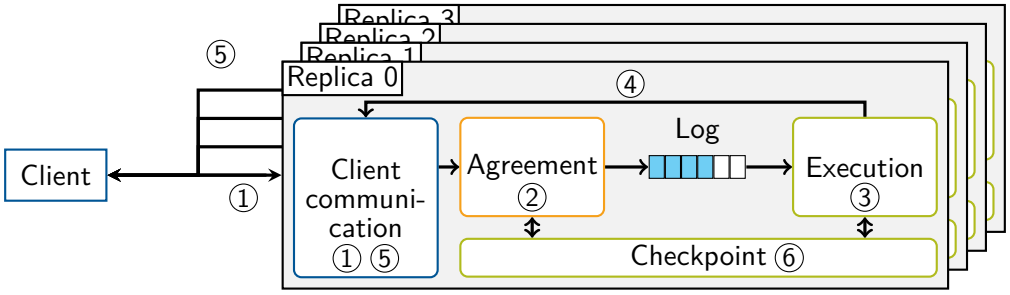


Figure 2.2: High-level overview of the replica components involved in replicating and executing client requests.

We first give an overview in Section 2.3.1, before describing the client in more detail in Section 2.3.2 and the replicas in Section 2.3.3.

2.3.1. Overview

A state-machine replication protocol generally consists of at least the components shown in Figure 2.2. The main idea is to establish an order on the client requests by assigning them to sequence numbers and then to execute the requests in this exact order.

The client first sends its request ① to the replica that is responsible for initiating the agreement protocol. This replica then proposes to assign the request to the next unused sequence number ②. After several communication steps between the replicas, three in case of PBFT, that sequence number assignment becomes permanent, which guarantees that it will not change in the future, even in case of faults. The resulting log of requests is then passed on to the execution ③, which processes the request in the order given by the sequence numbers. The application executes the request, updates its own state and also generates a reply message ④, which is then sent back to the client ⑤. As a single reply could also originate from a faulty replica, the client has to wait until it receives $f + 1$ matching replies to ensure that at least one of them is from a correct replica and thereby correct. Each replica also includes a checkpointing component ⑥, which allows the replicas to periodically garbage collect old requests to bound the size of its state.

Certificates

Due to the Byzantine fault model, replicas and clients cannot simply accept messages at face value, as their communication partner might be faulty. Even with cryptographically authenticated messages for which a node can accurately determine the sender, there is still the problem that a node can issue false claims. To handle those, nodes assemble so-called *certificates* [56] that consist of a certain number of matching claims from different replicas. Depending on the number of supporting claims, a certificate can be used to either prove the correctness of a value or that a quorum of replicas supports it. Once a node has collected a certificate with sufficient valid and matching messages, we say that the certificate is *stable*. The check whether messages are matching, unless specified otherwise, covers all fields of a message including the message type but excludes the

sender-specific message authentication data. The latter includes the sender identity, which is used to ensure that a certificate only includes a single message per sender.

To verify the correctness of a reply, a client has to collect a *weak certificate* consisting of $f + 1$ matching replies from different replicas. This set of messages then includes one reply which was sent by a correct replica and hence must be correct.

In order to prevent conflicting decisions in the agreement, a replica has to collect a *quorum certificate* consisting of $\lceil \frac{N+f+1}{2} \rceil$ matching message from different replicas. For the typical $N = 3f + 1$ this requires $2f + 1$ messages. The size of this quorum ensures that it overlaps with any other quorum in at least one correct replica which will then be able to transfer information between quorums. As the quorum size is at most $N - f$, it is possible to reach the quorum using only correct replicas. This ensures that faulty replicas cannot prevent the quorum formation.

2.3.2. Client

In order to send a request with *command* w to the service, a *client* c proceeds as follows. It constructs a signed message $\langle \textsc{Request}, w, t_c \rangle_{\sigma_c}$ and sends it to the *leader* replica, that is, the replica that is currently responsible for ordering new requests. If the selected replica is no longer the leader, then it has to forward the request accordingly. The counter value t_c represents a client-specific counter that must increase for each new request, for example by using a sequential counter. The replicas guarantee that a request is only executed once by using the counter value to determine whether a request is a retransmission of a previous one or a new one which should be ordered and executed.

Client requests use a signature to allow the replicas to unambiguously check the correctness of the request and provide the application with the client's identity. This prevents the creation of fake requests and avoids complex corner cases that can occur when using MAC authenticators [56].

After the request was processed, all replicas send a reply to the client, who has to collect a (weak) reply certificate consisting of $f+1$ matching $\langle \textsc{Reply}, u_c, t_c \rangle_{\mu_{r_i,c}}$ messages. The reply contains the execution result u_c and the client's counter value t_c that has to match the request. It is authenticated using a MAC instead of the more expensive signatures, as the replies are only verified by the client.

In case the client does not receive a reply within a predefined timeout, then it broadcasts its request to all agreement replicas. This allows the other replicas to monitor the progress of the leader replica and to replace it if necessary. The client must continue resending its request until it collects a stable reply certificate.

A correct client only issues a new request after receiving a reply to the previous one. If a client does not adhere to this requirement, then some requests may be skipped during execution in case the requests are not ordered according to the client counter values.

2.3.3. Replica

A replica consists of multiple loosely coupled components. We now follow the path of a request through the replica as sketched in Figure 2.2.

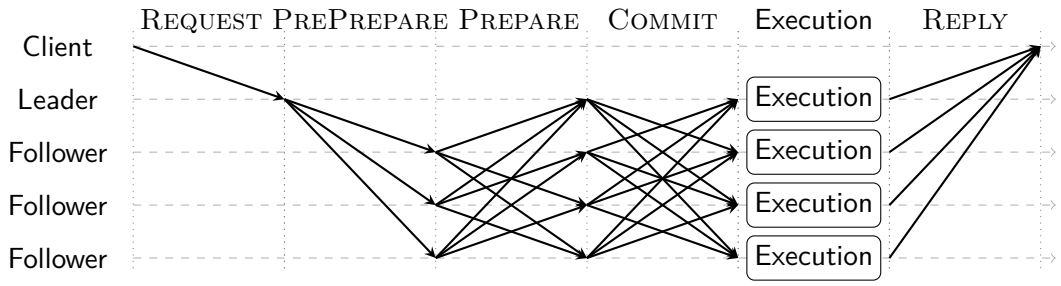


Figure 2.3: Protocol steps necessary to process a client request using PBFT.

2.3.3.1. Client Communication

After receiving a client request, a replica uses the counter value to check whether the request is old, a retransmission or new. For this purpose, replicas contain a reply cache which keeps track of the latest received request per client and the corresponding reply.

An old request is dropped silently. A retransmission of the latest request causes the replica to resend the reply if it is already available. A new request is passed on to the agreement component for ordering. The client communication component also monitors the agreement progress; we will discuss later on how a too slow leader replica is replaced.

2.3.3.2. Agreement

The agreement protocol is responsible for generating a totally ordered log of requests. For this the protocol manages a large number of *slots* that are identified and ordered by their sequence number and to which requests can be assigned.

The replicas use a view number v to determine the current system configuration. In particular, the leader replica l, which is responsible for proposing request assignments to slots, is calculated based on the current view v using a deterministic function such as $l = v \mod N$. In case enough replicas suspect the leader to be faulty, they can initiate a view change to increase the view number v and thereby replace the leader.

To order requests, the leader repeatedly proposes a *request batch* [94] (i.e., a bundle of requests) for the next unused agreement slot to construct an infinite log of requests. The other replicas then confirm the leader's proposal and exchange sufficient information to ensure that the request assignment cannot be lost, even in case of faults.

Batching

The request batch is commonly assembled by collecting requests that arrive within a short timeout, but only up to a certain size limit either in terms of the number of requests, their size in bytes or both. This amortizes the per-slot overhead of the agreement protocol over multiple requests, while also keeping the delay introduced by batching bounded. Large delays or batch sizes yield diminishing returns as the per-slot protocol overhead is already spread over several messages.

Request Assignment

As shown in Figure 2.3, the request assignment starts with a $\langle \text{PrePrepare}, v, s, r \rangle_{\sigma_l}$ message sent by the leader l and contains the current view v, the assigned sequence number s and the proposed request batch r. Once the other replicas - the *followers* - receive the PrePrepare, they process it only if the message is for their current view v. A replica locks on to the assignment from the leader replica by only storing it if it is the first one received for sequence number s in the current view. Additionally, only sequence numbers within a certain range, the *agreement window*, are processed to prevent a faulty leader from proposing requests for arbitrarily high sequence numbers. This window is discussed later on together with the checkpointing component.

After accepting a PrePrepare, each follower f_i broadcasts a $\langle \text{Prepare}, v, s, h(r) \rangle_{\sigma_{f_i}}$ message, where $h(r)$ is the hash of the request contained in the PrePrepare. Each replica then waits until it collects a quorum certificate of matching Prepares plus the PrePrepare. The PrePrepare also counts as one message in the certificate, thus reducing the number of required Prepares by one. A replica only accepts the first Prepare per agreement slot that it receives from each replica for the current view v. After the certificate at a replica is stable, we say that the request has locally *prepared* at the replica. As correct replicas only accept the first PrePrepare per sequence number in each view and thus send only one Prepare, this Prepare quorum certificate guarantees that only a single request batch can prepare in view v.

Once a replica r_i has prepared a request, it broadcasts a $\langle \text{Commit}, v, s, h(r) \rangle_{\sigma_{r_i}}$ message to all replicas. After collecting a stable quorum certificate of matching Commits, a replica has locally *committed* the slot and forwards the request to the execution.

The protocol now guarantees the following central invariant: **as soon as any correct replica has committed a slot, the request assignment for the slot is permanent and will never change in the future**.

2.3.3.3. View Change

To prevent a faulty leader from disrupting the agreement process, the other replicas closely monitor the leader. After a replica receives a request from client c with counter value t_c, its client communication component starts a timer that will expire when it takes too long for the request to be executed. The timer is stopped once the request or a later one from the same client is executed. If the timer expires for the first time, the request is forwarded to the current leader to ensure that the leader has actually received it and the timer is restarted. This guarantees that a correct leader learns about a request, even if a faulty client only sent a request to some replicas but not the leader, and it protects a correct leader from being accused of censoring a request.

If the timer expires a second time, then the replica suspects that the leader is faulty and initiates a view change to replace it; that is, the replica switches to the next view $v' = v + 1$ and stops processing messages from earlier views. Before restarting the request processing, it is necessary to reconcile the replica state for which the replica broadcasts a $\langle \text{ViewChange}, v', \mathcal{P} \rangle_{\sigma_{r_i}}$ message containing the latest stable prepare cer-

tificates $\mathcal{P}$ for all slots inside the replica's agreement window. If a slot has only prepared in an earlier view than view v, then the replica includes its latest prepare certificate.

View Synchronization

As soon as $f + 1$ replicas, that is, at least one correct replica, have suspected the leader and moved to a new view, the old view is no longer guaranteed to make progress and has to be abandoned. For this replicas keep track of the highest view they have received from each replica. Once the $f + 1$-highest received view $\tilde{v}$ is higher than the current view v at a replica, then the replica switches to view $\tilde{v}$ and sends a corresponding ViewChange. Once the request processing in a view no longer makes progress, eventually $f + 1$ correct replicas will send a ViewChange. This causes all remaining correct replicas to at least switch to the same view. The requirement for $f + 1$ ViewChanges ensures that faulty replicas cannot replace the leader on their own.

State Reconciliation

Once enough replicas have moved to the new view, the state reconciliation during a view change works by letting the leader l' for the new view v' determine the global system state based on reports from a quorum of replicas.

The leader l' collects a quorum certificate of ViewChange messages for view v' and processes it as follows. The messages only have to contain matching view numbers v', whereas the list of prepare certificates $\mathcal{P}$ may diverge. For each sequence number s, the leader extracts the valid prepare certificates included in the ViewChange messages and picks the certificate with the latest view. If there is no such certificate or there is a gap between sequence numbers, then these are filled with a no-op request, which is skipped during execution.

Then the new leader l' broadcasts a $\langle \text{NewView}, v', \vec{VC}, \mathcal{C} \rangle_{\sigma_{l'}}$ message to all replicas, where $\vec{VC}$ is the list of used ViewChange messages and $\mathcal{C}$ contains the selected prepare certificates. The leader also sends a new PrePrepare for each sequence number, which has to match the corresponding prepare certificate or proposes a no-op. Each replica verifies that the NewView includes valid ViewChange messages for view v' and that $\mathcal{C}$ was calculated correctly. Afterwards the replica enters the new view and resumes the agreement protocol's normal case for all slots. For agreement slots covered by the NewView, it only accepts a PrePrepare matching the certificate.

The other replicas have to monitor that the new leader completes the view change within a given timeout. Each replica starts a timer after receiving a quorum of ViewChange messages and stops the timer once it receives a valid NewView or switches to a higher view. If the timer expires, then the replica switches into the next view. Once a replica collects a quorum of ViewChange messages, the view synchronization ensures that the new leader will eventually also receive enough ViewChanges to be able to complete the view change.

Correctness

For the view change to maintain correctness, it has to guarantee that each committed slot keeps its value. A slot can only possibly commit if a quorum of replicas has previously prepared the slot. Thus, collecting ViewChange messages from a quorum of replicas

will include at least one correct replica which prepared the slot. Therefore, one of these ViewChange messages always includes a prepare certificate that the new leader has to include in its NewView message. In case a slot has only prepared at a few replicas but not committed, then depending on the used ViewChange messages the prepare certificate may be included or not. This is not a problem, as only committed request are guaranteed to be kept by the view change.

2.3.3.4. Execution

After the agreement the requests are assembled into an ordered log based on their sequence number. They are then executed in this order. In case there are gaps between sequence numbers, the execution has to wait until the corresponding slots are committed.

The application processes each request, modifies its state accordingly and returns a reply. Requests in a batch are passed one by one to the application. To prevent duplicate request executions, a request is only executed if it is a new request, that is, its request counter value t_c must be higher than that of the last executed request for the client. Otherwise, the request is skipped. The reply is then stored in the reply cache and sent back to the client.

2.3.3.5. Checkpoints

With the replica construction described so far, we end up with an infinite request log requiring unbounded amounts of memory, which is not practical. To limit the memory usage, a replica periodically creates a *checkpoint* containing a *snapshot* of the replica and application state, which allows the garbage collection of older data. A checkpoint also enables other replicas to update themselves without having to execute all intermediate requests and thereby allows replicas to catch up.

As the memory usage also depends a lot on the used application, we assume that the application is implemented such that it only uses a bounded amount of memory. This is necessary to prevent faulty clients from causing the replicas to run out of memory.

Checkpoint Creation

After the replica has executed a request with a sequence number divisible by the *checkpoint interval* k, it creates a checkpoint consisting of a snapshot of the application state, the corresponding cached replies and the vector $\vec{t_c}$ of their counter values [72]. Similar to the request execution, the snapshot must be deterministic, that is, all replicas must collect the exact same snapshot. This is necessary to allow replicas to verify the correctness of a snapshot. A checkpoint is only created after every kth request, as it can be very expensive to snapshot the state of an application.

Before a checkpoint can be used to update other replicas, a replica first has to collect a *checkpoint certificate* proving its correctness. For this, replicas exchange signed $\langle \text{Checkpoint}, s, h(cp) \rangle_{\sigma_{r_i}}$ messages containing the last sequence number s executed before creating the checkpoint and its hash $h(cp)$. Once a replica has collected a

certificate of $2f + 1$ CHECKPOINT messages[2], the checkpoint becomes *stable* and allows the replica to forget all checkpoints and requests with lower sequence numbers.

As the application state consists of many small objects, their data has to be concatenated for the hash calculation. For a more efficient calculation of the checkpoint hash, a Merkle tree [154] or incrementally updatable hashes [56] can be used. Either way, every replica must use the same order for state objects when calculating the hash, for example by sorting the objects according to their identifier.

Checkpoint Transfer

Once a replica has collected a stable checkpoint certificate, it broadcasts the certificate to all replicas to inform them about the checkpoint. As the CHECKPOINT messages only contain the checkpoint hash, this step only results in a small amount of network traffic.

If a replica learns that it has fallen behind, for example, by collecting a certificate for a newer checkpoint, it can request the full checkpoint from another replica. Afterwards it has to verify that the received data matches the hash in the certificate.

Bounded State

To bound the agreement's memory usage, the number of slots that can be in use at a time must be limited. This is done by only processing slots for sequence numbers inside the agreement window. The lower bound of the window is set to the sequence number after the latest stable checkpoint. And the upper bound is given by the lower bound plus the window size. The latter is a small multiple of the checkpoint interval k to allow the request ordering to continue while the next checkpoint is not yet stable.

The view change also requires a small modification: each VIEWCHANGE message must additionally contain the latest stable checkpoint certificate known to the replica. The leader of the new view then selects the latest stable checkpoint and includes it in the NEWVIEW. Only prepare certificates with a sequence number newer than that of the checkpoint are included in the NEWVIEW message. When a replica receives a NEWVIEW containing a newer checkpoint certificate, then the replica first has to retrieve the checkpoint and update its state accordingly.

When a replica receives agreement messages for sequence numbers *after* the window, that is, with a higher sequence number than the upper bound, then the replica discards these messages. This ensures that a replica does not run out of memory if a faulty leader assigns requests to slots with very high sequence numbers. If a dropped message becomes necessary later on, the replica has to request a retransmission [56]. As an optimization, a replica can keep a limited number of such future messages [201].

2.4. State of the Art

A significant body of work has been dedicated to improving the performance of Byzantine fault-tolerant protocols. In the following we review approaches that explore different

[2]Due to the additional checkpoint-certificate broadcast described in the following, which is not included in PBFT [57], it is also possible to create checkpoints based on $f + 1$ signed CHECKPOINT messages. This also allows running the application at only $2f + 1$ replicas [210].

ways of reducing the response times for clients. We group them based on the protocol phase an approach optimizes: either the communication with the clients (Section 2.4.1), the agreement on a request order (Section 2.4.2) or the request execution and checkpointing (Section 2.4.3).

2.4.1. Reducing the Client Communication Latency

The first step in ordering a request is for a client to send it to the replicas. This section starts with approaches to select a system configuration offering optimal response times for clients, then discusses optimizations related to returning a reply to a client. Finally, we review approaches to optimize the transfer of client requests to the leader replicas.

Improving Client-Perceived Response Times

To check whether a different system configuration could lower the response time experienced by clients, the replicas need a way to determine the latency for alternate configurations. Our approach called Archer [82] lets clients issue probe requests that retrace the protocol phases and allow exploring which response time a different leader could provide. To prevent faulty replicas from skipping protocol steps to gain an advantage, the replicas compute a hash chain based on the protocol execution, which allows the client to verify the correct behavior of the replicas. When entering a new protocol phase, the resulting message sent by a replica includes a hash computed from the received messages and the secret the replica shares with the client to authenticate messages. This ensures that a faulty replica cannot fake another replica's message. The client then verifies that the hash value included in the replica's replies is consistent with a correct protocol execution. If a faulty replica skips an execution step, this will result in a wrong hash, which the client can detect.

The clients then send their latency measurements for each received reply message to the replicas. These run a special system application that analyses these measurements and can trigger a reconfiguration of the agreement protocol to optimize its latency. As shown by our experiments conducted using Archer, the location of the leader replica in relation to the client has a major influence on the response times experienced by clients at a certain location. A change of the current leader replica will also result in changes of the response time. Depending on the replica's locations, there is also no single leader location that offers the optimal latency for all clients such that for a shifting client load the leader replica has to be adapted to remain optimal.

Telling the Client

Returning a reply to the client also contributes to the overall response time. By differentiating between read requests, which do not modify the application state, and write requests, which do, it becomes possible to optimize them separately. PBFT [57] proposes a read optimization that works in a single message round-trip: a client queries all replicas, which directly execute the request without ordering it first and reply to the client. Once a client has collected matching replies from a quorum of replicas, it can accept the result. Otherwise, it has to resend its query as a regular request. This optimization comes at the cost of requiring the client to also wait for a quorum of replies to write requests, which

can increase their latency by about 10% [189]. In the presence of faulty replicas, PBFT additionally has to exchange the agreement results for already committed slots between replicas to ensure that a client can always get a reply [42].

SBFT [108] includes an additional protocol phase during which the replicas collect a threshold-signed reply that is sufficient to prove to the client that $f + 1$ replicas have confirmed the correctness of the reply. A threshold signature scheme ensures that a valid signature can only be created if a sufficient number of signers contribute a signature share, which are then combined into the final signature. Thus, it is sufficient to send a single reply containing the combined signature to the client. While this reduces communication, in terms of latency it is overcompensated by the additional protocol phase required to assemble the threshold signature.

By using client-side speculation [207], it becomes possible for a client to predict the outcome of a request based on the first reply it receives or sometimes even a cached result. The client can then continue executing without waiting for the full reply. However, as the result is still speculative at that point, the client cannot perform any actions that would externalize speculative state which limits the usability of this approach.

Rotating the Leader

Protocols with a rotating leader can allow clients to submit their requests to the nearest replica (in terms of latency), and thereby avoid communication with far away replicas. However, in existing protocols this usually has the result of requiring the replica to wait for its turn or to wait until the other replicas complete their requests. For example, BFT-Mencius [161] partitions the agreement slots in round-robin manner across replicas, such that each replica can propose received requests for its own agreement slots. To still end up with a global order, the replicas have to wait with executing a request until all slots with lower sequence number have been committed. This may require additional steps to fill these gaps and can delay the request processing. Essentially, the slowest leader replica can end up limiting the performance.

Spinning [201] continuously changes the leader replica in an attempt to limit the effect a faulty leader can have on the request processing. The protocol is not optimized for latency, as the leader changes occur consecutively, which requires a request sent only to the local leader to wait for its turn.

Protocols from the HotStuff [211] family do not follow the classical design of assigning requests to sequence numbers, but instead build a chain of request batches by including a hash of the previous batch in each agreement slot. This allows the protocol to continuously switch leaders without incurring an expensive view change protocol, but also limits the leader replicas to only propose requests one after another. As the communication pattern primarily consists of one-to-all communication and back, which avoids all-to-all communication as for example used in PBFT, this requires additional communication steps compared to PBFT-like protocols, thereby resulting in higher response times.

Leaders Everywhere

For protocols tolerating only crash faults, that is nodes either behave correctly or crash, Generalized Paxos [137] has shown that multiple replicas can concurrently propose non-

conflicting or commutative requests. Such requests can be executed in an arbitrary order, as they for example access independent parts of the application state. MDCC [132], which hosts replicas at multiple sites, uses this approach to allow clients to contact a local replica, which then orders the request concurrently to other requests within two wide-area communication steps. Only in case of conflicts, it becomes necessary to fall back to a single leader, which has to determine an order for the conflicting requests, before switching back to the concurrent request ordering.

Egalitarian Paxos [162] is able to even resolve conflicts in a distributed manner. To order a request, all replicas have to agree on a set of possibly conflicting requests. Using these sets, the replicas can then determine the order in which requests should be executed. If the conflict sets determined by the leader and a quorum of followers are identical, then the protocol can follow the fast path and commit the request within two wide-area communication steps. Otherwise, a fallback to the slow path is necessary in which the replicas reconcile the proposed conflict sets by merging them. For the execution, requests are sorted such that all conflicts of a request are executed first. In case of cyclic dependencies, all requests on such a cycle are sorted deterministically and are executed in this order. Overall, this ensures that conflicting requests execute in the same order on all replicas. As each replica is able to propose requests, clients can send their requests to the nearest replica which immediately starts ordering them.

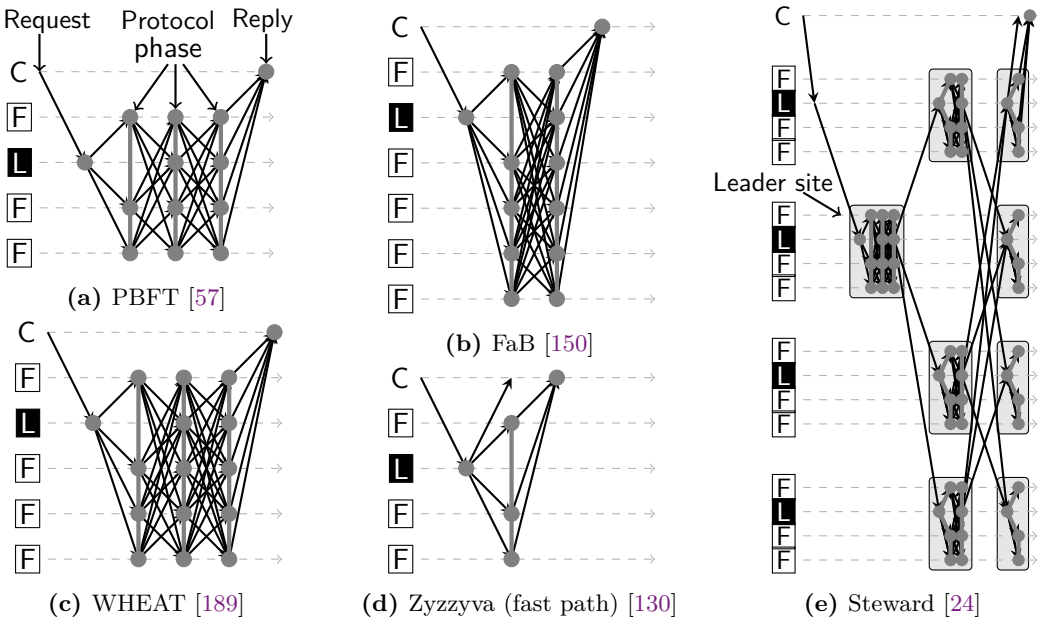
However, the just described protocols only tolerate crash faults. Byzantine Generalized Paxos [171] extends Generalized Paxos to also tolerate Byzantine faults. It adds another protocol phase such that the agreement consists of the usual three communication steps required for many Byzantine fault-tolerant protocols. Each replica can propose an order for requests that has to be confirmed by a quorum and only succeeds if the request orders proposed by different replicas do not conflict with each other. In particular, this requires that differences in request ordering only concern commutative requests such that each order still yields the same result. In case of conflicts, the protocol has to fall back to a single leader replica that resolves the conflict by proposing a canonical order for the pending requests. Only afterwards the concurrent request processing continues.

Conclusion

The response times for client requests can in some cases be reduced by lowering the communication latency between a client and the replica that proposes it for ordering. However, care must be taken that a lower client communication latency does not result in replicas having to wait for each other and thereby increase response times again.

2.4.2. Reducing the Agreement Latency

In this section we shift our focus to optimizing the agreement on a request order. We start with discussing several protocols using additional replicas or speculative request execution. Afterwards we review approaches to assign different weights for replicas in order to prefer those replicas that provide the best performance, followed by hierarchical replication protocols and weaker consistency guarantees.



(a) PBFT [57] **(b)** FaB [150] **(c)** WHEAT [189] **(d)** Zyzzyva (fast path) [130] **(e)** Steward [24]

Figure 2.4: Protocol structure of different Byzantine fault-tolerant protocols drawn with uniform latencies. We distinguish between the client (C), the leader (L) and followers (F). The leader is additionally indicated by an inverted label. Communication between replicas at the same site in Steward is not drawn to scale. The vertical lines mark the point at which a replica enters the next protocol phase.

More Replicas

The most obvious way to reduce the response time of a state-machine replication protocol is by reducing the number of protocol phases, which in exchange requires an increased number of replicas. The three protocol phases of the well-known PBFT protocol [57], which was presented in Section 2.3, are shown in Figure 2.4a. Here, PBFT serves as starting point which requires $3f + 1$ replicas and takes three communication steps in the agreement protocol between a request reaching the leader replica to successfully ordering the request. Along with the client communication, it takes at total of five communication steps between the client submitting a request and receiving a reply.

Using at least $5f+1$ replicas as in the protocol FaB [150], which is shown in Figure 2.4b, allows for agreement in two communication steps. Compared to PBFT the protocol skips the prepare phase by using a larger quorum size of $4f + 1$ replicas, which guarantees an overlap of at least $2f + 1$ correct replicas. During a view change, therefore a majority of replicas in the quorum will vote for the correct value. FaB can be further optimized to require only $5f - 1$ replicas [79, 135]. Especially when tolerating more than a single fault, this still requires more replicas in the agreement process.

While using additional replicas could reduce latency, the network traffic necessary to distribute requests to each of them increases [1], which can result in the leader becoming a bottleneck. To ensure fault independence, the higher number of replicas requires additional different versions of the protocol implementation.

Weighted Communication Phases

As discovered by WHEAT [189], which is sketched in Figure 2.4c, the selection of replicas used to complete the agreement can have a major influence on the latency. In a wide-area environment the communication latency between different pairs of replicas can vary significantly. Consequently, certain subsets of the replicas are more suitable for fast protocol executions than others. WHEAT uses additional replicas which allow it to prefer a set of well-connected replicas to handle the agreement process. However, the size of a Byzantine majority quorum normally increases when adding more agreement replicas. To avoid this WHEAT assigns weights to replicas such that votes from the (small) preferred replica quorum are sufficient to collect a Byzantine majority of votes. As a follow-up AWARE [43] automatically calculates and adjusts these replica weights. Whether it is possible to actually improve the response time or not depends on the location of the additional replica(s). The higher number of replicas also requires a leader to distribute a request to more replicas.

Speculation

Reducing the number of communication steps to a total of three is possible by speculatively executing a request and providing the client with the reply, even though the request is not yet committed. This approach is used by Zyzzyva [130], shown in Figure 2.4d. The costs for this reduction are that a client has to wait for replies from all $3f + 1$ replicas instead of only $f + 1$. In the wide-area setting this means waiting for a reply from the replica that is the farthest away from the client in terms of latency, which can negate any the latency improvements. The speculative execution also comes at a cost. If the leader replica misbehaves, it can become necessary to roll back the application to an earlier state, which requires an efficient method to revert to a prior state.

The quorum-based replication protocol Q/U [1] goes even further and allows updates to happen in just two phases, that is, replicas directly reply to client requests. However, this comes at the cost of requiring $5f + 1$ replicas and only supporting operations where a client can read or conditionally update an existing object. If conflicts between updates from clients occur, it even becomes necessary to back off exponentially.

As a middle ground, HQ [68] primarily relies on client to replica communication, but requires four communication steps to process write requests. In case of conflicts, it falls back to ordering the affected requests using PBFT, which further increases the protocol latency. Quorum-based protocols like Q/U and HQ do not rely on a leader replica, which precludes them from using batching and thereby limits their throughput [68]. In addition, clients and replicas have to exchange large amounts of certificates to prove the correctness of messages, resulting in a high network traffic overhead [68].

PBFT [58] supports a more conservative approach called tentative execution, which overlaps the commit and reply phase. As a trade-off a client has to wait for a quorum of replies instead of just $f + 1$ replies, which ensures that the request survives a view change and has a similar effect as a commit quorum. Like the previous approaches, it may become necessary to roll back the application state. Additionally, tentative execution is only possible if the previous request has already committed, making it less effective in wide-area networks when multiple slots are ordered concurrently.

The main drawback of protocols using speculation is their complexity. All system components must be able to roll back misspeculated requests, which can be very complex depending on the application or impossible if the request execution has already triggered external communication. This complexity is also showcased by the fact that Clement et al. [63] pointed out that the prototypes for HQ and Zyzzyva did not implement the full view change. In addition, nearly a decade after its initial publication, the view change protocol for Zyzzyva was shown to be subtly flawed [2, 3].

Hierarchical Replication

It is possible to make use of the low latency of local-area communication – even in a wide-area setting – by using a hierarchical protocol design. Protocols like Steward [24], shown in Figure 2.4e, or CustFT [20] use multiple sites spread across the world, which each contain a group of replicas. Every group is able to locally process read requests and to execute a part of the overall agreement protocol, reducing the amount of wide-area communication necessary. As the replicas in a group are located at the same site, they can communicate with each other with low latency and thus improve the response time. GeoBFT [110] follows another approach by partitioning the agreement slots to different sites, which each can locally order requests for their slots. This allows each replica site to order its requests independently of the other replica sites. However, the request execution still has to wait until every site has filled all earlier slots, such that a single slow site can delay all other sites.

For crash-fault tolerance several protocols [7, 65, 165, 170] designate small sets of replicas to be primarily responsible for managing specific objects. By selecting replicas located close to the clients that frequently access these objects, it becomes possible for those clients to only communicate with nearby replicas when accessing objects and therefore optimize for the normal case. Interactions with far away replicas are only necessary for rare accesses to non-local objects or to update the replica sets after replica failures have occurred.

Weaker Consistency Guarantees

Another way to reduce latency especially for read requests is to offer different consistency guarantees for them [65, 89, 196]. Weaker consistencies than linearizability allow clients to temporarily see an outdated state, which usually translates to faster response times, as they can allow replicas to process a request locally without requiring wide-area communication. In some protocols the choice of consistency is left to the client [24, 89, 122], Pileus [196] even allows a client to specify preferences for different consistency guarantees based on how long it would take to process the request.

Hierarchical replication protocols like Steward [24] are also well suited for this approach. Each replica group has sufficient members to process read requests with weak consistency locally and only requires wide-area communication to guarantee strong consistency. If a weaker level of consistency is sufficient for clients, then the read requests of clients that are located near a group can be processed locally at the group and therefore with response times comparable to a local-area setting.

Conclusion

As shown by several of the presented systems, reducing the number of protocol phases requires other trade-offs. Either in the form of additional required replicas, which increase the overall resources necessary, or additional complexity at the protocol level and the application in case of speculative execution. However, it is also possible to add protocol phases to reduce the response time. Hierarchical replication protocols, which differentiate between local- and wide-area communication, can use additional local, low-latency communication steps to reduce the overall latency.

2.4.3. Reducing the Execution Latency

Besides latency induced by the communication between clients and replicas, executing a request can also require a significant amount of time. We first discuss how requests can be handled which take a long time to execute and afterwards discuss delays caused by the periodic creation of checkpoints.

Execution Slowdowns

Delays during processing can lead to spikes in the response time for individual requests, the so-called *tail latency* [71]. Different from random transmission delays, which likely only affect some replicas, delays during the request execution can affect all replicas at once. If some requests take a long time to execute, this delay will become visible to the clients. It can be partially alleviated by executing non-conflicting requests concurrently [60, 131, 142]. But ultimately it remains the responsibility of the application to ensure that operations can be executed sufficiently fast.

Hyperledger Fabric [26] and EVE [128] turn the request processing on its head by first executing requests without explicit coordination and afterwards agreeing on the execution results. In Fabric [26] a client submits its request to the execution replicas and collects a certified execution result. This result is then sent to the agreement for ordering. If conflicts arise, the execution result is dropped, and the client has to retry its request. As this approach is not tightly coupled with the agreement protocol, it can be integrated into other systems. EVE [128] lets replicas execute requests in parallel and verify that they arrived at the same results later on. In case of a mismatch, the execution is rolled back followed by executing the problematic requests sequentially.

Checkpointing

After executing requests, replicas periodically take an application snapshot to create a new checkpoint. For applications with a large state, taking a snapshot can require a significant amount of time and thus lead to service interruptions which are visible to users. If these reach the order of several hundred milliseconds, they stand out even when considering the wide-area communication latency.

The straw-man approach of copying the full application state also has the downside of requiring a replica to keep multiple copies of the full application state. PBFT [57] and BASE [59] therefore store differential snapshots, which only contain the parts of the application state that are different from the next snapshot. When the application is about to modify a state object, it then has to notify the replication library. If necessary,

the library backs up the old object version and stores it in the latest snapshot. That way, snapshots only store copies of the state objects which were modified by requests after creating the snapshot. Together with the current application state, it is possible to reconstruct the state that existed when one of the snapshots was taken. The approach also removes the need to copy the application state when creating a checkpoint. However, the checkpoint hash still has to be calculated immediately after creating the application snapshot and thus still blocks the request execution.

BASE [59] introduces an abstraction layer enabling the replicas to use different service implementations by on-demand translating the concrete state into an abstract state that is identical across implementations. This allows implementing a form of N-version programming, for example, by using different COTS databases. When backing up state parts that are about to change or when calculating the checkpoint hash, this approach adds a translation step. Depending on the size of the state part or the conversion process, this can introduce significant further delays.

With the standard checkpointing approach, all replicas take their snapshot at the same logical point in time, that is, after processing the same sequence number, which can result in service disruptions. Dura-SMaRt [46] staggers the checkpoint creation across replicas such that it does not overlap in time. However, to apply a checkpoint, it is necessary that multiple replicas confirm its correctness. Therefore, replicas that have created their last checkpoint at a different point in time additionally provide a list of requests that were executed in the meantime. A replica requiring a checkpoint then requests multiple checkpoints and installs the $f + 1$-newest checkpoint cp it has received. To verify the checkpoint, it then executes requests until reaching the f-newest checkpoint, verifies that its own state matches that checkpoint, proceeds to the $f - 1$-newest checkpoint and so on until it has verified the newest checkpoint. This approach has the huge drawback that an application has to apply a checkpoint containing unverified state. Thus, faulty replicas could try to compromise other replicas by sending a manipulated checkpoint to exploit vulnerabilities in the procedure to apply a checkpoint.

Upright [62] proposes multiple approaches to create checkpoints. The simplest one reduces the checkpointing frequency using so-called hybrid checkpoints, which consist of an application snapshot and an incremental list of ordered requests to execute afterwards. A lower checkpoint frequency requires more requests to be replayed after applying a checkpoint, slowing down the recovery of a replica. Another approach runs a second copy of the application process that is paused for checkpointing. However, this doubles the resource usage. An application can also be modified to use copy-on-write data structures, however, this can require significant code changes. Finally, usage of the fork syscall, which creates a copy of the current process whose memory is shared in a copy-on-write fashion, is limited to single-threaded applications, which in particular rules out programs written in Java and is also incompatible with executing requests in parallel.

Conclusion

Even though the reviewed approaches can reduce the service interruptions caused by creating checkpoints, they still can experience significant delays when creating checkpoints or require applying unverified application state.

2.5. Summary

In this chapter we have presented the system model used throughout this thesis and discussed special characteristics of wide-area networks. Afterwards we have introduced the structure of a state-machine replication protocol and its main components. We have reviewed the state of the art relevant for optimizing the latency of the client to replica communication, different agreement protocol structures requiring varying numbers of protocol phases and replicas and the possible causes of delays in the request execution in particular when creating checkpoints. In the following chapter, based on the wide-area characteristics and the state of the art, we analyze the potential for lower response times by optimizing the client communication and request ordering latency. In addition, we examine how checkpointing approaches can avoid processing delays.

3
Problem Analysis and Suggested Approach

Based on the state of the art presented in Section 2.4, we first analyze in Section 3.1 for wide-area environments which problems arise from different approaches to reduce the latency for different steps of the state-machine replication protocols. After the analysis we present in Section 3.2 three approaches that each reduce the latency for one of the protocol steps as far as possible. We define our central questions in Section 3.3 and sketch the design goals guiding the development of our protocols. Finally, Section 3.4 concludes the chapter.

3.1. Problem Analysis

For the purpose of this analysis, we divide the request processing steps in a state-machine replication protocol into client communication-, agreement- and execution-specific steps. Figure 3.1 shows this division using the PBFT [57] protocol as example. In the following

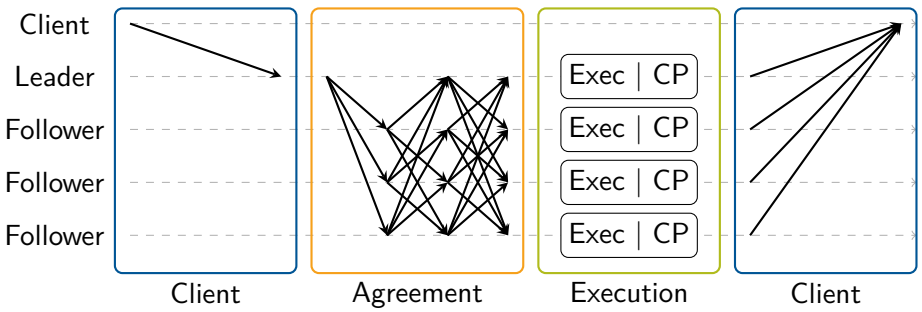
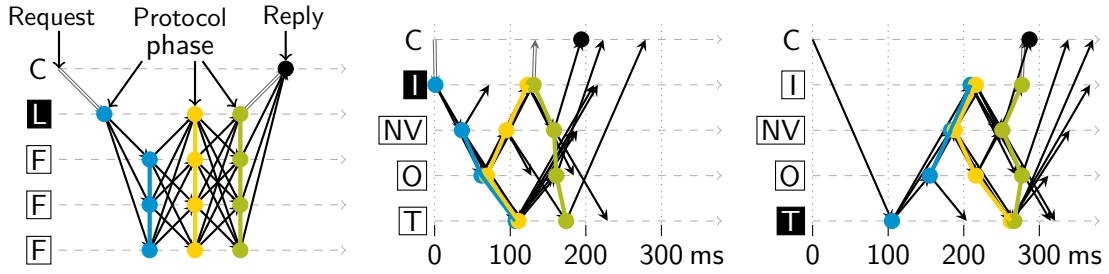


Figure 3.1: Protocol steps that are necessary to process a client request grouped into client communication-, agreement- and execution-specific steps at the example of PBFT.



(a) Uniform latencies **(b)** Leader in Ireland **(c)** Leader in Tokyo

Figure 3.2: PBFT protocol progress for uniform latencies and for real wide-area latencies from Amazon EC2. For uniform latencies, we only distinguish between leader (L) and followers (F). For the wide-area setting the replicas are located in Ireland (I), Northern Virginia (NV), Oregon (O) and Tokyo (T). The client (C) is always located in Ireland. Depending on the location of the leader replica (inverted label) the response time can vary significantly.

we analyze for each of the protocol steps whether it is possible to **reduce its latency nearly down to zero** and review drawbacks of existing approaches.

Client Communication

The usual way of drawing protocols with uniform network latencies, which is shown in Figure 3.2a for PBFT [57], suggests that each replica is equally suitable for tasks like the leader role. While this works well in a local-area network where the communication latency is roughly uniform, the situation is less clear when it comes to wide-area networks, where the communication latency can be very different. Communication between replicas in the same region takes only a few milliseconds, whereas communication between replicas located at distant regions can require hundreds of milliseconds.

For example, the response time improves from 287 ms for a leader in Tokyo (Figure 3.2c), to which the request submission already takes 105 ms, down to 195 ms with the leader in Ireland (Figure 3.2b), which is also the region the client is located in. This shows that depending on the location of a leader replica in relation to a client, the initial request submission can already add a significant amount of latency to the overall request processing. For protocols using a single leader replica, the response time experienced by a client strongly depends on the current location of the leader replica [43, 82, 189]. If a protocol changes the current leader replica, this consequently leads to large changes in the response times for clients.

Protocols that use a rotating leader [161, 201] reduce the time it takes for a request to initially reach a leader by allowing clients to submit their requests to the nearest replica. However, as new requests are proposed in a round-robin manner, the replicas have to wait for each other to finish their turns. This can lead to situations in which the slowest replica determines the overall performance, as all other replicas have to wait for it to complete its turn.

In contrast, in an egalitarian protocol, such as EPaxos [162], only requests that conflict with each other must be coordinated across replicas. However, EPaxos is only crash-fault

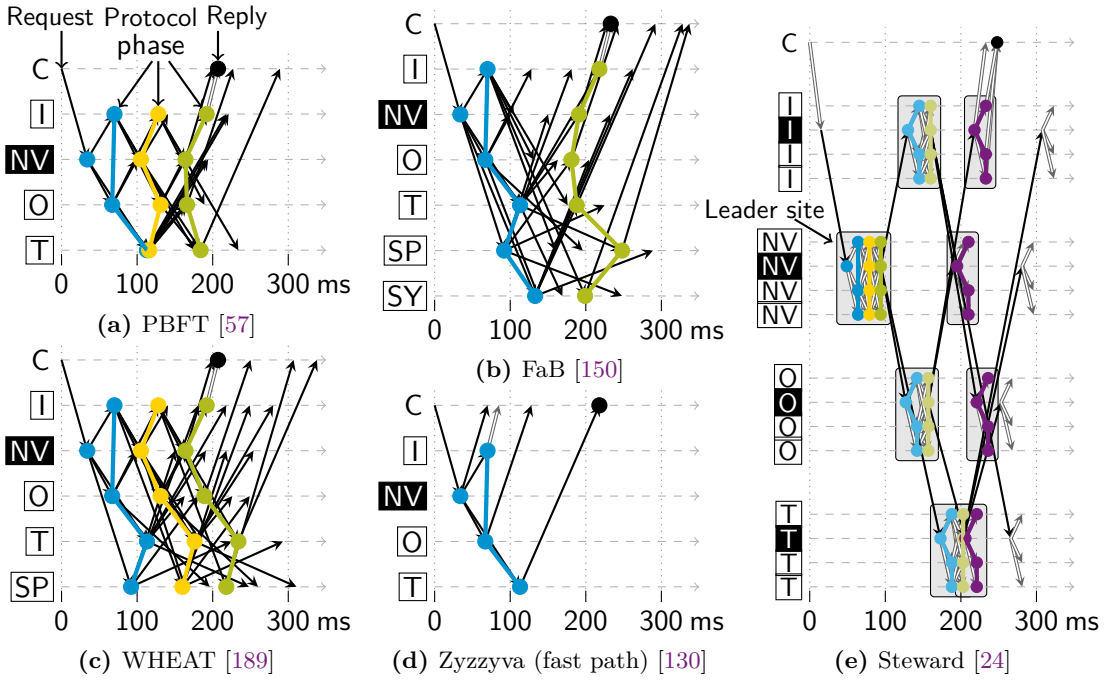


Figure 3.3: Protocol structure of different Byzantine fault-tolerant protocols. The client, marked as C, is located in Ireland, the replicas are in Ireland (I), Northern Virginia (NV), Oregon (O), Tokyo (T), São Paulo (SP) and Sydney (SY). The leader is indicated by an inverted label. Grey arrows (⟹) represent local communication (not to scale), whereas black arrows (⟶) are used for wide-area communication. Equivalent protocol phases are marked in the same color.

tolerant, making it unsuitable for our system model. Byzantine Generalized Paxos [171] also tolerates Byzantine faults, but has to alternate between a fast path allowing all replicas to propose requests without further coordination and a slow path using a central leader in case of conflicting requests. To avoid the costs associated with a central leader a protocol should not use a slow path with a central leader.

Agreement

Judging from a protocol drawn with uniform network latencies as in Figure 3.2a, it seems obvious that fewer protocol phases translate to lower latency. However, as shown in Figure 3.3 this depends on the actual communication latency and the location of additional replicas. For example, when comparing PBFT [57] in Figure 3.3a with FaB [150] in Figure 3.3b, the result can even be a higher response time if the additional replicas increase the time to collect a quorum. The efficacy of weighted voting used by WHEAT [189] is also limited by the available replicas. Figure 3.3c shows that, if the existing replicas are well-connected, additional replicas do not reduce response times. Even a reduction to a single communication step as in the speculative agreement protocol Zyzzyva [130], shown in Figure 2.4d, does not automatically improve response times. Now, a client has to wait for replies from all $3f + 1$ replicas instead of $f + 1$. This means

waiting for a reply from the replica that is the farthest away from the client in terms of latency, which can negate any latency improvements.

Thus, **counting protocol phases is not sufficient to accurately judge the performance of a protocol**. Instead, it is necessary to decrease the number of high latency communication steps as opposed to just the overall number of steps. Especially when taking local communication into account, this can mean that introducing new low-latency communication steps reduces the overall latency.

In general, the previous approaches can only reduce the response time by a limited amount. To reduce the agreement latency nearly down to zero, a more drastic change is necessary. All replicas involved in the agreement should be located at the same site to allow for local area communication. This is partially the case for the hierarchical agreement protocol Steward [24], shown in Figure 3.3e, which however also has to run an agreement protocol between sites that reintroduces wide-area communication into the agreement protocol and thus increases the latency before a request can be executed. Thus, to reduce the agreement latency to nearly zero, we need a protocol which places the replicas responsible for the agreement close to each other without introducing additional wide-area communication and without compromising the system's availability.

Execution

Checkpointing the application state after executing a certain number of requests, is necessary to ensure that the system has a bounded state size. As opposed to the execution of a request which is completely handled by the application, a *replication library* typically provides supporting infrastructure for checkpointing [46, 47, 57, 59, 62]. To create a checkpoint the application either has to completely dump its current state [47, 62] or continuously notify the library about changes to parts of its application state. However, even collecting just these changes can lead to long pauses of the execution [46, 81] that are noticeable for the user as tail latency [71]. These delays can reach multiple seconds, which lets them stand out even in wide-area networks. This is ultimately caused by a common property of the checkpointing support in the replication libraries, namely that a replica has to stop the request execution while collecting a copy of the application state. Removing the need for such a pause could prevent these latency spikes.

Challenges

In addition to the above problems, which we want to solve for each protocol phase, we have identified several common properties a protocol should provide:

Byzantine Fault Tolerance A system must remain correct despite a limited number of Byzantine-faulty replicas and an arbitrary number of faulty clients. In particular, faulty clients must not be able to prevent correct clients from receiving a reply.

Low Latency The system should reduce the latency for either the client communication, agreement or execution-specific steps as far as possible.

Strong Consistency All write requests must be processed with strong consistency. For read requests, a client must be able to read with strong consistency, but may be offered to read with a weaker consistency level.

Resource Efficiency The protocol should only use as few replicas as possible. That is, the agreement parts of a protocol should only require $3f + 1$ replicas.

Bounded State An implementation should only require a bounded amount of state to process requests. In particular, this requires a protocol to support checkpointing and limiting the number of concurrently processed requests.

3.2. Suggested Approach

Our goal is to reduce the response times for client requests in a geo-replicated Byzantine fault-tolerant system. To achieve this, we propose to structure the system such that it reduces the latency for one of the processing steps, ideally yielding a latency close to zero. For this we present three approaches that each focus on a different processing step. Note that not all approaches can be combined with each other: lowering the client communication or agreement latency are opposite ends of a trade-off, minimizing both at the same time is not possible. Somewhat counterintuitively the approach to minimize the client communication latency primarily focuses on how to adapt the agreement accordingly and vice versa.

Low-Latency Client Communication

The longer it takes for a client request to reach a replica that is able to start the agreement process for the request, the more time passes for just transmitting the request. To minimize this latency, a client should be able to directly submit its request to the nearest replica. This replica then must be able to independently propose the request without waiting for other replicas. As approaches that rely on partitioning the sequence numbers between replicas can introduce such an interdependency, a system should instead only rely on dependencies between requests. To consistently order requests, the replicas have to agree on the dependencies for each request. For non-conflicting requests it should be possible to do so using a fast path requiring just three protocol phases like for PBFT, but with the difference that each replica can propose requests for ordering.

Low-Latency Agreement

Based on the insight that the latency of a communication step strongly depends on which replicas take part and their relative location to each other, we can structure the system such that it minimizes the time necessary to complete the agreement process. We achieve this by placing the (agreement) replicas close to each other, ideally in the same region. In order to ensure that this nevertheless does not compromise the availability of the system, we rely on the structure of modern cloud infrastructures, which offer availability zones [19, 101, 159] that are designed to fail independently but are still located in proximity of each other. That way, the replicas can communicate with each other over short-distance communication links, which only adds little latency to the agreement process. Only a few cloud regions offer sufficient replicas to host all $3f + 1$ agreement replicas in different availability zones. To offer even lower latency to clients in exchange for weaker consistency guarantees, we move the execution closer to the clients by adding execution groups consisting of $2f + 1$ replicas located in regions near the clients. This smaller

number of replicas can then be distributed to the three availability zones commonly available in a region [19].

Low-Latency Execution
Capturing all data required to create a checkpoint can lead to significant service interruptions. Instead of collecting the checkpoint data after executing a certain sequence number, we propose to run the checkpoint collection concurrently to the normal request execution. Ideally, this can drastically reduce the pause time necessary to capture the application state. However, this yields a fuzzy checkpoint of the application state, containing state parts captured at different points in time. The application has to provide an interface allowing the replication library to efficiently track changes of the application state while collecting the checkpoint data. In order to reach a consistent checkpoint that is also comparable across replicas, a post-processing phase should use the tracked changes to make the snapshot deterministic again.

3.3. Central Questions

In the course of this thesis, we investigate the three approaches presented in Section 3.2 in regard to the following two central questions.

Improving Client-Perceived Response Times Can our approaches be used to reduce the response times experienced by clients, and what are the implications for the overall system performance and throughput?

Reducing Performance Variation Can our approaches be used to reduce the performance variation that is caused by different system configurations like the current location of the leader replica or periodic tasks like checkpointing the replica state?

For each of the approaches further questions and design goals arise which must be solved during the development of the systems.

Low-Latency Client Communication
Our protocol Isos is described in Chapter 4 and allows clients to submit their request to the nearest replica which can immediately start the agreement process. This poses the problem of integrating dependencies between requests into a Byzantine fault-tolerant protocol that must ensure that only valid dependencies can be proposed and that all replicas eventually learn about them. Similarly, the fault handling must be able to maintain the correctness of dependencies between requests. Based on these dependencies, the replicas require a way to determine an execution order for the requests. In line with the challenges identified earlier on, the replicas also have to coordinate the checkpoint creation across all replicas. Finally, we will analyze the performance provided by the protocol along with the effects of having multiple leader replicas at the same time.

Low-Latency Agreement
In Chapter 5 we present Spider, which follows the approach of reducing the agreement latency as far as possible. The use of cloud availability zones raises the question of how to

structure the system to benefit the most from this common cloud architecture. Splitting replicas into agreement and execution groups requires them to exchange information about requests and in which order they should be executed. We will investigate how to couple these groups and how to do so in a modular way to lower the complexity of the system while remaining efficient. Having an execution group located near clients allows providing faster replies to clients in exchange for relaxed consistency guarantees. In addition to the question which response times can be achieved by the system and how stable these are across leader changes, we want to research which additional consistency semantics the system can offer and their influence on the response times. To provide optimal latency for all clients, this also requires the system to be able to adapt its set of execution groups.

Low-Latency Execution

In Chapter 6 we design a mechanism called Deterministic Fuzzy Checkpointing, which allows a replica to collect a fuzzy checkpoint while executing requests and to convert the checkpoint into a deterministic one that is identical across all replicas. We investigate which information is necessary to make a checkpoint deterministic again and how the application interface has to be structured to support this mechanism. As the amount of data that must be copied influences how much work is necessary, we analyze different ways to reduce the amount of copying necessary. To collect a checkpoint for the same sequence number, all replicas have to finish their data collection at that sequence number. For this the replicas need a mechanism to determine a suitable starting point that is early enough to copy all data without having to pause the execution, but late enough to only copy as little data as necessary. The data collection for a checkpoint still requires resources, therefore we will analyze how far our approach can reduce the execution interruptions, its run time costs and the effect on the response times.

3.4. Summary

Our analysis has shown that protocols forgo a lot of potential to reduce response times by not differentiating between the costs for different protocol steps and by not considering costs that only occur periodically. We have suggested three approaches each focused on reducing the costs for one basic protocol step as far as possible. Chapter 4 enables all replicas to immediately propose new requests, which allows clients to submit their request to the closest replica and thus reduce the response time. In Chapter 5 we use availability zones provided by modern cloud infrastructures to place all agreement replicas in close proximity to minimize the agreement costs. Afterwards Chapter 6 runs the checkpoint data collection in parallel to the request execution, which yields a fuzzy checkpoint and can spread the cost of periodically creating a checkpoint over a longer timespan.

4
Egalitarian Byzantine Fault Tolerance

Before a client request can be ordered by the replicas running the state-machine replication protocol, it has to arrive at them first. Often, agreement protocols require the request to be sent to a leader replica, which can add a significant amount of latency before the actual agreement starts if it is located far away from the client. Instead, our approach Isos allows each replica to immediately propose an order for requests and thereby clients can submit their requests to the nearest replica, for example, one running in the same region. Isos is based on only ordering conflicting requests in respect to each other, that is, those which access the same parts of the application state. If the replicas agree on the dependencies between such requests, then ordering is possible via a fast path that requires only three communication steps between the replicas. Otherwise, the replicas have to reconcile the dependencies via a single additional communication step.

Section 4.1 discusses the problems with existing approaches in regard to reducing the request submission latency. Afterwards Section 4.2 gives an overview of Isos, which only explicitly orders requests that conflict with each other. How replicas agree on dependencies between the requests is presented in Section 4.3. Then Section 4.4 derives a valid execution order based on the dependencies. Support for checkpointing, which is necessary to bound the state of a replica, is added in Section 4.5. We present a correctness proof in Section 4.6. Afterwards we introduce several optimizations in Section 4.7 to allow the protocol take the fast path more often, optimize its efficiency and improve its resilience to faulty replicas. Section 4.8 evaluates the request processing latency of Isos in comparison to two other systems. In Section 4.9 related work is discussed and Section 4.10 summarizes the chapter.

4.1. Problem Statement

In this chapter we focus on reducing the latency of the client communication to lower the overall request processing latency. More precisely, each replica that receives a client

request should be able to immediately initiate the agreement process. That way, a client can submit its request to the nearest replica to minimize the communication latency for the request submission.

We follow the approach of only establishing a partial order between conflicting requests to reduce the amount of coordination required between replicas. Two requests *conflict* with each other if executing them in a different order leads to diverging application states or replies, for example, when both requests modify the same state object. Requests that do not conflict with each other are also said to be *commutative*. As long as all conflicting requests are ordered in regard to one another, then the state of the replicas remains consistent [137, 169]. That is, the overall execution order may differ between replicas as long as conflicting requests are always executed in the same order. This allows lowering the processing latency for commutative requests.

In the following we present challenges regarding the agreement latency in Section 4.1.1 and avoiding a fallback to a single leader which can become a bottleneck in Section 4.1.2. Afterwards we discuss problems with resource efficiency in Section 4.1.3 and unbounded replica state in Section 4.1.4.

4.1.1. Reducing the Agreement Latency

The appeal of letting clients send their request to the - in latency terms - nearest replica is that it reduces the time between sending the request and the start of the agreement protocol. This leads to the challenge of ensuring that the replica is able to immediately propose newly arrived requests without increasing the agreement latency.

This rules out protocols like Aardvark [63] or Spinning [201], which rotate the leader, but at each point in time only have a single active leader. For minimal latency, clients would have to broadcast their request to every replica such that it reaches the next leader as soon as possible. However, this results in a high network overhead for the client and is affected by varying request processing latencies depending on the location of the current leader [82, 189] or may even be slower than a single well-placed leader replica [189] if a slow replica slows down the whole system.

Instead, a protocol should not rely on just a single leader replica, but allow for multiple replicas to process requests at the same time. One way for this is to decouple the request distribution and ordering. DBFT [69] and PRIME [22] first execute a broadcast phase in which each replica can independently distribute requests. The replicas then run an agreement phase to decide which requests to use and in which order. However, this separation increases the latency to 4 and 6 wide-area communication steps, respectively, not counting the client communication.

Protocols like EPaxos [162], CAESAR [29] or ATLAS [90] are more promising in regard to latency, as they allow each replica to start ordering requests without a central leader. The replicas agree on dependencies for a request and if a quorum of replicas agrees, then the request can be ordered via a fast path. Afterwards the dependencies are used to determine the final execution order and to ensure that conflicting requests are executed in a consistent order. The fast path requires only as many communication steps as a central leader replica would, but in contrast every replica is able to directly initiate the

agreement and does not first have to forward a request to the central leader replica. For optimal performance, ATLAS also includes an optimization allowing it to take the fast path more often, even if a dependency is proposed by only f replicas and not a full quorum. However, the just mentioned protocols can only tolerate crash faults and are therefore not suitable for us, as we target the Byzantine fault model.

We do not discuss ezBFT [30] in further detail even though it targets the Byzantine fault model, as it was shown to be incorrect [184] such that it is neither able to provide safety nor liveness.

Approach of this Thesis

Our approach Isos is like EPaxos based on the idea of letting replicas only explicitly order conflicting requests. If the replicas agree on the dependencies for a request, then it can be ordered via a fast path that only requires three communication steps. This is the same number of steps as for requests proposed by the leader replica in PBFT. Compared to EPaxos, Isos adds a third protocol phase allowing it to tolerate Byzantine faults and to consistently agree on the request together with its dependency set, which contains all conflicting requests that must be executed first. The requests are then executed in an order that conforms to the agreed upon dependencies.

To order as many requests as possible via the fast path, Isos uses an optimization, inspired by ATLAS, allowing the protocol to take the fast path, even if some dependencies are only proposed by $f + 1$ replicas instead of a full quorum.

As protection against malicious replicas, the replicas only accept dependencies for which they know that the referenced requests actually exist. This ensures that faulty replicas cannot introduce dependencies which are not satisfiable.

4.1.2. Always Using Multiple Leaders

In order to prevent a single replica from becoming the bottleneck, a protocol should not require a central leader replica neither for a fast path nor for a fallback mechanism.

The quorum-based HQ [68] lets a client issue its requests directly to a quorum of $3f + 1$ replicas, thus avoiding a central leader on the fast path. If enough replicas have the same state for the accessed objects, this allows the client to receive a reply in two round trips, that is, four wide-area communication delays. However, if the same objects are modified by multiple requests, then the protocol has to fall back to PBFT with its single leader replica to resolve conflicts. In addition to the possible leader bottleneck, this significantly increases the latency for ordering a request.

Byzantine Generalized Paxos [171] belongs to the family of *generalized consensus* protocols [137]. These allow multiple, selected replicas to concurrently propose new requests, as long as those do not conflict with each other. Each replica independently assembles a sequence of requests whose correctness has to be confirmed by a quorum of replicas. As long as only the order of non-conflicting requests differs, the concurrent ordering can proceed. However, in case of conflicts the agreement protocol has to fall back to a single leader replica, which then coordinates the decision on the correct order.

That way, the leader replica can become a bottleneck in addition to the protocol having to temporarily switch between modes.

Approach of this Thesis
In our protocol Isos, the replicas agree on dependencies for each request and sort the requests accordingly before execution. Cases in which two replicas propose different dependencies for conflicting requests, like in EPaxos, can only result in circular dependencies between the requests, but do not require a fallback to a central leader replica. Before request execution, all requests on a dependency cycle are sorted deterministically to guarantee a consistent execution order. Thereby, Isos can avoid falling back to a central leader replica if the replicas disagree on the dependencies of conflicting requests.

4.1.3. Being Resource Efficient

Without special trusted components [40, 203] the minimum number of replicas to tolerate Byzantine faults in a partially synchronous system is $3f + 1$ [49]. In order to make efficient use of its resources, a system should only require this number of replicas.

Q/U [1] lets clients optimistically issue its requests to a quorum of replicas. If enough replicas have the same state for the accessed objects, then the request can complete in a single round trip. Otherwise, it becomes the client's job to synchronize the state of the replicas. However, in order to guarantee correctness, Q/U requires a large quorum of $5f + 1$ replicas. Additionally, for conflicting requests clients have to perform an exponential back off, which can lead to significant delays.

Byblos [38] orders requests by assigning them non-skipping timestamps to define the execution order. The requests for a certain timestamp are executed once it is guaranteed that all new requests will use a higher timestamp. To further optimize the protocol, requests can be executed prematurely, if based on the request semantics it is clear that no more conflicts can arise with earlier requests that are still being ordered. As the timestamp assignment works by querying a quorum of servers for their latest timestamp, this allows the protocol to work without relying on a leader replica. However, it comes at the price of requiring at least $4f + 1$ replicas to guarantee the correct assignment of timestamps. Faulty clients can also make it necessary for the replicas to fall back to an agreement protocol to resolve diverging views between replicas.

Both systems require more than the minimum of $3f+1$ replicas which has the drawback of an increased resource consumption.

Approach of this Thesis
Isos's third protocol phase allows the replicas to verify the requests together with their dependencies before committing them and thus enables Isos to tolerate Byzantine faults while only using the minimum number of $3f + 1$ replicas.

4.1.4. Guaranteeing a Bounded State

For an agreement protocol to be practical, it must be able to work with a bounded state. In particular, this requires the protocol to garbage collect old state from time

to time. The traditional approach (cf. Section 2.3.3.5) is to let all replicas create a checkpoint after processing every k-th sequence number, which once confirmed by a quorum of replicas allows garbage collecting all earlier state. This requires all replicas to capture the application state after executing the exact same set of requests. However, with a generalized consensus protocol there is no longer a single order in which requests are executed. This makes it impossible to use the traditional approach of creating a checkpoint after executing a specific sequence number, such that replicas need a different way to coordinate the checkpoint creation.

Although necessary for a practical system, checkpointing is often neglected. For example, both EPaxos [162] and ATLAS [90] do not describe a checkpointing mechanism and to the best of our knowledge do not contain a proper checkpointing implementation. While it is possible in crash-fault tolerant systems to just request the full state from any of the replicas, this does not work when considering Byzantine fault tolerance.

Byzantine Generalized Paxos (BGP) [171] introduces a special command C* that is used to cut off the command sequence at that request. It must be proposed by the central leader replica and also requires a fallback to the slow path. However, no mechanism is described to force the leader to issue the special command in regular intervals.

Another problem for checkpointing in EPaxos-like protocols is that dependencies are contributed by multiple replicas such that a request can gather a dependency on a future request, which in turn can depend on an even newer one [175]. The requests in such a dependency chain can only be executed once it stops growing. These potentially endless chains can form spontaneously [162, 175] or faulty replicas can also try to generate such messages patterns intentionally. Both cases can therefore require unbounded amounts of memory and delay the request execution.

Approach of this Thesis

In order to allow garbage collection in Isos, we introduce so-called checkpoint requests whose execution causes a replica to create a new checkpoint. By conflicting with all other requests, they separate other requests into two groups of requests: before and after the checkpoint. Thereby, when processing a checkpoint request, all replicas are in the same state and consequently create identical checkpoints. These requests must be periodically proposed by each replica. As an infinite dependency chain could delay the execution of requests, it would also affect the checkpoint creation. To prevent this problem, we introduce a mechanism to limit the size of these dependency chains by deterministically splitting too long chains into smaller parts.

4.2. Isos - Egalitarian Byzantine Fault Tolerance

Isos is based on ordering requests concurrently if they do not conflict. This allows every replica to propose requests without first sending them to a central leader replica and thereby avoids one wide-area communication step. We start with an overview of how Isos processes requests in Section 4.2.1 and afterwards discuss how conflicts between requests are determined in Section 4.2.2.

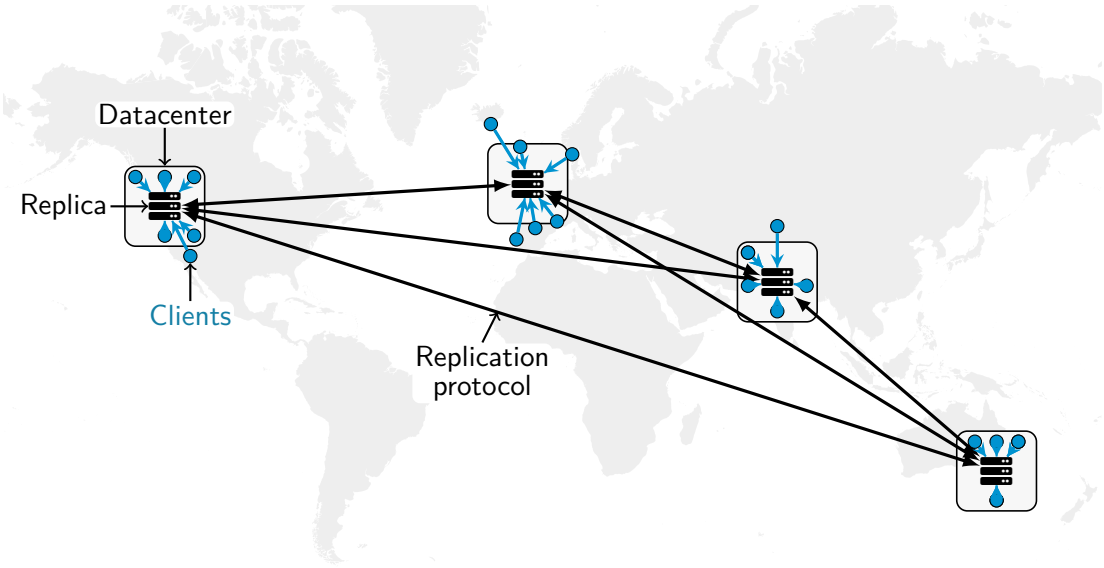


Figure 4.1: Basic system structure of Isos. Clients submit their requests to the nearest replica, which can immediately start the agreement process for the request.

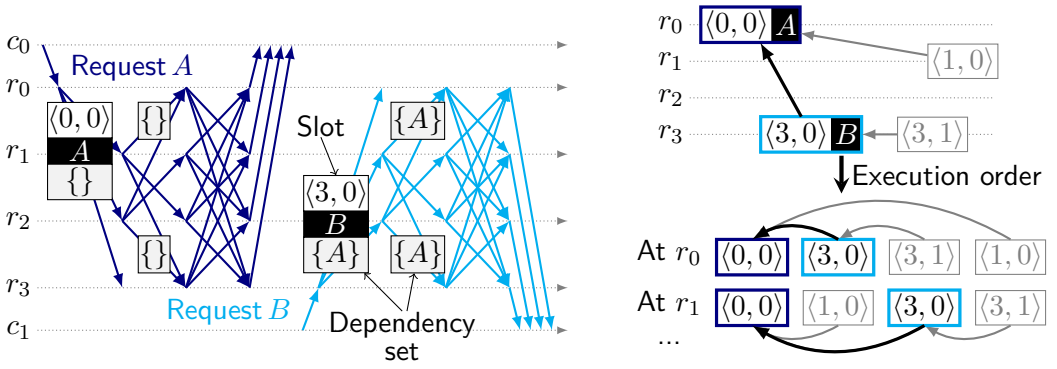
4.2.1. Request Processing

In Isos each of the $3f + 1$ replicas, which are located in different cloud regions, as shown in Figure 4.1, can independently propose new requests. This allows each client to submit its requests to the nearest replica such that this communication step is nearly for free, especially for collocated clients. To establish an order between conflicting requests, the replicas collect and agree on dependency sets that contain all previously proposed conflicting requests. If all involved replicas propose the same dependencies for a request, then the agreement can proceed on the fast path, allowing it to complete in three communication steps. Based on these dependencies, the execution component then sorts the requests such that all of their dependencies are executed first. For conflicting requests, at least one of them will depend on the other one, which ensures a consistent execution order across all replicas.

Sequence Numbers

The replica to which a client sends its request becomes responsible for ordering it and is called *request coordinator*, all other replicas are referred to as *followers*. This role distribution is determined individually on a per-request basis.

Each replica controls its own sequence number space for which it can propose new requests. A sequence number $s_i = \langle r_i, sc_i \rangle$ consists of the replica identifier r_i and a replica-specific counter sc_i. These sequence numbers are used to identify *slots* for which the corresponding replica can propose requests. Each replica stores the slots of all replicas as these are necessary to participate in the agreement for them.



(a) Agreement **(b)** Execution

Figure 4.2: Replicas r_i agree on dependencies between conflicting requests submitted by clients c_i and use those to determine an execution order. Replicas can execute requests in different orders, as long as the dependencies are satisfied.

Agreement

To order a request, as shown in Figure 4.2a, a client submits its request to the latency-wise nearest replica, for example, the replica collocated in the same region as the client. This replica then becomes request coordinator for the request, assigns the request to its next unused slot and calculates a dependency set containing all conflicting requests known to the replica. This limits the ordering between requests to the necessary extent. The proposal is then sent to a fast-path quorum consisting of $2f$ followers, which verify the assignment and also report dependencies. If all replicas report matching dependencies, then after an additional protocol phase, the request commits on the fast path, resulting in a total of three protocol phases. As the dependencies are determined as part of the agreement process, this ensures that all replicas learn the same dependencies.

Once a replica calculates and broadcasts a dependency set for a request, it promises to consider the request in all further dependency set calculations. Together with collecting dependencies from a quorum of replicas, this mechanism ensures that all conflicting requests are ordered by at least one dependency between them.

If the replicas propose different dependency sets for a request, which for example occurs if conflicting requests are ordered concurrently, then the request has to pass through the reconciliation path. The latter consists of two protocol phases to combine the dependency sets and to finally commit them. Together with the two initial phases of the agreement protocol, which are shared with the fast path, this results in a total of four phases. Conflicting requests that are ordered after each other, can nevertheless commit via the fast path as shown for request B in Figure 4.2a. Either way, once the agreement for a slot is complete, then all correct replicas use the same request and dependency set.

Execution

The ordered requests have to be sorted such that the dependencies of each request are executed first, as shown in Figure 4.2b. As conflicting requests are guaranteed to be connected by a dependency, this ensures that all replicas execute those requests in the

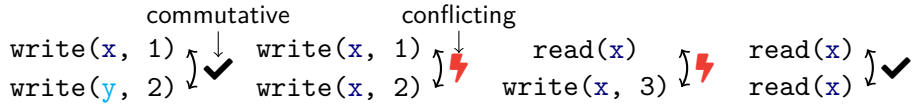


Figure 4.3: Conflicts between requests depending on the accessed state objects and operations.

same order. After executing a request, the result is sent to the client, which has to wait for $f + 1$ matching replies. This ensures that the result was sent by a correct replica.

Non-conflicting requests can execute in a different order on each replica. As these requests are commutative, they nevertheless yield the same results on all replicas. In addition, replicas are more likely able to execute a request without having to wait for other requests. The safety property discussed in Section 2.1.3 still holds, as it is sufficient if the clients cannot tell the execution order apart from a single total order [137, 169].

Bounded State

In order to bound the state required by the protocol, Isos periodically garbage collects old slots. This requires the replicas to create and collect a stable checkpoint. As Isos no longer has a single sequence number that can be used to periodically trigger the creation of a checkpoint, each replica instead injects checkpoint-request messages into the ordering process in regular intervals. These checkpoint requests conflict with every other request and thus establish a barrier that divides requests into before and after. The snapshot of the application data is created when executing the checkpoint request, which ensures that each replica is in the same state as it has executed the exact same set of requests. Once a replica collects a certificate for the checkpoint, it becomes stable and the replica can garbage collect all requests before the checkpoint barrier.

4.2.2. Conflicts between Requests

In order to allow a replica to check for conflicts between requests, the application must provide a predicate $\mathit{conflict}(a, b)$, which for two requests a and b determines prior to execution whether the requests could conflict with each other or are commutative. That is, requests commute if the application state or replies are independent of the execution order of both requests.

It is often feasible to provide such a predicate [38, 74, 90, 131, 162]. In general, requests accessing disjunct sets of state object are commutative. For example, for the conflict calculation of a key-value store it is possible to rely on the operation and the accessed key, which both are part of the request and are thus known before execution. This is illustrated in Figure 4.3, which we discuss from left to right. Requests accessing different objects do not conflict with each other. Writes of an object conflict with all other writes and reads for that object, whereas reading the object does not conflict with other reads of the object. In doubt the predicate must be conservative and report a (possible) conflict; it may overestimate conflicts but must not underestimate them. That is, it would be safe, but inefficient, to always report a conflict.

Like EPaxos [162] we target use cases with low conflict rates of less than 5%. These match systems like lock services such as Chubby [52] where more than 90% of the requests access client-specific data structures and less than 1% are write requests that can conflict with each other, or services with a high read to write ratio like Google's advertising backend F1 where less than 0.3% are write requests [67].

Batches

To determine the conflicts for a request batch, the predicate has to check each individual message within a batch for conflicts. Dependencies nevertheless apply to the batch as a whole. As a special case, messages within the same batch cannot conflict with each other. For those the relative execution order is already fixed, thus making it unnecessary to add further dependencies.

Requests from the Same Client

In addition to conflicts that stem from the application semantics, it is also required to establish an order between the individual requests of a single client. Requests in Isos include a client-specific counter value, which is used to skip old and duplicate requests during the execution. This makes it necessary that all replicas execute the requests of a specific client in exactly the same order, as otherwise some replicas could skip a request whereas it would be executed on other replicas. That is, the *conflict* predicate must also report conflicts between all requests of the same client. Thereby, faulty clients that propose two different requests with the same counter value cannot cause replicas to execute diverging requests. As both requests conflict with each other, the replicas will agree on a consistent order in which to execute those requests.

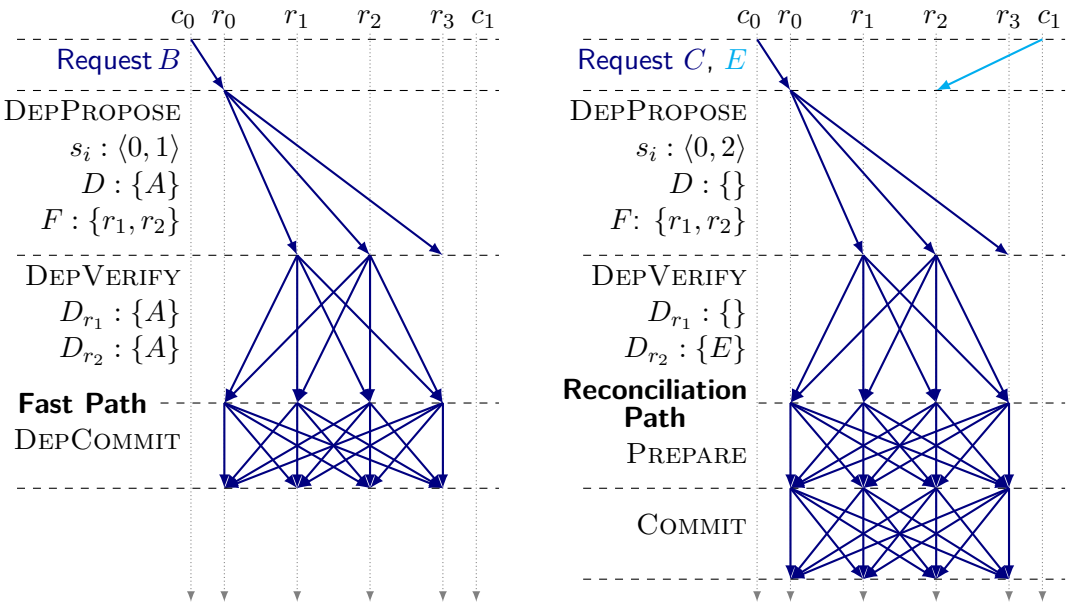
4.3. Request Ordering

In the following we describe in detail how requests are ordered. In Section 4.3.1 we first present how the client interaction in Isos works. Afterwards in Section 4.3.2 we discuss the fast and reconciliation path, which are used to order requests in the normal case. We describe in Section 4.3.3, how the replicas perform a view change to selectively recover failed slots. And finally in Section 4.3.4, we conclude with the mechanisms used by Isos to guarantee liveness. The protocol's pseudocode is available in Appendix A.

4.3.1. Client Handling

In order to submit a command w for execution, a client c has to send its signed request $r = \langle \textsc{Request}, w, t_c \rangle_{\sigma_c}$ to the nearest replica. The client counter value t_c must increase monotonically for each new request and is used to filter out old or duplicate requests. The Request is signed to ensure that all replicas can verify its validity.

The client then waits until it receives a certificate of $f + 1$ matching authenticated replies $\langle \textsc{Reply}, u_c, t_c \rangle_{\mu_{r_i}}$, which confirms the correctness of the result u_c. If the client does not collect a stable result within a certain timeout, then it broadcasts its request to all replicas. In case the nearest replica to a client repeatedly fails to order requests in time, then the client switches to another replica to submit its requests.



(a) Fast Path **(b)** Reconciliation Path

Figure 4.4: A request can be ordered on the fast path if all proposed dependencies match, or has to take the reconciliation path to agree on the final dependency set.

4.3.2. Fast Path and Reconciliation Path

Ordering a request proceeds in multiple phases to ensure that all correct replicas agree on the request as well as its dependencies. As first step, the replicas collect dependencies in a DepPropose and DepVerify phase. Depending on whether the involved replicas propose matching dependencies or not, the protocol then has to follow the fast or reconciliation path, respectively.

Request Proposal

Once a new request r arrives at a replica, it acts as *request coordinator co*. As shown in Figure 4.4a, the replica picks its first own unused agreement *slot* with the sequence number s_i and assigns the request to it. For brevity, we refer to it as *slot* s_i. Old requests, that is those with a counter value t_c that was already processed for the client c, are either ignored or result in sending back the reply again. In general, if a message is not signed correctly, then it is silently dropped by the replicas.

As next step, the coordinator calculates the dependency set D for the request, which includes all conflicting requests known to the request coordinator. In particular, this also includes all prior requests of client c. Afterwards the request has to be considered for all future dependency calculations. The coordinator also selects a fast-path quorum F containing $2f$ followers. By default, the replicas with the lowest communication delay to the request coordinator are selected. The request is then broadcasted in a signed $\langle\langle \text{DepPropose}, s_i, h(r), D, F\rangle_{\sigma_{co}}, r\rangle$ message to all followers, where $h(r)$ is the hash of the client request.

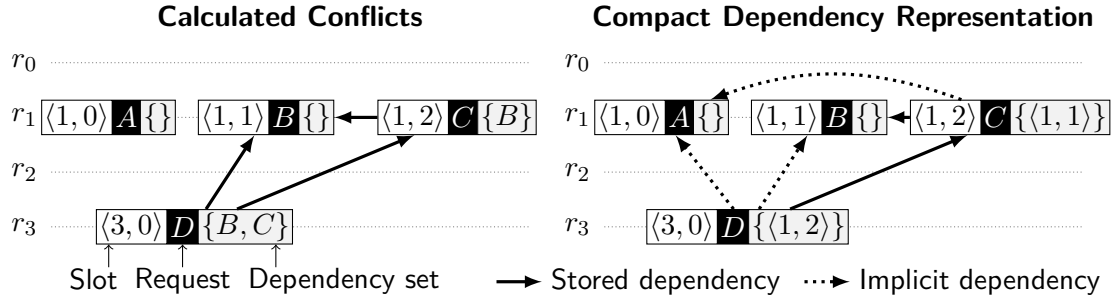


Figure 4.5: Compact dependency encoding, which only stores the dependency with the latest sequence number for each replica.

Compact Dependency Encoding

For each slot, the replicas determine dependency sets, such as those depicted on the left side of Figure 4.5. However, with many conflicting messages, the dependency sets could grow very large. To prevent this, similar to the approach used by EPaxos [162], a dependency set for each request coordinator only explicitly stores the dependency on the slot with the highest sequence number. All earlier slots of each request coordinator are then implicitly included as dependencies, as shown on the right side of Figure 4.5. This results in a *compact dependency encoding* that only has to store one sequence number for each replica. That is, the encoding has a fixed size, which prevents faulty replicas from generating huge dependency sets to flood the other replicas. The additional dependencies do not affect correctness, as the *conflict*(a, b) predicate is allowed to overestimate conflicts between requests.

Another major benefit of the compact encoding is that it allows for a significantly simplified calculation of conflicts. Instead of checking all known requests, for example for a key-value store, the conflict calculation has to remember for each state object per request coordinator the latest read and write request that accessed the state object. The conflict calculation for a request then only has to look up for each request coordinator the latest request that accessed the same object; all older conflicting requests are then also implicitly covered. This both speeds up the dependency calculation and at the same time limits the state necessary to efficiently determine conflicts between requests.

Dependency Verification

Once a follower f_i receives a correctly signed DepPropose from the request coordinator, it starts to verify the message. The accompanying request r must match the hash $h(r)$ included in the DepPropose message, the slot s_i must belong to the request coordinator and not yet contain a different message, and the fast-path quorum must contain a valid selection of $2f$ different followers. If any of these conditions is not satisfied, then the follower discards the message. In addition, a follower strictly processes DepPropose messages from a coordinator in sequence number order. This guarantees that a faulty request coordinator cannot skip sequence numbers. Otherwise, missing slots would eventually block the request execution, as a dependency on a later slot also results in implicit dependencies on all earlier slots potentially including the missing ones.

A follower then waits until it learns that the ordering process has been started for all dependencies included in the DepPropose. This ensures that a faulty leader cannot include dependencies to nonexistent requests and thereby try to prevent the request from being executed. A dependency on a slot is *known* once the follower has either completed the dependency calculation for a corresponding DepPropose, has received $f + 1$ matching DepVerify messages or has started a view change for the slot. Each case guarantees that the slot will eventually commit and thus will allow dependent slots to execute. We describe these progress guarantees in further detail in Section 4.3.4.

Each follower then calculates a dependency set D_{f_i} and remembers the request for future conflict calculations. Only the followers included in the fast-path quorum F broadcast a signed $\langle \text{DepVerify}, s_i, h(dp), D_{f_i} \rangle_{\sigma_{f_i}}$ to all replicas where $h(dp)$ is the hash of the DepPropose message. The latter allows checking that all replicas belonging to the fast-path quorum have received the same DepPropose.

Fast-Path Commit

A replica r_i accepts a DepVerify only if it matches the received DepPropose, the sender is part of the fast-path quorum F and after all dependencies included in the message are known. After collecting a quorum certificate consisting of a DepPropose and $2f$ matching DepVerify messages, the replica checks whether the slot can commit on the fast path. DepVerify messages are considered matching if they refer to the same sequence number s_i and contain the correct hash $h(dp)$ for the DepPropose.

To commit on the fast path, the *fp-verified* predicate must hold, which requires the DepPropose and the corresponding $2f$ DepVerifys to contain identical dependency sets. Once the slot is *fp-verified*, the replica broadcasts a $\langle \text{DepCommit}, s_i, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ message where $h(\vec{dv})$ is the hash of all used DepVerify messages. If the request coordinator and the replicas in the fast-path quorum are correct, then all replicas will use the same DepVerify messages and thereby calculate the same hash.

Once a replica has collected $2f + 1$ matching DepCommit messages, the slot becomes *fp-committed*, which ensures that the request and its dependency set can no longer change even in case of failures. As the DepCommit contains the hash of all used DepVerifys and indirectly the DepPropose, this guarantees that all replicas have processed the same messages. The request and its dependency set is then passed to the execution.

Reconciliation Path

If the *fp-verified* predicate cannot be satisfied due to mismatching dependency sets contained in the DepPropose and DepVerifys, the replica switches to the reconciliation path as shown in Figure 4.4b. The replica then no longer processes messages which belong to the fast path. This ensures that completing the fast and reconciliation path is mutually exclusive.

In order to ensure agreement on the dependency sets, each replica broadcasts a $\langle \text{Prepare}, s_i, v_{s_i}, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ message. The slot-specific view number v_{s_i}, which we discuss in the next section, is initialized to -1. After a replica has collected a quorum certificate of matching Prepare messages for the current view, the slot becomes *rp-prepared*. This ensures that all replicas are aware of the same dependency sets.

Each replica then broadcasts a $\langle \text{COMMIT}, s_i, v_{s_i}, h(\vec{dv}) \rangle_{\sigma_{r_i}}$ message. After collecting a quorum certificate of matching COMMITs, the slot becomes *rp-committed* and the replica forwards the request together with the union of all dependency sets to the execution.

Invariant
For correctness, a slot must only commit either via the fast path or the reconciliation path (unless there is a view change). The protocol guarantees that only either *fp-committed* or *rp-prepared* can be satisfied at any replica. Sending a DEPCOMMIT or a PREPARE are mutually exclusive, as a correct replica will not enter the reconciliation path (without a view change) after reaching *fp-verified* and vice versa. As both *fp-committed* and *rp-prepared* require matching messages from a Byzantine majority quorum, these would overlap in at least one correct replica that, however, will only send the message required for either the fast path or the reconciliation path.

4.3.3. View Change

If the agreement process for a slot does not complete within a certain timeout, a replica initiates a view change. We discuss how to select the timeout in Section 4.3.4.

In contrast to other protocols like PBFT [57], in ISOS the view change works on a per-slot basis which means that each slot s_i has its own view v_{s_i} and starts in view -1. The per-slot view change allows the replicas to selectively recover slots and avoids the overhead of reprocessing all slots since the latest checkpoint.

The view change has to ensure that a committed slot keeps its value along with its dependencies and that the value selected by the view change has correct dependencies. Therefore, for a slot it either selects the request which might have committed on the fast or reconciliation path together with its dependencies, or fills the slot with a no-op message. That is, the slot either contains the request that was initially proposed and used in the dependency calculations, or a no-op, which by construction does not conflict with any requests and consequently does not require any dependencies. Substituting a not yet committed request with a no-op is possible, as the dependency encoding references slots instead of requests.

Once a replica initiates a view change, it stops processing messages from the old view for that slot. The replica then broadcasts a $\langle \text{VIEWCHANGE}, s_i, v'_{s_i}, cert \rangle_{\sigma_{r_i}}$ message, which contains the new view number v'_{s_i} and a certificate *cert* proving that a request with the given dependencies was prepared. The certificate has one of the following types:

Fast-Path Certificate (FPC) A fast-path certificate (FPC) consists of a DEPPROPOSE and $2f$ matching DEPVERIFY messages from different followers that all contain identical dependencies. These messages prove that the slot was *fp-verified*.

Reconciliation-Path Certificate (RPC) A reconciliation-path certificate (RPC) consists of a DEPPROPOSE, $2f$ matching DEPVERIFYs and $2f+1$ matching PREPAREs from different followers. All PREPARE messages must be from the same view. Together, these messages prove that the slot was *rp-prepared*.

A replica includes its latest, that is, from the highest view, reconciliation-path certificate in the ViewChange message, or as a fallback a fast-path certificate. If neither exists, then *cert* is set to a placeholder $\perp$.

View Synchronization

Once a replica receives $f + 1$ valid ViewChange messages from different replicas for slot s_i that contain a higher view number than the replica is in, it switches to the $f + 1$-highest received view number and sends a corresponding ViewChange message.

NewView Calculation

The request coordinator responsible for the new view v'_{s_i} of this specific slot s_i is calculated using $co = \left(s_i.r_i + \max(0, v'_{s_i})\right) \bmod N$, where $s_i.r_i$ is the replica identifier part of the sequence number s_i. This calculation gives the original request coordinator a second chance to complete the agreement if a replica in the fast-path quorum F was faulty.

Once the new request coordinator has received $2f + 1$ valid ViewChange messages for the same view, it starts to reconstruct the agreement state. It either selects the latest included certificate, which is a proof that some request with certain dependencies was prepared, or if no certificate exists, then a no-op message, which will be skipped during execution. The certificate selection prioritizes the RPC with the highest view and uses an FPC only as fallback. If both a valid RPC and FPC exist, then the RPC was generated as the result of a view change and is therefore newer than the FPC.

The request coordinator broadcasts the result of the certificate selection in the form of a $\langle \text{NewView}, s_i, v_{s_i}, dp, \vec{dv}, VCS \rangle_{\sigma_{co}}$ message, where dp and $\vec{dv}$ are the DepPropose and DepVerifys belonging to the selected certificate, or in case a no-op was selected, both values are filled with placeholders. VCS is the set of the $2f + 1$ ViewChange messages used to create the NewView. When a follower receives the NewView, it repeats the certificate selection process to verify its correctness. If successful, the agreement continues in the reconciliation path using the selected dp and $\vec{dv}$.

In case a request was replaced with a no-op, the original request coordinator has to retry ordering the request in a new slot.

4.3.4. Progress Guarantees

In the following we describe how Isos ensures liveness during synchronous phases of the network by using timeouts, agreement result forwarding and the fast-path quorum selection. During asynchronous periods, the protocol's progress may stall until the network becomes synchronous again.

Propose Timeout

With the protocol described so far, a faulty coordinator *co* could create a DepPropose for slot s_{co} containing a conflicting request A and only send it to replica r_i, which then has to include the request as a dependency in the future. As just a single replica has received that request, the agreement for the corresponding slot of the faulty coordinator will not complete. However, this would also prevent new proposals issued by the correct

replica r_i that include a dependency on request A from making progress, as these would be blocked until the other replicas know about slot s_{co}.

To prevent such an attack, after receiving a DepPropose, a follower starts a *propose timer* with a timeout of 2Δ, where Δ is the maximum one-way communication delay between replicas during synchronous phases (cf. Section 2.1.2). The timer is stopped once the follower receives $2f$ matching DepVerifys with known dependencies, which ensures that all replicas will know about the slot. If the timer expires or a view change is triggered for the slot before that, then the replica broadcasts the signed part of the DepPropose message (i.e., without the actual request) to all replicas. This reduces the network traffic required to ensure a reliable distribution of the DepPropose and also prevents a faulty coordinator from forcing other replicas to distribute the requests for its message slots. As the DepPropose is signed, it proves that the request coordinator initiated the agreement process for that slot. Thus, if a correct replica includes a dependency to that slot, then all replicas will eventually know the slot.

The timeout is chosen to not expire in the normal case. If a correct replica broadcasts a DepPropose to all replicas, it reaches them after at most Δ and the resulting DepVerifys arrive at all replicas after 2Δ, which is within the propose timeout. While the network is in a synchronous phase, longer delays only arise due to faulty replicas.

Commit Timeout

To force a slot to commit eventually, as soon as a replica learns that the agreement process for a slot was initiated, it starts a *commit timer* with a timeout of 8Δ. Once the timer expires, the replica triggers a view change for the slot. A replica considers the agreement process for a slot as started after broadcasting the DepPropose as request coordinator, completing the dependency verification for a DepPropose, or receiving $f+1$ DepVerify messages for the slot. The latter proves that at least one correct replica has learned about the slot's existence. Together the propose and commit timeout ensure that eventually every correct replica will learn about the existence of a slot and either commit it or request a view change. The timer must be used by the request coordinator to monitor its own slots to detect problems within the fast-path quorum.

A DepPropose or DepVerify message x sent by a correct replica will be accepted after 3Δ by other correct replicas. The message x itself is delivered within Δ; however, the receiving replica may have to wait until it can verify the dependencies. A correct replica only includes dependencies in x for which it has already seen the DepPropose. That is, the propose timer, which expires after 2Δ, is already active for them, such that after an additional Δ all replicas either already know the dependencies or receive the corresponding DepPropose. Thus, after 3Δ each replica has learned about every direct dependency and will accept message x. The same argument also recursively applies to all dependencies included in the DepPropose, whose propose timers must already expire earlier on.

In total, the DepPropose and DepVerify phases each take at most 3Δ. The DepCommit phase to finish the fast path completes in Δ, whereas the reconciliation path requires up to 2Δ for the Prepare and Commit phase. As a replica either completes the fast or reconciliation path, the maximum delay required is 8Δ.

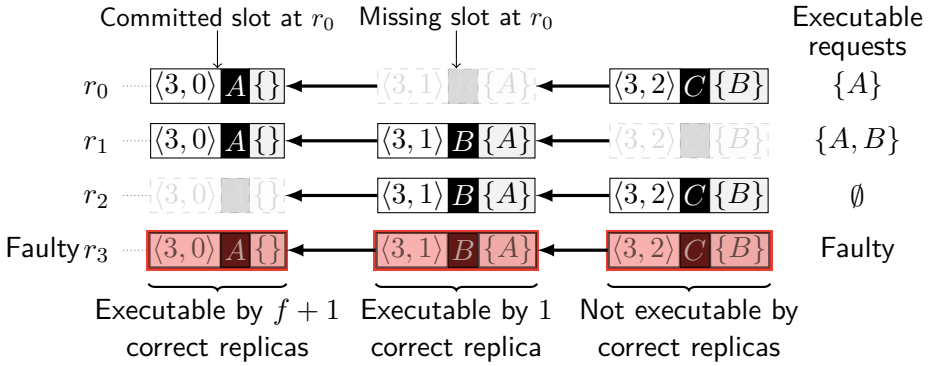


Figure 4.6: Blocked request execution caused by the faulty request coordinator r_3, which selectively excludes up to f correct replicas from the agreement process. The f excluded replicas per slot cannot trigger a view change.

View Change Timeout

Similar to PBFT, the new request coordinator has to complete the view change within a certain timeout. For Isos, once a replica has received $2f + 1$ ViewChange messages, it starts a *view-change timer* set to 3Δ. If the timer expires before the replica receives a valid NewView, then the replica switches to a new view. Once $2f + 1$ replicas, which include at least $f + 1$ correct replicas, have sent a ViewChange, it is guaranteed that after Δ all other correct replicas will also send their ViewChange. Therefore, the request coordinator receives $2f + 1$ ViewChange messages after 2Δ allowing it to complete the NewView. The latter arrives at all replicas after at most 3Δ.

Afterwards the reconciliation path has to complete within another 3Δ that are necessary to distribute the NewView and to complete the Prepare and Commit phases.

Agreement Result Forwarding

A faulty request coordinator can exclude up to f correct replicas from the agreement process [42, 109] by not sending them the initial message of the agreement protocol, in case of Isos the DepPropose message. This prevents the selected replicas from completing the agreement for the affected slots. As only f correct replicas are affected, these are unable to initiate a view change on their own, because the other replicas cannot distinguish whether they are correct or faulty. This is not a problem for a protocol with a single leader replica, as the latter gets replaced if the protocol becomes stuck.

However, this approach is not always possible with independent view changes for different slots. A faulty leader replica can prevent overall progress by omitting messages to different sets of correct replicas [109]. For each individual slot only up to f correct replicas are affected, which prevents a view change from occurring. However, the remaining correct replicas can get stuck in their execution at different slots thus preventing any progress. As shown in Figure 4.6, for the requests A, B and C, where C depends on B and B on A, f different correct replicas are excluded for each request. Thus, no correct replica can execute request C and a client would never receive $f + 1$ replies. At the same time no view change would be possible, causing the protocol to lose liveness.

In Isos the f omitted replicas will learn about the slot and eventually initiate a view change. However, the latter cannot complete, as explained previously. In order to resolve this situation, once a replica detects that a view change does not make progress, it starts to query all replicas for their agreement result for slot s_i using $\langle \text{QueryExec}, s_i \rangle_{\sigma_{r_i}}$ messages. After collecting a certificate of $f+1$ matching $\langle \text{Execute}, s_i, r, D \rangle_{\sigma_{r_i}}$ messages containing the agreed-upon request r and the corresponding dependency set D, it updates its state accordingly.

Crashed Replicas
A crashed replica has only limited impact on the system. Once all slots of this replica that were getting ordered have completed and the other replicas have adapted their fast-path quorums, a crashed replica has no longer an impact on the system. As it cannot propose new requests, other replicas only add dependencies to already ordered slots of the crashed replica, which thereby cannot delay the ordering of newer requests.

Fast-Path Quorum Selection
The request coordinator initially selects the fast-path quorum F to contain the replicas to which it has the lowest round-trip times. A replica can either use data provided by an administrator or for example measure the time between sending a DepPropose and receiving the corresponding DepVerify from a replica.

If the fast-path quorum contains faulty replicas, these can prevent the slot from committing. In this case, the request coordinator has to replace replicas in the fast-path quorum and restart the agreement for the request until it completes. This ensures that eventually all replicas in F are correct, allowing the request to commit. In the worst case a request coordinator has to try out all possible variants of the fast-path quorum, which is nevertheless still feasible for a small number of faulty replicas. In Section 4.7.5, we describe how to optimize this quorum selection process.

4.4. Request Execution

After the replicas have agreed on the request and dependencies assigned to a slot, the next step is to determine the execution order. We first discuss in Section 4.4.1 how requests are executed in general and then describe in Section 4.4.2 how Isos bounds the state required for the request execution despite the possibility of infinite dependency chains between slots. Executing such slots also requires a special treatment that is presented in Section 4.4.3. Section 4.4.4 concludes with discussing how to efficiently manage the data necessary for the dependency calculations.

4.4.1. Standard Execution Approach

The execution has to ensure that all conflicting requests are executed in the same order on every replica. To do so, their order is determined based on the dependencies between requests. If a request B depends on request A, then A should be executed before B, as illustrated in Figure 4.7. The dependencies for a slot are collected from multiple replicas such that it is possible for cyclic dependencies to exist, which require special handling.

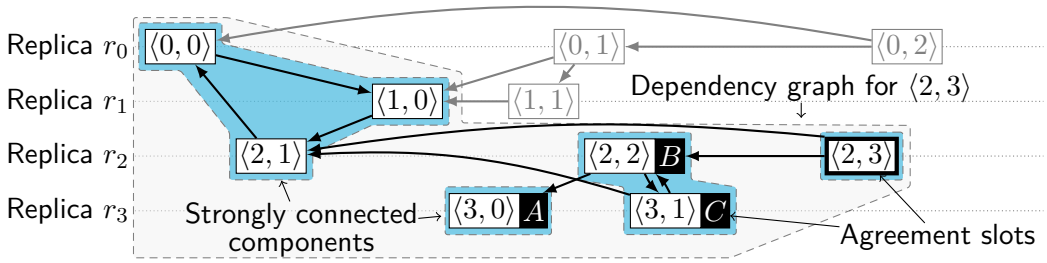


Figure 4.7: Calculation of the dependency graph and the contained strongly connected components for slot $\langle 2,3\rangle$ to determine the execution order.

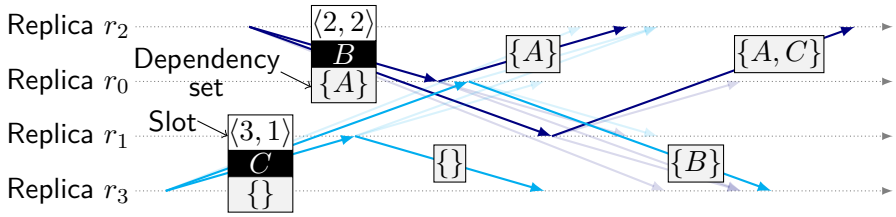


Figure 4.8: Dependency collection for the conflicting requests B and C from Figure 4.7. For clarity, only the DepPropose and DepVerify messages from and to replica r_2 and r_3 are shown. The cyclic dependency results from the fact that replicas r_0 and r_1 process both requests in a different order.

For example, as shown in Figure 4.8 the requests B and C conflict with each other, which can result in both requests collecting the other one as a dependency if they are ordered concurrently. Thus, in Figure 4.7 both B and C depend on each other and therefore are blocked by waiting until the other one is executed. As shown by EPaxos [162], after all slots of a cycle have committed, the cycle contains the same slots on all replicas, such that it is possible to collect all slots of a cycle, deterministically sort and execute them. This ensures a consistent execution order on all replicas. Requests that do not conflict with each other can be executed independently and in a different order on each replica.

The execution proceeds as follows. Once a slot has committed, a replica builds the dependency graph for the slot by recursively expanding its dependencies and the dependent slot's dependencies and so on, as illustrated in Figure 4.7 for slot $\langle 2,3\rangle$. Slots correspond to graph nodes and the dependencies form directed edges between the nodes. A slot can have one of the following states. It can be *missing* if it has not committed yet, *waiting* for missing dependencies, *executable* or *executed*. Already executed slots are skipped while building the graph. If some slot in the dependency graph is missing, then the expansion has to wait until it is committed. After the dependency graph is complete and all slots are executable, the execution calculates the strongly connected components[1]. These are the largest possible subsets such that each slot in a component

[1]The calculation can, for example, use Tarjan's strongly connected components algorithm [195] which also returns the components in inverse topological order.

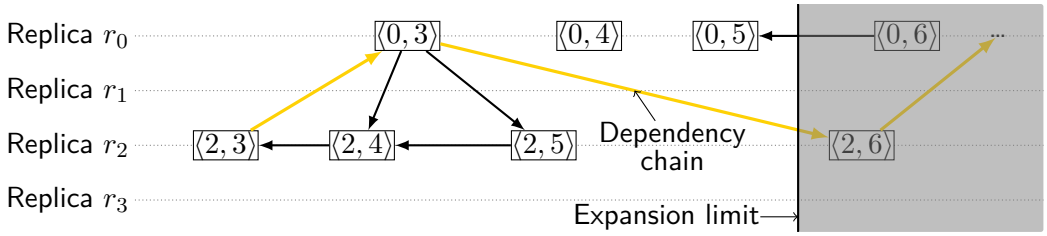


Figure 4.9: Dependency chain blocking the execution of slot $\langle 2,3 \rangle$ for an expansion limit with a window size of $k_x = 3$. The dependency chain is marked using a yellow arrow →.

depends on every other one in that component. As shown in Figure 4.7 each component either contains a single slot or multiple slots forming a cycle. These components are then processed in inverse topological order, which ensures that the dependencies of a slot are executed first. To ensure a consistent execution order, the slots within each strongly connected component are additionally sorted according to their sequence number before executing the corresponding requests.

Client requests are only executed if their counter value t_c is larger than the latest value stored for the client. Afterwards the execution result u_c is sent to the client using a $\langle \text{REPLY}, u_c, t_c \rangle_{\mu_r}$ message.

4.4.2. Limiting Dependency Chains

As the dependencies for a slot are collected from multiple replicas, which know about different slots, it is possible for nearly infinite dependency chains to form in which a slot depends on one with a higher sequence number, which in turn depends on a slot with even higher sequence number and so on [162, 175]. An example of such a chain is sketched in Figure 4.9. It can arise during standard request processing and theoretically get extended as long as new requests are proposed. A faulty replica could also intentionally introduce additional dependencies to enforce the creation of such a dependency chain.

To prevent infinite dependency chains from exhausting the available memory and from blocking the request execution for too long, the execution component only considers requests within a bounded *execution window* for each request coordinator. It contains k_x slots per request coordinator and starts at the slot with the lowest sequence number that is not yet executed. We say that the upper bounds of the execution windows of the request coordinators form an *expansion limit*. All requests for slots after the expansion limit are enqueued and not executed until the window has advanced sufficiently.

This adds a new slot state called *future*, which means that the slot is after the expansion limit. When expanding the dependency graph, future slots also count as missing and thereby prevent the request from executing. While this ensures that the replicas maintain a consistent request execution order, it can prevent slots from being executed if a dependency cycle or chain does not fit within the execution window.

In Figure 4.9 only the first $k_x = 3$ requests are currently considered for execution which prevents slot $\langle 2,3 \rangle$ from executing due to a dependency on a future slot.

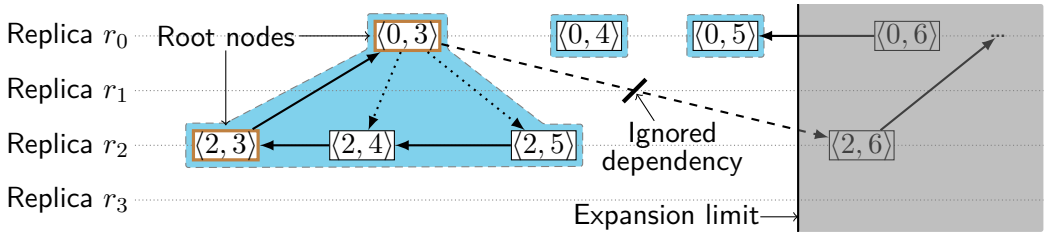


Figure 4.10: Unblocking the execution by ignoring dependencies across the expansion limit with a window size of $k_x = 3$.

4.4.3. Unblocking the Execution

In order to unblock the execution, the replicas have to deterministically ignore dependencies on slots beyond the expansion limit to allow the request execution to continue. For correctness, we must ensure that all replicas ignore the same dependencies while processing the same dependency graph.

The modified execution procedure works as follows. A replica first normally executes all requests within the execution window until no further requests are executable. Afterwards for the first slot in the execution window of each request coordinator, the so-called *root nodes*, the execution checks whether these would be executable when ignoring dependencies to future slots. In Figure 4.10 this is the case for root nodes $\langle 0, 3\rangle$ and $\langle 2, 3\rangle$. If this special case applies, then the execution temporarily removes these future dependencies and starts executing the root node's dependency graph. For Figure 4.10, this affects the dependency to slot $\langle 2, 6\rangle$. After the first strongly connected component is processed, the execution procedure switches back to the standard execution method. This is repeated until no further requests can be executed.

With the compact dependency encoding, a dependency on a slot also implicitly adds dependencies on all earlier slots of the same request coordinator. When ignoring dependencies to future slots, these implicit dependencies to slots before the expansion limit still have to be considered as shown in Figure 4.10, that is, the dependency to $\langle 2, 5\rangle$ and earlier slots is still relevant.

We give an intuition why this approach ignores the same dependencies on all replicas when executing requests via the special case, a detailed proof can be found in Appendix A.3.2. Due to the compact dependency encoding, a dependency on a future slot also implicitly results in a dependency on the root node for the same replica. The root node in turn must also directly or indirectly depend on some future slot, as otherwise it would already have been executed[2]. This results in a dependency cycle encompassing the full execution window of one or more request coordinators, which causes all replicas to deterministically block at the same root nodes. A root node can then be part of this cycle or depend on it in which case the cycle is executed first. Consequently, this cycle is the strongly connected component that will be executed by the special case. The size of

[2]If the root node or one of its dependencies were missing, this would prevent the execution in every case. Thus, we have to assume that the whole dependency graph of the root node is already committed.

this component is defined by the execution window size, which is identical on all replica, and therefore guarantees that the same dependencies are ignored. For a given root node this results in identical dependency graph modifications on all replicas and therefore a consistent execution order.

4.4.4. Graph Management

The dependency graphs constructed for different slots usually overlap in parts. Thus, instead of constructing a full dependency graph for each request, it is more efficient to construct a single graph containing all slots within the execution window. The dependency graph for a slot x is then the subgraph containing all slots reachable from the slot x. In order to avoid building a separate graph to unblock the execution, we modify the search for strongly connected components such that it dynamically ignores dependencies to slots after the expansion limit in this case.

To efficiently execute requests out of order, for each slot the graph must also store a set containing all slots which depend on the given slot. After executing a slot, all dependent slots have to be checked for whether these can also be executed. This has to be repeated until no new slots become executable.

Taken together these mechanisms allow Isos to efficiently execute slots within the execution window as soon as they become executable.

4.5. Checkpointing

The replicas have to create checkpoints and run the garbage collection in regular intervals in order to bound the size of the agreement and execution state. The content of a checkpoint must be confirmed by at least $2f + 1$ replicas to guarantee its correctness. This in turn requires checkpoints from different replicas to be comparable, that is, they have to capture the state at the same logical point in time. However, with Isos each replica can execute commutative requests in a different order, such that no natural point exists at which the replica's progress is guaranteed to be identical. In Section 4.5.1, we first introduce checkpoint requests, which are ordered similar to normal requests in order to act as a barrier and consistently split requests into those before and after the checkpoint. Afterwards Section 4.5.2 details how checkpoints are created when executing these requests.

4.5.1. Checkpoint Requests

In Isos, each replica must propose a checkpoint request $\langle \text{CheckpointReq} \rangle$ when reaching a slot for which the replica-specific counter sc_i in sequence number $s_i = \langle r_i, sc_i \rangle$ is a multiple of the checkpoint interval k, that is, $sc_i \mod k \equiv 0$. Such a slot must not be used for any other request. A checkpoint request conflicts with every other request, even reads or no-ops, and thereby acts as a barrier separating slots in before and after the checkpoint as shown in Figure 4.11. By processing the request via the agreement protocol, the checkpoint request and all other requests are ordered relative to each other,

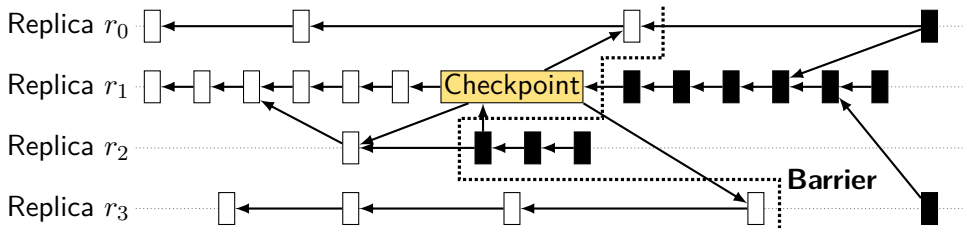


Figure 4.11: A checkpoint request acts as a barrier separating normal requests into before and after the checkpoint.

ensuring that the checkpoint request acts as a barrier. The dependency set created while ordering the checkpoint request then describes which requests will be covered by the corresponding checkpoint.

To limit the state of the agreement protocol, for each request coordinator, a replica only processes slots within a bounded window containing at least $2k$ slots[3]. The window starts after the barrier of the latest stable checkpoint and bounds the state the agreement has to keep. Similar to protocols like PBFT [57], slots before the barrier are garbage collected. Messages for slots after the window are dropped to bound the state and have to be retransmitted later on once the window has advanced sufficiently.

View Change

The view change requires an extension to properly handle checkpoint requests, as it is otherwise possible that a checkpoint request is replaced with a no-op. This would result in not creating a new checkpoint, which can cause the agreement to run out of unused slots such that the replicas become stuck. In addition, checkpoint requests and no-ops created by the view change have to be ordered in respect to each other.

As the checkpoint requests are tied to slots with fixed sequence numbers, each replica already knows the request for this slot, allowing Isos to provide a fallback for this case. Similarly, it is known that for all other slots the fallback is a no-op request. We refer to these fallback requests as *default request*. During the view change, each replica r_i extends its ViewChange message with a synthetic $\langle \text{DepVerify}, s_i, h(dr), D_{r_i} \rangle_{\sigma_{r_i}}$ unless it has a valid fast or reconciliation path certificate. The DepVerify contains the hash of a default request dr instead of a normal DepPropose. If the replica has already calculated a dependency set D_{r_i} for a checkpoint request, then it reuses that or otherwise calculates a new dependency set. The set is only recomputed if necessary, as this can only result in additional dependencies, which could slow down the execution. In case of a no-op request, the dependency set is chosen to include all known checkpoint requests. Before accepting a ViewChange message, a replica now has to wait until all dependencies included in the synthetic DepVerify are known. The NewView calculation falls back to a *default-request certificate (DRC)* consisting of $2f + 1$ valid synthetic DepVerifys, if neither a fast-path nor reconciliation-path certificate exists. This guarantees that

[3] A limited number of agreement slots technically also prevents infinite dependency chains. However, these could encompass the full agreement window, which can contain thousands of requests, and therefore could significantly delay the request execution.

checkpoint requests are committed in regular intervals and are also ordered in regard to no-op requests. The timeout to complete a view change increases to 5Δ to account for the delay until a replica knows about all dependencies.

4.5.2. Checkpoint Creation

Once a checkpoint request is executed, the replica creates a checkpoint by taking a snapshot of the application state and broadcasts a $\langle \textsc{Checkpoint}, cp.seq, barrier, h(cp)\rangle_{\sigma_{r_i}}$ message. *cp.seq* is a counter which is incremented for each new checkpoint. As a checkpoint request acts as a barrier, by dividing requests into before and after the checkpoint, they are totally ordered and consequently each replica will assign the same counter value to a certain checkpoint. *barrier* states which slots were executed before creating the checkpoint and is set to the checkpoint request's dependency set plus its own slot. And $h(cp)$ is the hash of the checkpoint. Once a replica has collected a quorum certificate of matching checkpoint messages, the checkpoint becomes stable and allows the replica to garbage collect all older slots included in *barrier*. As a substitute for the garbage-collected requests, the dependency calculation adds dependencies on all slots covered by *barrier*. This ensures that future requests are ordered after the garbage-collected ones.

Dependency Cycles and Checkpoint Requests

As a checkpoint request conflicts with all other concurrently proposed requests, it can be part of a dependency cycle. In this case, the checkpoint barrier is used to partition the corresponding strongly connected component into before and after the checkpoint. If multiple checkpoint requests are part of the same strongly connected component, then their barriers are merged and extended to include the checkpoint requests themselves. The merged barrier is also extended to cover all already executed slots and bounded to slots within the expansion limit. Both situations can arise when unblocking the execution, however, as the unblock case makes the same modifications on all replicas and slots after a checkpoint request have to wait for that request to be executed, this results in the same barrier modifications on all replicas.

Afterwards all requests before the barrier are executed, followed by creating the checkpoint. Finally, the execution is restarted for the remaining slots of the dependency cycle. This ensures that a replica continues the request execution in the same way as a replica does after applying the checkpoint.

4.6. Correctness

In the following we present a proof sketch showing that replicas agree on the request and its dependencies for a slot. Additionally, we sketch that Isos guarantees a consistent execution order for conflicting requests. Finally, we show that the replicas only require bounded state. Please refer to Appendix A for the full proof.

Agreement

All correct replicas that commit a certain slot, will commit the same request and dependency set. A request can only commit in a slot if a replica collects a quorum certificate of

matching DepCommit or Commit messages. As shown in Section 4.3.2, a correct replica only sends one of these messages in each view, such that at most one of these certificates can exist. The messages include a hash that binds them to a specific request and dependency set, therefore only this specific combination can commit.

To show that the property also holds across view changes, we show that *once a correct replica has committed a slot, all future view changes will select the committed value.* Once a slot has committed, at least $f + 1$ correct replicas have collected a corresponding fast or reconciliation path certificate. As the view change requires ViewChange messages from $2f + 1$ replicas, at least one of them will include the certificate. We now show that the correct certificate is selected during the view change. If the most recent commit was via the fast path, that is, the slot was *fp-committed*, then no RPC can exist, as both certificates are mutually exclusive. Otherwise, if the slot was *rp-committed*, then the RPC is selected and if multiple RPCs exist, then the latest one is selected. In each case, if the slot committed, the corresponding value will be selected in all future view changes.

Alternatively, a replica can query other replicas for their agreement result (cf. Section 4.3.4). As this requires $f + 1$ matching Execute messages, including one from a correct replica reporting the correct result, the learned result is also correct.

Execution Order

For the execution we show a property also used by EPaxos [162]: *Two conflicting requests A and B, will be executed in the same order on all correct replicas once committed.* We start by proving that two conflicting requests A and B are always connected by a dependency between them if no view change occurs. Taken together the DepPropose and DepVerify contain dependencies reported by a quorum of replicas. Once a correct replica has accepted the DepPropose for a slot, it will include the request as a dependency for all later conflicting requests. The quorums for the two requests overlap in at least one correct replica, therefore at least request A or B will collect a dependency on the other, or possibly both. As shown before, all correct replicas commit the same dependencies for a slot. Thus, at least one of the requests depends on the other.

Together with sorting slots according to their dependencies, this ensures that one request is consistently executed before the other. The special case to unblock the execution, which is discussed in Section 4.4.3, results in identical dependency graph modifications on all replicas, such that the execution order remains consistent.

We now extend the proof to also cover view changes. These will only either select a request which was *fp-verified*, *rp-prepared* or fall back to a default request instead. For a slot only a single DepPropose can ever collect the necessary $2f$ matching DepVerifys, this protocol phase is not repeated after a view change. Without them the slot can only commit the default request for which the replicas collect dependencies during the view change. As the default request does not require other slots to add new conflicts, it can be safely used as fallback value.

Thus, the view change either selects the request together with its dependency set as discussed in the previous paragraph or the default request. A reconciliation-path certificate contains $2f+1$ matching Prepares which implicitly also confirm the correctness of the included dependencies. A fast-path certificate consists of a DepPropose and

$2f$ matching DepVerifys from all replicas in the fast-path quorum. The DepVerifys prove the correctness of the request and that the fast-path quorum agreed with the initial dependency set. The verification of a fast-path certificate additionally has to verify that the DepVerifys do not include additional dependencies, such that the requirements for *fp-verified* are fulfilled. This prevents faulty replicas from tampering with the dependencies included in a fast-path certificate.

As fallback a default request certificate for the checkpoint request or a no-op is dynamically assembled during the view change. The replicas already know the default request for a slot, which allows them to calculate an appropriate dependency set. The certificate contains dependency sets from a quorum of replicas, such that the above correctness discussion applies. The default request afterwards still has to complete ordering via the reconciliation path to ensure that the replicas agree on the request and its dependencies.

A faulty request coordinator that proposes conflicting DepPropose cannot introduce broken dependencies. As the DepVerifys are bound to a specific DepPropose, the proposal can only commit if all replicas in the fast-path quorum received a matching DepPropose. Correct replicas only vote for one proposal for each slot, such that only one DepPropose can commit. Otherwise, it will be replaced by the default request during the view change, which does not introduce new conflicts. Thus, in the worst case, other slots have unnecessary dependencies if the request was replaced by a no-op.

Bounded State

The order window contains at least $2k$ slots from each replica, resulting in a total of $O(k \cdot N)$ slots. For each slot, a replica has to store the request r and the corresponding messages sent by each replica. In addition, the messages can contain a dependency set with N explicit entries. Their verification only requires tracking the latest dependency for each replica, that is, only up to N dependencies per message. As a slot is only accepted if its immediately preceding slot already is, this is sufficient to ensure that all earlier slots also were accepted. That is, the required state is in $O\left(k \cdot N \cdot (|r| + N \cdot \mathit{signature\_size} + N \cdot N)\right)$, where $|r|$ is the request size and $\mathit{signature\_size}$ is the size of the signature used to sign messages exchanged between replicas. The term $N \cdot N$ corresponds to the dependency sets from each replica.

As discussed in Section 4.3.2, the compact dependency encoding allows the application to implement the dependency set calculation such that for each state object per replica only the latest slot that read or wrote it has to be stored. Thus, the state required to calculate dependencies is in $O(N \cdot |\mathcal{O}|)$ where $\mathcal{O}$ is the set of all state objects.

The execution component only expands up to $k_x \cdot N$ slots at the same time. Tracking dependencies between slots can result in edges between every slot in the graph, that is, up to $(k_x \cdot N)^2$ edges. Implicit dependencies on future slots only have to be converted into edges once the corresponding slots are inside the execution window. The calculation of strongly connected components also requires space linear in the number of slots. Thus, the state required to manage the dependency graph is in $O((k_x \cdot N)^2)$. The requests themselves are not counted here, as the execution only stores requests that are still within the agreement window. Thus, Isos only requires a bounded state.

4.7. Optimizations

The performance of Isos can be further improved by allowing the replicas to take the fast path even if certain dependencies are only proposed by $f + 1$ replicas as described in Section 4.7.1. Additionally, Section 4.7.2 discusses how the request batches could be optimized to reduce the conflict rate. We also present how to reduce the number of signature computations in Section 4.7.3 and encode the view change messages more efficiently in Section 4.7.4. Finally, Section 4.7.5 discusses defenses against performance attacks from malicious replicas.

4.7.1. Taking the Fast Path More Often

In order to improve the protocol throughput, Isos includes an optimization to increase the amount of slots that can be ordered via the fast path. We modify the predicate *fp-verified* as follows. A slot is *fp-verified* if every dependency reported by the DepVerifys is included in at least $f + 1$ of them; all dependencies included in the DepPropose must be included in every DepVerify. This covers the previous definition, where the DepPropose and DepVerify had to contain identical dependencies, but also allows for example taking the fast path if $f + 1$ DepVerifys include the same additional dependency. As the DepCommit or Commit messages include a hash that covers all DepVerifys, this still guarantees that all replicas agree on the same dependency sets.

Faulty replicas cannot manipulate a fast-path certificate to add or remove new dependencies, even if they replace their DepVerifys with manipulated messages. If a faulty replica removes a dependency such that it is no longer included in at least $f + 1$ DepVerifys, then the certificate becomes invalid. As at least one of these DepVerifys is from a correct replica, which will not issue a second different DepVerify, it is not possible to remove the dependency without invalidating the certificate. Likewise, introducing a new dependency is not possible, as the faulty replicas can only generate f such DepVerify. Consequently, faulty replicas can only replace their DepVerifys if this does not change the value proven by the fast-path certificate.

A reconciliation-path certificate includes a quorum of Prepares. These contain the hashes of the DepVerifys such that these cannot be manipulated without detection.

With the optimized *fp-verified* predicate, we have to revisit the correctness proof for the view change. Assume that a slot was *rp-committed* with f correct replicas reporting a dependency to request A and a faulty replica reporting dependencies on requests A and B. The remaining replicas report an empty dependency set. Then the faulty replica can replace its DepVerify with one that just includes request A and thereby assemble an FPC with only dependency A. However, the result of the view change is still correct, as an RPC has a higher priority than an FPC and at least one correct replica will submit its RPC if the slot committed. In particular, the RPC proves that the slot *rp-prepared*, which makes it impossible for the slot to be *fp-committed* in the same view. Thus, the view change correctly ignores the faulty FPC.

4.7.2. Optimized Batch Cutting

The grouping of requests into batches can have a significant influence on the number of conflicts. If even a single request in a batch conflicts with some other slot, then the resulting dependency applies to the whole batch. This makes it more likely that a slot cannot be ordered on the fast path and instead has to take the reconciliation path.

Depending on the application scenario, certain requests may be more likely to conflict than others. Thus, it can be beneficial to group conflicting and non-conflicting requests into separate batches. For example, it may be possible to partition requests based on the accessed key, such that requests for keys that are only used by a single client, which therefore do not conflict, are separated from others. Similarly, it can be useful to group by frequently and less frequently accessed keys.

If multiple requests that wait for being ordered are conflicting with each other, then these should be added to a single request batch. Conflicts between requests within a batch are ignored for ordering, as the requests in a batch are already ordered in regard to each other.

Like with the standard batching optimization, the request coordinator should delay proposing a request only by a short amount of time. Otherwise, the lower conflict rate is overcompensated by the increased proposal delay.

4.7.3. Fewer Signatures

All messages exchanged between replicas rely on signatures for authentication. However, for messages which are not relayed between replicas, a message authentication code (MAC) would be sufficient, which also can be computed much faster than a signature [57]. Replicas can use MACs for the DepCommit, Commit, NewView, QueryExec and Execute messages. For these, either an individual MAC per recipient or a MAC authenticator [57] can be used.

This reduces the number of signature computations necessary by half for the fast path and by one third for the reconciliation path. Due to the use of batching, this reduction does not directly translate into a similar improvement in throughput, as the signature computation costs are already amortized over a whole request batch. However, especially for small batch sizes this can still improve the performance.

4.7.4. View-Change Efficiency

The view change, as described in Section 4.3.3, uses a NewView message that contains ViewChange messages from a quorum of replicas, which in turn each contain a certificate with messages from a quorum, resulting in a large message size. To reduce this size, we make use of an insight by Abspoel et al. [4], who have shown that the NewView message created by the view change in a PBFT-like protocol has to prove two separate statements. The NewView has to provide a *proof of recency*, showing that view $\tilde{v}$ is the latest view up to v' that might have committed, and a *prepare certificate* to prove that a certain value prepared in view $\tilde{v}$. By including only the message parts necessary for each statement, we can reduce the size of the ViewChange and NewView message.

The view change assembles a *proof of recency* stating that up to a view v', which is the view to enter after completing the view change, the latest possibly committed value is from view $\tilde{v}$. This proof consists of $2f + 1$ confirmations from different replicas r_j that the latest view up to v' in which the replica prepared a request was $\tilde{v}_j \leq \tilde{v}$. At least one of these confirmations must be for view $\tilde{v}$. As shown in Section 4.6, once a slot is committed, at least one correct replica which prepared it, will take part in the view change and thus contributes at least this view to the proof of recency.

In addition, the ViewChange of a replica r_j has to include a *prepare certificate* for view $\tilde{v}_j$, proving that a certain value was prepared in the reported view $\tilde{v}_j$. This requirement prevents faulty replicas from reporting too high views.

To integrate this optimization into Isos, we split the ViewChange message into two parts, one for the proof of recency and one for the prepare certificate. Now, replicas broadcast a $\langle\langle \text{ViewChange}, s_i, v'_{s_i}, \tilde{v}_{s_i}, cert\_type, h(cert)\rangle_{\sigma_{r_i}}, cert\rangle$ message, where the signed part only contains the hash of the prepare certificate $h(cert)$. The last prepared view $\tilde{v}_{s_i}$ and *cert_type* match the values from the certificate, but are now also directly included in the ViewChange message. If no certificate exists, their value is set to $\perp$. Together v'_{s_i}, $\tilde{v}_{s_i}$ and *cert_type* form a confirmation as required for the proof of recency.

The $\langle \text{NewView}, s_i, v'_{s_i}, VCS, cert\rangle_{\sigma_{co}}$ message only contains the $2f+1$ ViewChanges without the certificate, which together form the proof of recency, and the prepare certificate *cert* for the latest prepared view according to the proof of recency. Both *dp* and $\vec{dv}$ are available as part of *cert*. For the verification, replicas additionally have to check that the ViewChanges match the view v'_{s_i} in the NewView and that the *cert* matches the latest $\tilde{v}_{s_i}$ and *cert_type* according to VCS.

Furthermore, for the view synchronization mechanism, the replicas only require the signed ViewChange without the corresponding *cert*. Thus, when broadcasting the ViewChange, a replica sends *cert* only to the view-change coordinator.

4.7.5. Defense Against Performance Attacks

Isos, as described so far, is able to maintain liveness despite faulty replicas; however, these can still affect the performance by forcing the system to initiate view changes. In the following we discuss mechanisms to limit the impact of faulty replicas.

Tampering with the Fast-Path Quorum

Correct replicas switch to different permutations of their fast-path quorum if a view change was initiated for a slot, or a follower takes longer to create its DepVerify than expected. This ensures that eventually the fast-path quorum only contains correct replicas, which prevents faulty replicas from continuously interfering with slots proposed by correct request coordinators.

To prevent faulty replicas from delaying the agreement for a slot by deliberately selecting a non-working quorum, the other replicas have to monitor how many view changes are triggered for the slots of each request coordinator within a certain timeframe. For that, they differentiate whether a slot committed after the first view change for which the original request coordinator acts as new-view coordinator, or after multiple ones.

Committing after one view change is an indication, that the fast-path quorum likely contained a faulty replica. These are less problematic as a correct replica will eventually select a working fast-path quorum. However, if the quorum selection takes too long, the replica must nevertheless be suspected to be faulty.

All replicas periodically issue status-update requests, which are ordered like normal requests, and contain the above monitoring information. As each replica may make different observations, ordering the monitoring information first allows for deterministic computations. After creating a new checkpoint, each replica uses the ordered monitoring information to deterministically check for performance issues and decide whether to temporarily forbid up to f replicas from proposing new requests. As all replicas are in the same state when creating a checkpoint, the calculation guarantees that all replicas reach the same decision. For the suspected replicas, this prevents all proposals to sequence numbers after the current ordering window. Once a replica receives a proposal for slots inside the ordering window of suspected replicas, then it immediately triggers a view change to at least view 1 for this slot. This ensures that faulty replicas cannot use these slots to delay the agreement of other requests.

The selection of a working fast-path quorum can require many attempts. For a small number of faults f, it is feasible to try out all $\binom{3f}{2f}$ possible quorums and use a working one. However, the number of possible quorums quickly rises for an increasing number of replicas. Following the approach from Jehl [125], it is possible to try out at most $O(n^2)$ quorums by collecting suspicions between replicas and selecting a quorum in which only replicas are included which do not suspect each other. The quorum calculation itself is still NP-hard, but we expect it to be more efficient than iteratively trying out a much larger number of quorums.

Unnecessary Dependencies

A faulty replica, in the following also called sender, could also try to slow down the protocol by reporting unnecessary dependencies to requests, even though they do not conflict. To detect such misbehavior, the other replicas, in the following called verifier, can use the $\mathit{conflict}(a, b)$ predicate to verify that the reported conflict actually exists. An unnecessary dependency can only be used to suspect a sender as being faulty, but it is no definite proof of misbehavior, as correct senders can also create apparently unnecessary dependencies in the following two cases.

Firstly, if the request in the depended upon slot is replaced with a no-op during a view change, then multiple correct replicas might report unnecessary dependencies such that the check must be skipped in this case. As described in the previous paragraph, a replica which causes too many slots to be replaced by no-ops will eventually be considered faulty.

Secondly, a faulty replica could propose multiple contradictory DepPropose for one slot. Then different replicas reach different conclusions on the validity of a dependency. To eventually detect this case, a replica which accepts a DepPropose *dp*, but also receives $f + 1$ DepVerifys for different DepPropose messages, must broadcast the DepPropose *dp*. Thus, another replica will receive both conflicting DepPropose messages and then broadcasts them to all replicas. As DepPropose messages are signed, this proves that the corresponding request coordinator is faulty. If a replica receives less

than $2f$ matching DepVerifys, then it also broadcasts its DepPropose as described in Section 4.3.4. Thus, in either case, by sending contradictory DepProposes a faulty request coordinator provides the other replicas with a proof that it is faulty.

The garbage collection requires special handling, as all slots covered by the barrier of a checkpoint are added as dependencies once a checkpoint becomes stable. That is, dependencies to slots up to this checkpoint barrier must be considered as legitimate. To allow the verifier to check the correct usage of a checkpoint barrier, a dependency set must include the counter and hash of the checkpoint whose barrier was used during the dependency set calculation. The verifier then has to delay the dependency verification until that checkpoint is stable. If the checkpoint hash does not match, this provides a proof that the sender is faulty. Dependencies that were already garbage collected are skipped by a verifier. To prevent faulty replicas from bypassing the dependency verification by adding a too high checkpoint counter to their dependency sets, the verifier also performs the following check. If the dependency set is garbage collected due to a checkpoint with a lower counter than stated in the dependency set, this also proves that the sender included a manipulated checkpoint dependency.

To reduce the message size overhead, it is sufficient to include the first byte of the checkpoint hash. A faulty replica that adds a fake dependency on a not yet existing checkpoint has only a small chance of guessing correctly, that is, the replica can cause a limited performance impact in exchange for a very high risk of being detected as faulty.

Flooding of Conflicting Requests

Conflict verification cannot prevent a faulty replica collaborating with a faulty client from creating requests that conflict with as many other slots as possible. As this attack relies on executing legitimate but latency-wise expensive requests, its impact can only be limited by throttling the amount of expensive requests that each client can submit in a certain timespan. If a replica does not adhere to these limits, then other replicas delay the processing of new slots from that replica as well. It is sufficient for the replicas to have clocks with approximately synchronized clock speeds (cf. Section 2.1.1), as a non-uniform rate limit can only cause a certain amount of delay for the request processing.

4.8. Evaluation

In the following we evaluate the performance of Isos against two other protocols. We run experiments on Amazon EC2 to first analyze the achieved response times and throughput using a microbenchmark. For a more comprehensive picture we conduct further experiments using the Yahoo! Cloud Serving Benchmark (YCSB) [66].

4.8.1. Setup

In our evaluation we compare the performance for three different protocol types with each other. Firstly, **PBFT** represents the classical leader-based approach in which all requests are sent to the central leader replica. Secondly, we created a protocol **CSP** which like Byzantine Generalized Paxos (BGP) uses a centralized slow path, such that in case

	Oregon	Ireland	Mumbai	Sydney
Oregon	-			
Ireland	118.139	-		
Mumbai	222.515	119.872	-	
Sydney	137.970	254.782	138.653	-

Table 4.1: Average round-trip times in milliseconds between replicas in the used Amazon EC2 regions as measured on April 22, 2021.

of conflicts between requests, a central leader replica has to explicitly initiate the slow path. Unlike BGP, our variant does not have to exchange large sets of requests between replicas which can be prohibitively expensive. Thus, we expect CSP to offer better performance than BGP. And finally, **Isos** is our egalitarian protocol, which allows replicas to concurrently propose requests and is able to agree on an order between conflicting requests without falling back to a central leader replica.

Replicas

We configure the system to tolerate one fault $f = 1$ which requires four replicas $N = 4$. To conduct our measurements in a wide-area environment, we use Amazon EC2 and deploy the replicas in the regions Oregon, Ireland, Mumbai and Sydney. Each replica runs in a small virtual machine of type t3.small (2 vCPU, 2 GiB RAM) and uses Ubuntu 18.04.5 LTS with OpenJDK 11. The slow-path leader for CSP is placed in Oregon.

We set the estimated one-way communication delay between replicas to $\Delta = 200ms$, based on half the round-trip times in Table 4.1, which vary between 59 and 127 ms.

All protocols are implemented as part of the same codebase to allow for a meaningful comparison. From the optimizations described in Section 4.7, our implementation of Isos only uses the fast-path optimization from Section 4.7.1. Messages exchanged between the replicas as well as the client requests are signed using 1024-bit RSA PKCS1 signatures [163]. The reply sent to the client is authenticated using hash-based message authentication codes (HMACs) [133] with SHA256 [164]. The replicas use a checkpoint interval of $k = 2{,}000$ and in case of Isos an expansion limit of $k_x = 20$ for the execution. Each request coordinator creates request batches with up to 5 requests, smaller batches are only proposed if these cannot be filled within 5 ms.

Clients

The clients run in a separate virtual machine in each region and use the same setup as the replicas. Requests are issued in a closed loop, that is, a client submits a new request to the replica in its region once it receives a stable reply for the previous one.

The workload uses a key-value store as application to which clients send read and write requests that access small amounts of data stored for the given key. Write requests modifying the same key conflict with each other, whereas read requests for the same keys only conflict with write requests but not with each other.

Experiments

We run each measurement for 180 seconds of which a warm-up period of 30 seconds and a shutdown time of 10 seconds is cut of. The timing of each request is recorded individually and is used to derive the average and percentile values.

We want to answer the following questions with our evaluation:

1. How does the latency of Isos compare to that of the other systems?
2. Which influence does the conflict rate have?
3. Which throughput can Isos achieve for different request sizes?

4.8.2. Latency

In order to answer the first two questions, we start by analyzing the latency provided by the different protocols. We first measure the latency using a microbenchmark followed by additional measurements using YCSB.

4.8.2.1. Microbenchmark

In the microbenchmark, clients write data to a randomly selected key. Based on a conflict rate p, like in EPaxos [162] and ATLAS [90], a client with probability p selects a fixed key which can result in conflicts between the requests. All other requests are issued for a unique, client-specific key ensuring that these requests do not conflict. As highlighted by Tollman et al. [198], the conflict rate p controls the number of *possible* conflicts. Depending on the request timing the number of conflicts with an effect on the protocol execution may be lower.

Based on the targeted use cases, discussed in Section 4.2.2, we expect low conflict rates of 0%, 2% and 5% to be the most realistic [52, 162]. However, to present a full picture we also measure high conflict rates of 10% and 100%.

For PBFT, which is not influenced by the conflict rate, we instead measure the performance for each possible leader location. At low conflict rates, the fast path of CSP yields similar results as Isos, such that for clarity we only present measurements of CSP for a conflict rate of 5% and above.

In our first experiment, we run 10 clients in each region which issue write requests containing a 200 byte payload. Figure 4.12 shows the median and 90th percentile of the response time experienced by clients located in each region.

When using PBFT, the response times depend significantly on the location of the client and the leader in respect to each other. The leader is only able to provide the lowest response time for clients located in the same region as the leader. Clients in other regions first have to use wide-area communication to send their request to the leader which consequently increases the response time. For example, for a client in Ireland, the median response time increases from 264 ms when the leader is colocated in Ireland to 410 ms when the leader is located in Sydney, resulting in an up to 56% increase depending on the current location of the leader replica.

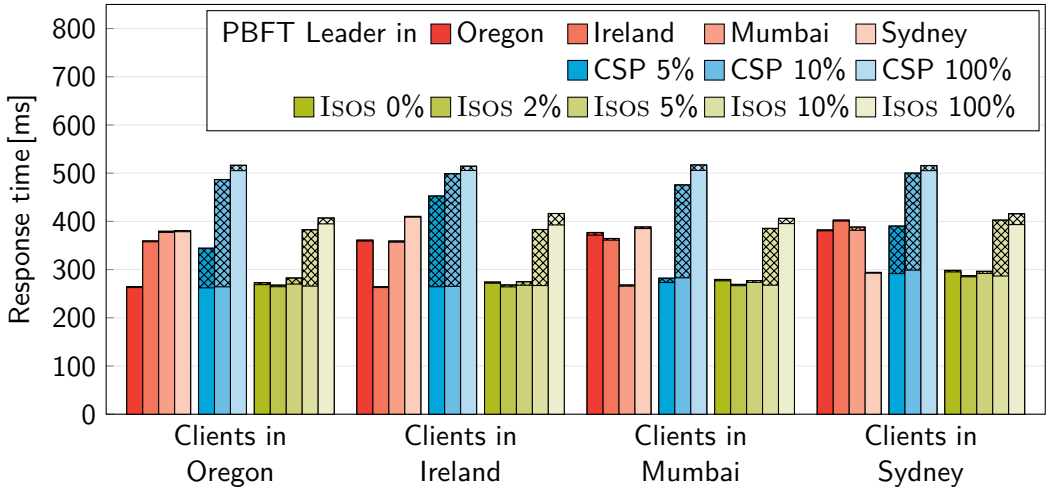


Figure 4.12: Median ☐ and 90th percentile ▩ of the response times for requests depending on the client and leader location as well as the conflict rate.

At low conflict rates of up to 2%, Isos provides response times similar to the response times of the best PBFT configuration in each region. Even at a higher conflict rate of 5%, the response times are only up to 7% higher than those of the best PBFT configurations. In contrast to PBFT, Isos is able to offer these low response times for clients in all regions at the same time. This is a result of the egalitarian protocol structure of Isos which allows every replica to directly initiate the ordering process for the request. Thereby, there is no central leader whose location influences the response times.

At high conflict rates, the response times of both CSP and Isos rise. For CSP, at a conflict rate of 100% the median and 90th percentile of the response time increase to up to 517 ms. For Isos, the response times only rise up to 416 ms. Even though Isos is designed for low conflict rates, it still offers median response times comparable to those of PBFT when the leader is located in an unfavorable region. Compared to CSP, the measurements show the benefit of Isos which does not rely on a central leader replica to initiate its reconciliation path and therefore avoids the additional communication step resulting in lower response times for Isos.

4.8.2.2. YCSB

We run additional experiments using the Yahoo! Cloud Serving Benchmark (YCSB) [66]. For the experiment we use a total of 200 clients, that are distributed equally across all regions, which issue a mix of read and write requests to the service. The service state consists of 1,000 entries containing 1 KiB of data each. The YCSB benchmark is configured to use its standard Zipfian distribution, which skews requests to focus on a few popular entries whereas most other entries are rarely accessed. The most popular entry is accessed with a probability of nearly 3.9% and the ten most popular entries with

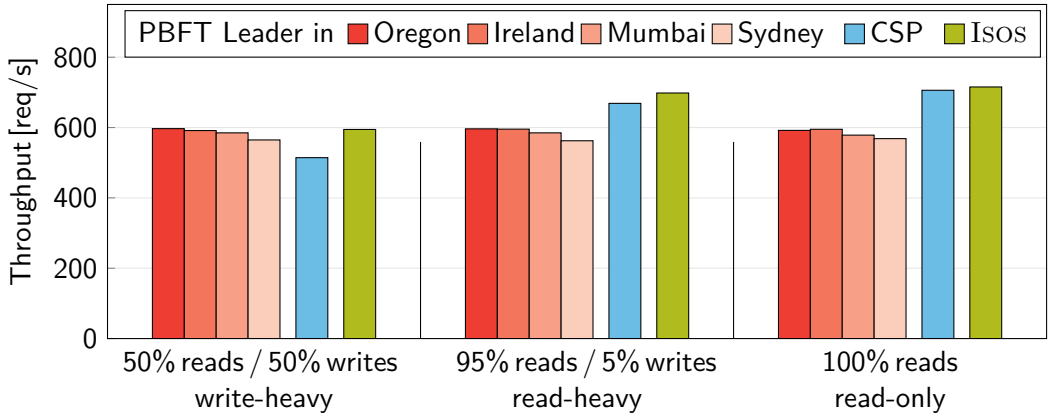


Figure 4.13: Average throughput for different read-write workloads using YCSB.

about 13.1%, thus resulting in a significant chance of conflicts. Clients in each region continuously issue requests for the full duration of the experiment.

We run the write-heavy (50% reads / 50% write), read-heavy (95% reads / 5% write) and read-only (100% reads) workloads. The measurements for each workload are shown in Figure 4.13.

For the write-heavy workload, Isos achieves a throughput of nearly 600 requests per second, which is similar to the best PBFT configuration. Due to the high fraction of write requests, a significant number of requests conflict with each other and thus limit the throughput for Isos. This is also visible in the throughput for CSP, which is 14% lower than that of Isos.

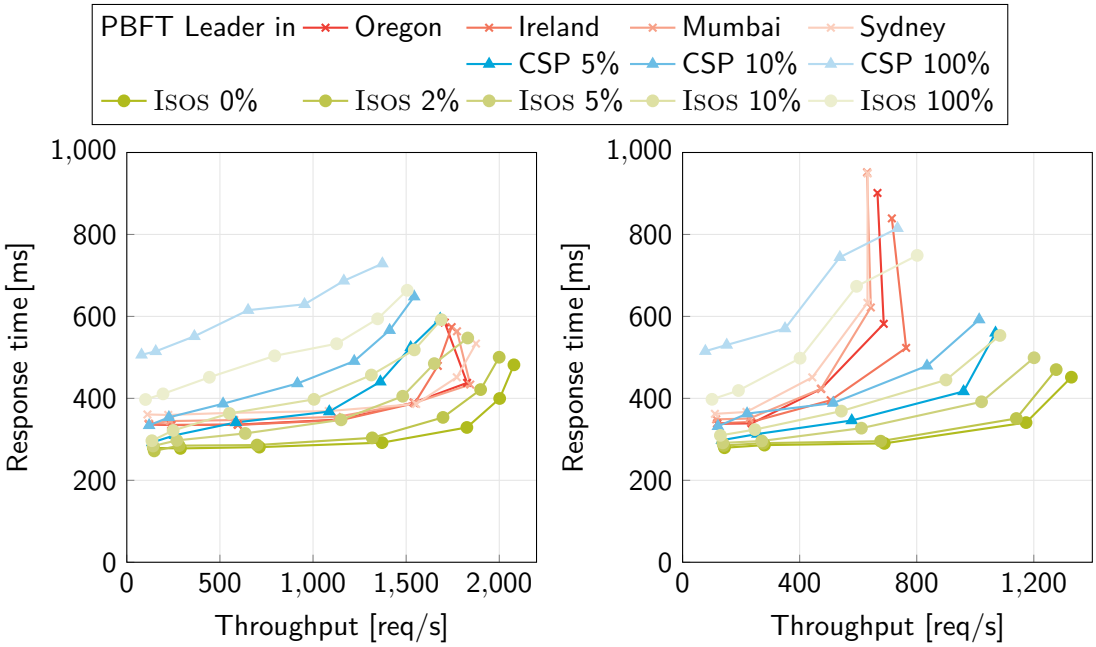
For the read-heavy and the read-only workload, Isos outperforms PBFT by 17% and 20%, respectively. As read requests do not conflict with each other, this results in a low conflict rate for these workloads, which allows Isos to take full advantage of its egalitarian system design.

4.8.3. Throughput

In our next experiment, we address the third question by measuring the throughput and response times using our microbenchmark from Section 4.8.2.1 at different request loads using up to 1,000 clients that are distributed equally across all regions. We first focus on 200 byte requests and switch to larger ones afterwards. The results are shown in Figure 4.14a and report the average over all requests issued during the experiment.

For PBFT using up to 400 clients the average response time remains stable below 369 ms and starts to rise afterwards. The throughput reaches a maximum of up to 1,875 requests per second. At this point the CPU of the leader replica saturates and prevents a further throughput increase.

At low conflict rates up to 2%, Isos is able to maintain an average response time below 304 ms for up to 400 clients. Afterwards the response time starts to increase. The maximum throughput reaches up to 2,079 requests per second, at which point all replicas



(a) 200 bytes

(b) 16,384 bytes

Figure 4.14: Average response time and throughput for increasing numbers of clients for two different request sizes.

saturate their CPU. In total, this translates to up to 18% lower response times or an up to 11% higher throughput compared to PBFT.

For a conflict rate of 5%, the performance of Isos is still roughly similar to that offered by PBFT. However, compared to lower conflict rates, the response times start to grow at a lower number of clients. We assume this to be a side effect caused by batching. By grouping multiple requests into a batch, the chance that a batch contains at least one conflicting request, increases to nearly 23% at a batch size of 5. A high load at the replicas also increases the chance that the ordering is currently processing another conflicting request which forces the replicas to enter the reconciliation path more often. We expect that the batch cutting optimization described in Section 4.7.2 could significantly reduce the conflict rate.

At even higher conflict rates, the latency of both CSP and Isos rise above that of PBFT. This is not a problem, as we target use cases with low conflict rates. The measurements also show that Isos consistently outperforms CSP. This is once again a result of the additional communication step necessary for the slow path used by CSP.

Large requests

We repeat the experiment with a large request payload size of 16,384 bytes and up to 600 clients for which the results are shown in Figure 4.14b.

The maximum throughput of PBFT is now limited to between 632 and 764 requests per second depending on the leader location. As the leader replica is responsible for

distributing each request to every replica, at this point its network connection becomes the bottleneck and prevents a further increase in throughput.

In contrast, Isos is able to process up to 1,328 requests per second, outperforming PBFT by up to 110%. Isos maintains its higher throughput up to a conflict rate of 10%. In addition, Isos provides lower response times for this workload than PBFT. The better performance is a result of enabling every replica to concurrently propose requests, which splits the work of distributing requests across all replicas.

4.9. Related Work

In the following we discuss related works regarding the overall agreement structure, optimizations for the fast path and the request execution.

Partitioning the Agreement Slots

A large body of work has been dedicated to agreement protocols that involve multiple leader replicas to spread the work of ordering requests. These are often built around assigning each leader replica its own subset of the sequence numbers, similar to the request coordinator-specific slots in Isos, to which the corresponding leader can then propose requests [22, 69, 84, 109, 161, 191, 192, 193, 212]. The main differences lie in the fault handling for these slots and how they are merged together. We start by discussing protocols which use a deterministic merge step [72], that is, one in which the replicas agree in advance on how to merge a certain range of slots into a total order.

Mir-BFT [192], which is based on PBFT, and the more generalized ISS [193], which can employ different agreement protocols, proceed in epochs consisting of a limited number of slots belonging to a selected subset of leader replicas. These slots are then merged in a round-robin manner. The leader set is updated after every epoch to exclude misbehaving replicas. As a related construction, RCC [109] proceeds in rounds in which every replica is responsible for a single slot. To handle failures, the replicas use an additional consensus protocol. These approaches allow multiple replicas to propose requests in parallel, however, as a slot can only be executed once all previous ones have been executed, this causes the leaders to depend on each other's progress. Thus, in the worst case, the response time is determined by the slowest leader replica involved. In contrast, in Isos slots only depend on each other if their requests conflict with each other.

BFT-Mencius [161] allows correct replicas to abort slots of other slow replicas to bound how far a replica can lag behind. Replicas can also voluntarily skip their slots if they learn that another replica has proposed requests for a later slot. This ensures that slots which could block the execution of other slots are filled and also allows a lagging replica to catch up by proposing no-ops, which require fewer resources for processing. Nevertheless, this results in unnecessary work when the request load is not balanced between replicas. In Isos it is not necessary to fill slots of a lagging replica with no-ops, as the other replicas only add dependencies to already proposed slots.

To better handle heterogeneous replicas, Omada [84] uses different limits for the size of batches proposed by different leaders. More powerful replicas can propose larger batches,

whereas weaker ones work with smaller batch sizes. With Isos each replica can propose new slots at its own speed, as dependencies are only added to slots that are in use.

These different mechanisms for coordinating the ordering progress of the replicas are a result of using a predetermined way to merge the slots from individual replicas into a single total order. In contrast, Isos dynamically determines the request ordering based on dependencies. Not yet used slots never show up as dependencies and thus have no effect on the request execution.

Dynamically Merging Agreement Slots

Besides merging slots using a predetermined order, it is possible to use the slots of each request coordinator to pre-order requests and only afterwards agree on how these slots are merged together into a single total order. That approach is used by PRIME [22], which offloads the request pre-ordering into three additional phases and lets the central leader replica only propose how to merge these streams of pre-ordered requests. For this the replicas exchange vectors containing the pre-ordering progress, which are then used to agree on which requests to merge in which order. This design avoids a bottleneck at the leader replica, but has the downside that the agreement itself now consists of six communication steps which significantly increases the latency. This represents a major latency increase compared to Isos which can order requests in three communication steps on the fast path.

Leaderless Agreement

In an egalitarian agreement protocol each replica is responsible for its own slots and coordinates the agreement process for them. If that replica fails, another one has to fill in to complete the agreement after a view change. In contrast, a leaderless agreement protocol does not rely on a specific replica to reach consensus. Instead, replicas exchange opinions and iteratively converge to a single one if necessary.

When using DBFT [69], the replicas each first pre-order a request and then agree on which of them should be used as consensus result. This last step can complete in a single communication step if all replicas have the same initial opinion. The minimum number of communication steps required are thus four steps. In case of different opinions, a rotating coordinator is used to nudge the replicas towards a common decision. BFT-Archipelago [27] is fully leaderless and instead proceeds in rounds consisting of three message exchanges to either commit if only a single current proposal exists or to iteratively converge on the maximum proposed value. Compared to Isos, these protocols are not designed to agree on multiple requests concurrently, instead they decide on a single out of multiple requests, which results in many proposals which are not used in the end.

The problem of unused proposals is addressed by Bullshark [191], which proceeds in rounds where replicas propose and certify blocks of requests. Each block references $n - f$ blocks of the previous round and each reference also counts as a vote for a block. These blocks then form a directed acyclic graph of certified blocks whose structure is used to commit them. If the block of the leader replica for a round receives sufficient votes, then the block and all preceding referenced blocks become committed. The constructed graph is very different from that in Isos as it does not reflect conflicts between requests,

but instead only serves to confirm older blocks. This also results in a higher latency as it requires assembling and broadcasting at least two rounds of blocks, which takes at least six communication steps.

Weaker Consistency

The eventually consistent PnyxDB [48] does not use a leader but instead only requires each request to collect a quorum of endorsements in order to commit. These endorsements represent a vote for the request that is valid as long as certain other conflicting requests did not commit. Non-conflicting requests receive unconditional endorsements and thereby can commit quickly. In case of conflicts, some involved requests may not collect enough endorsements to commit and are dropped after a timeout by running a checkpointing protocol. Similar to Isos, every replica in PnyxDB can propose new requests for ordering, however, unlike Isos it only guarantees eventual consistency.

Fast-Path Optimizations

The performance of an egalitarian agreement protocol [90, 162] is sensitive to how many requests can be ordered on the fast path, thus making it crucial for the protocol to take that path as often as possible. In an improvement over EPaxos [162], ATLAS [90] lets a request coordinator choose a specific fast-path quorum, which allows the fast path to be taken as long as the dependency sets reported by the followers in the fast-path quorum only diverge by a certain amount. More specifically, it is sufficient if each dependency is proposed by at least f followers, which allows ATLAS to always take the fast path for $f = 1$. Isos uses a similar optimization in which a dependency must be proposed by at least $f + 1$ followers, this higher bound is necessary to tolerate malicious replicas.

Another way to reduce the actual conflict rate is to use synchronized clocks to coordinate the request processing via timestamps. With Timestamp-Ordered Queuing [198] a request coordinator attaches a timestamp to proposed requests that states the time at which a replica should process the proposal. By synchronizing the time at which the replicas calculate their dependency sets, this increases the chance that those are equal, making it more likely that the request can be ordered on the fast path. In addition, the leader replica sends its initial proposal without a dependency set and delays the dependency computation until the same time as the other replicas. A similar optimization could be applicable to Isos, however, it would require additional safeguards to prevent faulty replicas from manipulating timestamps to introduce excessive delays.

Parallel Execution

For applications that require a non-trivial amount of computation to process a request, it can be beneficial to parallelize the execution. By making use of the dependency sets calculated by the agreement, PePaxos [60] is able to schedule independent strongly connected components to multiple cores. Requests within a strongly connected component have to be executed sequentially, whereas independent components can be executed in parallel. This approach is also applicable to Isos.

In SAREK [142] the agreement and execution are each split into multiple partitions. Each request is assigned to the partition responsible for the accessed data and is totally ordered and executed within that partition. If a request concerns multiple partitions,

then it is divided into sub-requests, whose execution must be coordinated between the involved partitions. The execution has to wait until all sub-requests are the next ones to execute in order to ensure a consistent result, and then executes the whole request once. Sub-requests from different multi-partition requests can block each other by waiting for the other request to execute first. When such a dependency cycle is detected, it is resolved by deterministically reordering the involved requests. This mechanism is related to unblocking the request execution in Isos for an expansion limit of $k_x = 1$.

4.10. Summary

Submitting a request to a central leader replica can significantly delay the agreement process if the client and the leader replica have to communicate via wide-area links. Instead, Isos allows clients to submit their requests to a nearby replica, which is immediately able to initiate the agreement process. The ordering between requests that are proposed by different replicas is established by agreeing on dependencies between the requests. This allows commutative requests to be ordered concurrently on the fast path. The execution then uses these dependencies to execute conflicting requests in a consistent order. To limit the necessary protocol state, replicas periodically propose checkpoint requests, which enable the creation of consistent snapshots by dividing requests into before and after the checkpoint.

Our evaluation shows that at low conflict rates, Isos is able to provide response times matching those of the best PBFT configuration for each client region. But in contrast to PBFT, Isos provides these response times for all regions at once. For large requests, Isos outperforms PBFT by up to 110%, as Isos is able to spread the task of distributing requests over all replicas instead of a single leader.

5
Cloud-Based Hierarchical Replication

Agreeing on the order of client requests requires several communication steps between the replicas. In the following we use a hierarchical system structure to reduce the latency for ordering client requests. By using replica groups that are each located in a different region, it becomes possible to differentiate between fast local-area communication within a region and the slower wide-area communication between groups. This allows our approach called Spider to trade additional, local communication steps for fewer wide-area communication and consequently a lower latency. Key to this is an abstraction called Inter-Regional Message Channel (IRMC) which allows for a modular implementation while also enabling the system to adapt to new client locations.

Section 5.1 introduces the problems with reducing the request processing latency. In Section 5.2 we give an overview of Spider, which uses a hierarchical system structure to reduce the latency. We modularize Spider by introducing multiple building blocks with well-defined interfaces in Section 5.3. Section 5.4 then combines these building blocks into a full system to process client requests either with strong consistency or optionally in case of read-only requests with weak consistency. Afterwards in Section 5.5 we discuss how Spider handles different kinds of client and replica misbehavior and how it ensures that these cannot cause correctness issues. We provide in Section 5.6 two different implementations for IRMCs along with further optimizations. Next, Section 5.7 describes optimizations to reduce the signature processing costs and to strengthen the consistency guarantees offered for read-only requests. Section 5.8 evaluates the request processing latency achieved by Spider in comparison to two other replication protocols. Related work is discussed in Section 5.9 and Section 5.10 concludes the chapter.

5.1. Problem Statement

Reducing the client-perceived request processing latency, as analyzed in Section 3.1, is either possible by designing a protocol to use fewer communication steps, by removing

the need to wait for some communication steps or by converting expensive wide-area communication to local-area communication. In this chapter we follow the third approach by using a hierarchical agreement protocol.

Section 5.1.1 motivates the use of a hierarchical system structure to trade wide-area communication for local-area communication and discusses the associated challenges. We analyze read request optimizations in Section 5.1.2 and how weaker consistency guarantees allow answering read requests without requiring wide-area communication. In Section 5.1.3, we present the problem that a hierarchical system consists of more parts than a plain agreement protocol and thus requires measures to limit its complexity. To allow a system to react to changes of the clients needs, it must be able to adapt which we will discuss in Section 5.1.4.

5.1.1. Reducing Client-Perceived Response Time

We want to minimize the time it takes to complete the agreement on a request order. Ideally processing a request only takes the time necessary for a single wide-area network round-trip to submit the request and receive the result from the service. For a replicated service this makes maintaining consistency a challenge, as the agreement protocol now has to work without adding (much) latency. To achieve this, the replicas have to be located at the same site to allow for local-area communication with low latency. However, running all replicas at the same site increases the risk of correlated failures.

Hierarchical replication protocols like Steward [24] group their replicas into sites and first reach a decision within the sites using a Byzantine fault-tolerant protocol followed by using a simpler protocol tolerating crash-faults to share that result across sites. The decision of the Byzantine fault-tolerant protocol is secured using a computationally expensive [24] threshold signature [182] that can only be created if a Byzantine majority quorum of replicas agree. Combined with the assumption that at most f replicas within each site can be malicious, a decision can be treated as originating from a replica group that is guaranteed to be correct. And thereby it becomes possible to run a crash-fault tolerant protocol between sites that only requires two communication steps. However, for clients not located at the leader site, the full protocol still can require up to three wide-area communication steps. In addition, this approach comes at the cost of a very high complexity, which we discuss in more detail in Section 5.1.3.

Approach of this Thesis

SPIDER simplifies the wide-area communication to reduce the communication steps to the minimum of two, by making use of the structure of modern cloud infrastructure. Platforms such as Amazon EC2 [19], Google Compute Engine [101] or Microsoft Azure [157] operate datacenters in multiple geographically distributed regions. Each region consists of multiple availability zones, designed to reduce the risk of correlated failures [19, 101, 159]. Their availability zones are typically located within 100 kilometers of each other [19], which reduces the risk that a problem affects multiple datacenters while still keeping the communication latency between them low. Latency is typically below one millisecond between servers within a single datacenter and below two milliseconds [157] between

datacenters in the same region, in contrast to wide-area communication, which can take dozens or even hundreds of milliseconds.

This architecture enables SPIDER to place its replicas running the agreement close to each other without compromising the availability of the service. That way, complex protocol parts exclusively run within regions and then use a simpler protocol to share the resulting decisions between regions.

5.1.2. Reading with Relaxed Consistency

Read requests offer an opportunity for optimizations, as they do not modify the application state and therefore do not have to be replicated to maintain a consistent application state. Protocols like PBFT [42, 57, 189] include a read optimization, which lets clients directly query a quorum of replicas for the result of a read request and thereby bypass the agreement protocol. After receiving matching replies from a quorum of replicas, the client can accept the reply. Only in case of too many differing replies, the client has to submit its request to the agreement protocol.

However, for a hierarchical protocol this optimization only provides a limited benefit compared to a write request, as a client now has to query at least one group that is likely located in a different region or even all groups as would be required for Steward to receive an up-to-date result[1]. This process requires the client to send a large number of requests and depending on the location of the replica sites might even result in a higher latency than a regular request. For example, the latter case can occur for clients located at the leader site, where a read request would have to wait for replies from all sites, whereas a write request only has to wait for a majority of sites.

If a client can work with a weaker consistency guarantee than strong consistency, then a hierarchical system like Steward is able to offer a much lower latency [24]. As each replica group maintains a copy of the application state, it is able to answer queries without communication with other groups. By collecting replies from a subset of at least $f + 1$ replicas within its group, a client is able to verify that the reply is guaranteed to be correct; however, due to the weaker consistency guarantees the reply may be (slightly) outdated. Only in case of diverging replies the client may have to retry the request by issuing it as a regular, strongly consistent request.

Approach of this Thesis

SPIDER uses groups consisting of $2f + 1$ replicas to execute client requests, which reduces the number of replicas that have to store the application state. These are also sufficient to process weakly consistent read requests. Compared to Steward which uses $3f + 1$ replicas at each site, the lower number of replicas per group presents new trade-offs regarding the consistency guarantees the system can provide.

[1] Steward [24] claims that it is sufficient to read from a majority of sites to receive a strongly consistent (linearizable) result. This is incorrect, as for a write request clients in Steward only have to wait for replies from the replicas at their local site. This forces the read optimization to query every single replica at each site, then merge the replies of each site and select the latest reply. Only then it is guaranteed that the quorums used for reading and writing overlap in regard to the replica sites, which is necessary to receive a strongly consistent result.

SPIDER also includes an optimization for strongly consistent reads that offers a similar latency as for regular write requests but reduces network overhead.

5.1.3. Reducing Complexity through Modularity

A downside of a hierarchical protocol is that it adds more components and thus increases complexity. Take Steward [24] as an example. It includes a Byzantine fault-tolerant protocol to reach and certify a decision within a replica group. That decision is then shared across groups using a crash fault-tolerant protocol. These two layers of protocols are interwoven with each other to properly handle faults and, for example, require a hierarchy of carefully balanced timeouts to work. The result is a protocol consisting of 15 pages of pseudocode in the appendix of the paper [24], even though parts like the client-side behavior are not even described.

Amir et al. also recognized this level of complexity as a problem and created a modular, hierarchical protocol where each site forms a logical state machine, which is then used to let each site represent a replica of a state-machine replication protocol [20, 21]. The communication between the logical state machines relies on an abstraction called BLink, which is able to reliably transmit an ordered stream of messages from one site to another. Together this allows building a modular protocol to which we refer as CustFT [20] in the following. As all messages that are sent across BLinks have to be ordered first, CustFT requires an additional mechanism dubbed CLink to transmit client requests. It either optimistically directly forwards clients requests to the leader site and thereby bypasses the BLink in the hope that the involved replicas are behaving correctly. Or as a fallback, it wraps and orders the request locally to allow the request transmission to the leader site using the BLink. This special-case handling partially offsets the performance costs of modularity but comes at the price of introducing additional wide-area protocols.

Even with that optimization the increased modularity results in a higher communication and cryptographic overhead than for Steward, thus providing worse performance and higher latencies. Steward consistently performs as well as or better than CustFT such that we only consider Steward for our evaluation.

Approach of this Thesis

SPIDER is structured as a modular, hierarchical protocol while providing similar or lower latency than Steward. The agreement protocol is used as a black box, whereas the connection to the groups executing the requests is decoupled using a channel abstraction that is able to safely transfer decisions between groups. That way, we can avoid the complexity of developing an agreement protocol from scratch and instead make use of an already existing and proven agreement protocol. The channel abstraction is also generic enough to transfer client requests without introducing further subprotocols.

5.1.4. Adapting the System Configuration

The clients that issue requests to the system can change over time. For example, clients at a new location can start using the system and require it to adapt to offer the best latency. Or new cloud regions become available that are better suited to serve existing

clients. That is, the system has to add a new replica group near the clients to avoid time-consuming wide-area communication. In the opposite direction, clients at an old location can shut down and now the no longer necessary location should also be shut down to save resources. Such changes to the system structure require mechanisms to transfer the application state between groups and also the means to reconfigure the system to add or remove certain replica groups. Existing hierarchical protocols like Steward [24] or CustFT [24] to the best of our knowledge, do not offer such mechanisms.

Approach of this Thesis

SPIDER includes a mechanism to adapt to changes in client locations by establishing new replica groups located in the same or a nearby datacenter or by shutting down old replica groups. This also includes the means for clients to query the system for its current composition including the location of each replica group.

5.2. Spider – Resilient Cloud-Based Replication with Low Latency

In this section we present a high-level overview of SPIDER, how it is structured to tackle the aforementioned problems and how it uses the structure of modern cloud infrastructures to offer low latency for processing client requests. Afterwards we present the consistency guarantees provided by weakly consistent read requests. We also discuss criteria to select a suitable configuration in which to deploy SPIDER in a cloud.

System Structure

SPIDER organizes its replicas into groups that are distributed across different regions all around the world. Complex protocol parts run within a region, whereas the interaction across regions relies on a simpler protocol. We expect that the replicas run on a cloud platform like those offered by Amazon EC2 [19], Google Compute Engine [101] or Microsoft Azure [159]. These platforms consist of datacenters in various *regions* worldwide that provide the infrastructure to run virtual machines that are then used to host the service's replicas. Each region usually consists of multiple so-called *availability zones*, which are constructed to be as fault independent as possible from each other [19]. That is, each availability zone consists of different datacenters with independent networking, power supplies, cooling and so on. The datacenters of different availability zones are also physically separated from each other to reduce the risk of disasters affecting multiple availability zones at once, but remain close enough to each other to allow communication between replicas in a region within two milliseconds [19, 157].

SPIDER leverages this structure to form multiple groups of replicas, where the replicas of each group are distributed across the availability zones of a single region. This lets the replicas in a group interact over short distance links, which offer latencies resembling local-area communication, but have a reduced risk of correlated failures. Thus, protocol steps in which only replicas of a single group interact with each other contribute little overall request processing latency.

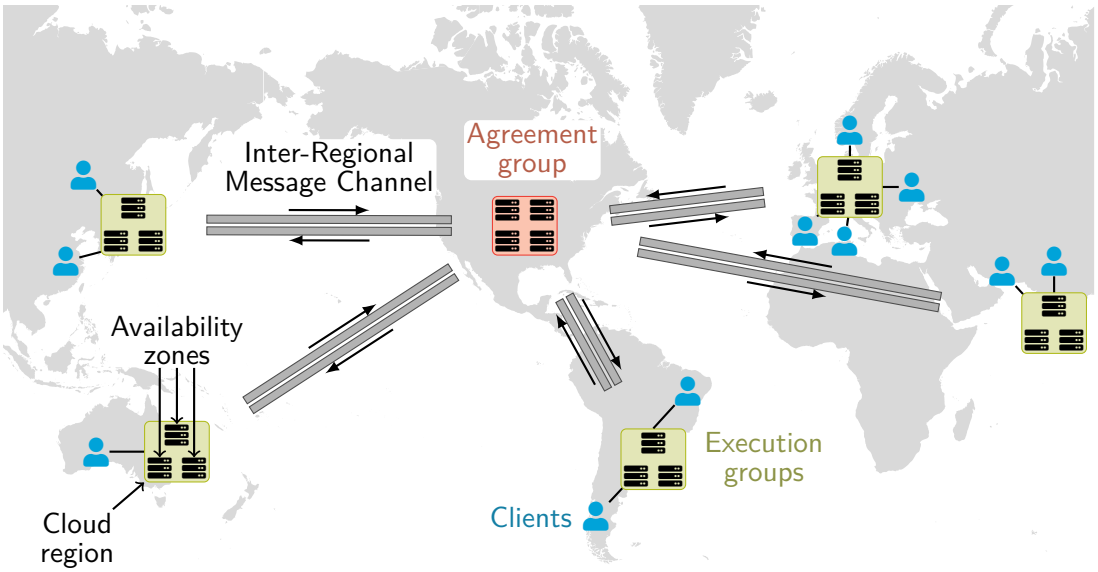


Figure 5.1: Architecture of SPIDER, which consists of a single agreement group ordering requests and multiple execution groups to process the client requests. The groups are coupled using Inter-Regional Message Channels.

As shown in Figure 5.1, SPIDER totally orders client requests using a single *agreement group*, which is hosted in a single region allowing the protocol to run with low latency. The agreement group is loosely coupled with multiple *execution groups*, which interact with the clients and execute the requests in the determined order. The groups communicate with each other using an abstraction called Inter-Regional Message Channel (IRMC), which forms a structure akin to a spider. To order requests, the agreement group uses an agreement protocol as black box, which allows the use of different agreement protocols depending on throughput or reliability requirements. Therefore, the number of replicas required for the agreement group is determined by the agreement protocol. When using PBFT [57], in order to tolerate f_a faults, the agreement group has to consist of $N_a = 3f_a + 1$ replicas.

In contrast, the execution groups each consist of only $N_e = 2f_e + 1$ replicas to tolerate f_e faults, which also reduces the number of implementations required to ensure fault independence. This lower number of replicas is sufficient as the consensus part is already handled by the agreement group such that the execution replicas only have to execute requests in the determined order and prove to the clients that their reply is correct [210].

Thus, a client only needs matching replies from $f_e + 1$ replicas to know that at least one of the replies is from a correct replica and therefore must be correct. As up to f_e replicas may exhibit Byzantine faults, f_e additional replicas are necessary to ensure that a client can always receive enough replies. With its loosely coupled execution groups, SPIDER is able to scale up or down by starting new execution groups near clients and stopping groups which are no longer necessary, thereby making use of the cloud's capabilities to provide resources on demand.

The number of faults f_a and f_e tolerated by the agreement and execution groups can be selected independently. For example, it would be possible to use $f_a = 2$ and $f_e = 1$ to let the central agreement group, which is critical for the system to make progress, tolerate more faults than the execution groups.

Execution Replica Registry

Before a client can submit a request, it first needs to discover a nearby execution group. For this purpose, the agreement replicas provide a read-only view of the current system configuration, which can be queried by clients. The configuration provides information about the execution groups, their location and the addresses of each replica. Whenever the system composition changes, after adding a new execution group, the replica registry state is updated to reflect these changes. A client located in the same region as an execution group selects the local execution group. In case the client is located elsewhere, it can base its selection on ping times to the execution groups. We explain this case in more detail in Section 5.4.1. This group selection should be considered a best practice; requests from a correct client that otherwise correctly follows the protocol will still be processed correctly, but they can result in a higher latency.

Request Execution

To interact with the service, a client sends the request to the replicas of its execution group. These will then verify the request and if valid, forward it using the message channel to the agreement group. The channel decouples the implementation of the groups, making the execution group independent of the used agreement protocol. The agreement group then hands the request over to the agreement protocol for ordering. Once the request is totally ordered, then the agreement replicas send the ordering decision back to the execution group, which executes the request and returns a reply to the client. In order to maintain a consistent state on all execution groups, each ordered request is also forwarded to all other execution groups. That is, all groups execute every totally ordered request and thereby maintain a consistent state.

This protocol structure runs complex agreement protocol steps only within a group and thus over intra-regional communication links. That way, the protocol execution benefits from the low communication latency within a region while at the same time avoiding the need to run a complex protocol across wide-area links. In total, the protocol requires two wide-area communication steps allowing SPIDER to process requests with low latency.

Read Requests with Weak Consistency

In addition to strongly consistent read and write requests, SPIDER also offers clients the possibility to issue read requests with weak consistency. These only require processing by the execution replicas at a single site, which improves performance as no wide-area communication is necessary.

To maintain safety, SPIDER always maintains strong consistency, or more formally linearizability, for write requests and for read requests only considers consistency models that relax the recency guarantees. That is, weakly consistent read requests return correct but possibly outdated values within certain bounds. SPIDER allows clients to choose

between the following two consistency models. For a more comprehensive explanation of consistency models, please refer to Viotti et al. [204].

Prefix Consistency Write requests must be executed in a total order, but reading can return a correct state that has existed at some arbitrary point in time. Combined with *eventual consistency* which ensures that replicas eventually converge on the current state, this ensures that clients eventually read up-to-date data.

Sequential Consistency Requests for each individual client are executed in the order they were issued and are integrated into a single total order. However, this does not require completed requests from one client to afterwards be immediately visible to another client as long as this does not conflict with the single total order. For example, it is possible for one client to finish writing some data and another client afterwards to read an older version of it. However, once a client has seen the effect of a request, then all later requests of that client continue to do so.

Reliability

For SPIDER to remain available, the agreement group must continue to work. When using PBFT, this requires that at most f_a out of $3f_a + 1$ replicas of the agreement group fail at the same time. As the replicas run in the cloud, a necessary precondition is that the underlying cloud systems running in the datacenters remain available and are not affected by correlated failures. In addition, the agreement group must be able to communicate with the execution groups.

Nowadays, cloud providers usually offer at least three availability zones for each region [19, 101, 159]. These availability zones are constructed to be as fault-independent as possible by having separate power supplies, cooling and also redundant network connections [19]. The network offers redundant connections between availability zones and also between regions. The datacenters of an availability zone are located up to 100 kilometers apart, but still remain close enough to each other to allow for low-latency communication between them [19]. That physical separation reduces the risk of correlated faults due to natural disasters like fires [176], flooding or thunderstorms. Besides the physical safeguards, each availability zone forms a separate update domain for which updates are scheduled such that they do not affect multiple availability zones at once [155]. Thus, if software updates by the cloud provider cause issues, these are likely limited to a single availability zone. This assurance is also reflected in the Service Level Agreements offered by the cloud providers, which promise a higher availability for services deployed across multiple availability zones [16, 98, 160]. Despite these efforts there are rare incidents which affect more than one availability zone at a time [15]. Nevertheless, the cloud providers try to learn from these incidents and improve the availability over time, for example, by engineering systems to limit the effect of outages to small parts of an availability zone [51].

Deployment Choices

Cloud providers usually offer at least three availability zones for each region [19, 101, 159], making them a suitable choice to deploy an execution group for $f_e = 1$.

For example when using PBFT, the agreement group requires four replicas to tolerate a single fault $f_a = 1$. A number of regions offer at least four availability zones (Amazon EC2: Northern Virginia (6), Oregon (4), Seoul (4), Tokyo (4) [19] and Google Compute Engine: Iowa (4) [101]), making them suitable to host the agreement group. With an ever-increasing number of regions provided by the cloud providers, an alternative can be to use a multi-cloud approach [5, 45], thereby spreading the replicas across the regions and availability zones of multiple providers. Well-connected regions are often served by multiple providers such that several regions like Frankfurt, Ireland and Paris host datacenters from multiple cloud providers [14, 101, 159], which in combination can also offer enough independent availability zones to host the agreement group.

To tolerate two faults the agreement group requires a total of seven replicas. As even the largest region of the previously mentioned cloud providers only offers up to six availability zones, one possible deployment would be to place up to two replicas in each zone. This would allow tolerating two independent Byzantine faults, however, the failure of a single availability zone would cause two replicas to become unavailable. Thus, despite the higher fault tolerance the system would still only be able to tolerate a single failed availability zone. To avoid this, the replicas can be split onto multiple regions that are located close enough to each other to still achieve communication roundtrip times of a few milliseconds. This allows the agreement replicas to communicate with each other, without causing a major increase in latency for the local protocol phases. In this configuration, the system can stay available with up to two failed availability zones, although the expectedly much rarer failure of a whole region would still cause the system to become unavailable.

Out of these possible configurations a region has to be selected for the agreement group. We expect that regions with a high number of availability zones are generally well-connected to other regions. Therefore, one of the main selection criteria is the latency to other regions hosting the execution replicas. Placing the agreement group at a well-selected location can help with providing similar maximum latencies for different execution groups. Another criteria to consider are the communication costs depending on the region hosting the agreement group. For Amazon EC2, the costs for communication between availability zones inside a region are fixed at one dollar cent per gigabyte [18]. However, for sending traffic between different regions the costs can vary between 2 and 15 dollar cent per gigabyte, which can have a major impact on the costs for running the system. Traffic from the cloud regions to other cloud providers or somewhere else on the internet is also generally more expensive than communication between or within regions [18]. Thus, it can be cheaper to place replicas in nearby regions of the same cloud provider than using availability zones from multiple cloud providers in the same region.

Failed or Outdated Execution Groups

Different from the agreement group, SPIDER is able to tolerate a configurable number of slow or not-reachable execution groups. Such groups will then fall behind while the rest of the system continues to make progress. Once a group recovers, it can retrieve missing requests from the agreement group in some cases or download a current checkpoint of the system state from another execution group.

In case an execution group is temporarily unable to contact the agreement group, like Steward [24] it can still reply to weakly consistent read requests. These replies may return stale state but allow reading data while the agreement group is unavailable. This is similar to systems such as ZooKeeper [122] where replicas answer read requests based on their local state, which has the benefit of a much faster processing as no wide-area communication is necessary.

If more than f_e execution replicas in an execution group are unavailable or when the group is unable to forward the client's request to the agreement group, then a client also has the option to temporarily fall back to a different execution group. The client then submits its request to the execution group to which it has the next lowest communication latency. Note that a client still has to rely on the assumption that at most f_e Byzantine faults occur within its execution group. In case that fault assumption is violated, then the faulty replicas can provide any reply to a client.

5.3. Building Blocks

Spider consists of several components, which are used to construct the overall system. In the following we present their interface and expected behavior. Section 5.3.1 describes the agreement protocol, which from the perspective of Spider is used as a black box. This makes it possible to pick a suitable protocol without affecting other parts of the system. For the communication between agreement and execution groups, we introduce our so-called Inter-Regional Message Channel (IRMC) abstraction, which is described in Section 5.3.2 and allows us to loosely couple these groups. These IRMCs provide an interface suitable for communication between groups, which is flexible enough to allow for different implementations as we discuss later on in Section 5.6. Section 5.3.3 discusses the application interface. Finally, the checkpoint transfer mechanism is presented in Section 5.3.4 and is responsible for distributing checkpoints of the system or application state between replicas of a region and if necessary also between regions.

5.3.1. Agreement Protocol Black Box

The task of the agreement protocol is to totally order requests by reaching consensus between the agreement replicas on the requests and their order. Spider requires that the protocol can tolerate up to f_a Byzantine faults and satisfies the following properties.

Safety The protocol must guarantee that the request delivered for a sequence number is the same at all correct replicas.

Liveness Requests submitted to all correct agreement replicas must be ordered eventually and delivered on correct replicas unless the requests are garbage collected.

Validity Only correctly authenticated client requests or no-op requests may be ordered.

We assume that the agreement protocol implementation uses the interface presented in Figure 5.2. There are several published agreement protocols that can be adapted to satisfy this structure [22, 47, 57, 63, 130].

```
1 interface Agreement {
2     // Blocks until the request is returned by the ordered callback on this replica
3     VOID order_request(CLIENTID c, CLIENTCTR t_c, REQUEST r)
4     // Return ordered requests one after another
5     callback ordered(SEQNR s, REQUEST r)
6     VOID collect_garbage_before(SEQNR s, CLIENTCTR[] ts)
7 }
```

Figure 5.2: Interface of the agreement protocol black box

Request Ordering

To order a request r from client c with client counter value t_c a replica passes it to the `order_request(`c`, `t_c`, `r`)` method. The client counter serves to detect duplicate and old client requests and must be increased for each new request. A request with a client counter value t_c less than or equal to that of the last ordered request for client c should be discarded. Once the ordering process has completed at a replica, then the black box delivers the request r with its associated sequence number s to the caller using the `ordered(`s`, `r`)` callback. For simplicity, we assume that the callback is triggered in increasing order of the assigned sequence numbers and that the first delivered sequence number is 1. An existing protocol can easily be adapted to that behavior. According to the validity property, the black box must also only deliver requests containing a valid client signature or no-op requests, which are skipped during execution.

Liveness

The liveness property requires that as soon as $N_a - f$ correct replicas have called `order_request()` for a request, then the request must eventually be delivered by at least $f_a + 1$ correct replicas. This is commonly implemented by starting a timer tied to a client counter value once a valid client request was received and by replacing a possibly faulty leader replica if the timer expires before a corresponding request was delivered.

Garbage Collection

The `collect_garbage_before(`s`, `ts`)` method controls the garbage collection of old requests. After it is called with a given sequence number s and an array containing the latest client counters ts for all requests ordered before sequence number s, then all slots with earlier sequence numbers can be garbage collected and may no longer be delivered by `ordered()`. The garbage collection can cause sequence numbers to be skipped, for example, when a replica applies a newer checkpoint.

The client counters are also used to stop timers for requests that were garbage collected. Once any correct replica has garbage collected a request, then the protocol is no longer required to guarantee the eventual delivery of the request.

Flow Control

SPIDER employs flow control to bound the state of a replica. To extend this to the agreement protocol, it must be possible to limit the number of requests queued for

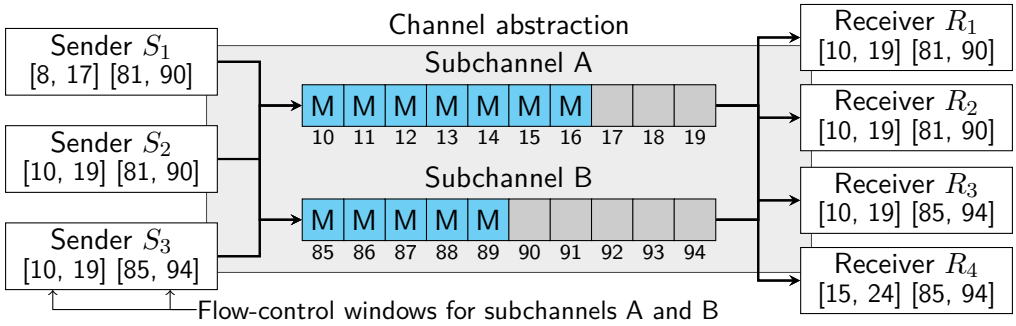


Figure 5.3: Logical representation of an IRMC configured to provide two first-in-first-out (FIFO) subchannels, which act as distributed queues connecting the sender endpoints S_* to the receiver endpoints R_*. Each subchannel has a fixed capacity of 10 messages M whose flow is regulated using per subchannel flow-control windows at the senders and receivers.

ordering and the number of requests waiting to be delivered must also be bounded. We expect the agreement protocol black box to conform to the following behavior.

To bound the number of queued requests, `order_request()` must not return before the request or a later one from the same client is ordered, however, concurrent calls of this method must be possible. This can be easily implemented by first submitting a request and then waiting for it to be delivered locally.

The `ordered()` callback is used to communicate back pressure to the agreement protocol. As long as the callback is blocked no further requests can be delivered and thus the protocol eventually has to pause ordering. Such a mechanism commonly exists in agreement protocols, for example, PBFT [57] implements this by using high and low watermarks to limit the number of currently active sequence numbers. Similarly, BFT-SMaRt [47] processes sequence numbers one after another and combines this with a bound on the number of requests waiting for execution.

5.3.2. Inter-Regional Message Channels

The Inter-Regional Message Channel (IRMC) abstraction is designed to handle the communication between different groups and thereby decouples them from each other. An IRMC enables a group of sender replicas S_* to safely transmit messages or decisions to a receiver group R_*. It is designed to handle up to f_s sender replicas and f_r receiver replicas that exhibit Byzantine faults. Each IRMC consists of multiple *subchannels*, which behave like message queues with numbered *message slots* that deliver a sequence of messages in order. Thus, the message transmission works in a first-in-first-out (FIFO) fashion. Each subchannel has a configurable, bounded capacity together with a per subchannel flow-control mechanism to prevent the senders from overwhelming the receivers. Figure 5.3 shows an example of an IRMC that connects a group of three senders to a group of four receivers and is configured to provide two subchannels. The participating replicas can use their sender or receiver endpoints to send and receive messages, respectively. The

```
1  // Sender endpoint
2  interface IRMC_Sender {
3      VOID send(SUBCHANNEL sc, POSITION p, MESSAGE m)
4      VOID move_window(SUBCHANNEL sc, POSITION p)
5  }

7  // Receiver endpoint
8  interface IRMC_Receiver {
9      MESSAGE receive(SUBCHANNEL sc, POSITION p)
10     VOID move_window(SUBCHANNEL sc, POSITION p)
11 }
```

Figure 5.4: Interface of the sender and receiver side endpoints of an Inter-Regional Message Channel (IRMC)

message flow for each subchannel is controlled independently using flow-control windows that are exchanged between sender and receiver endpoints.

BLinks [20] are a related type of Byzantine-fault tolerant message channels. However, compared to an IRMC a BLink is only able to transmit totally ordered requests, which either requires an additional ordering step beforehand that adds overhead or limits the applicable use cases of a BLink. In addition, a BLink does not include a flow-control mechanism that shapes the flow of messages between senders and receivers.

Authentication

An IRMC implementation must use authenticated messages for communication between replicas. If a replica receives a message that is not correctly authenticated, then the replica must discard the message without further processing. This is necessary to ensure that replicas know the real identity of a message sender and thus prevents faulty replicas from posing as another replica.

Endpoint Interface

To access an IRMC, all involved replicas instantiate a channel endpoint which serves as access point to the channel and handles the interaction with the other involved replicas. The channel endpoints of the senders and receivers together form the IRMC, with the endpoint encapsulating the channel implementation at each participant. Its interface is shown in Figure 5.4 and consists of four methods. To transfer a message m, the sender replicas have to `send()` it via their endpoints on a sender-selected subchannel sc. Afterwards the receiver replicas can use their endpoint to `receive()` the message m. And finally both the sender and receiver endpoints provide a `move_window()` method, which allows managing the flow-control window.

The central idea behind the fault tolerance of an IRMC is to always require support from at least $f + 1$ replicas and therefore at least one correct replica before forwarding information. Unlike a simple queue, transferring a message across an IRMC requires that at least $f_s + 1$ different sender replicas call `send(`sc`, `p`, `m`)` using the exact same

subchannel *sc*, position *p* and message *m*. Only then will a receiver replica be able to receive the message on subchannel *sc* and position *p* using `receive(`*sc*`, `*p*`)`. This ensures that at least one sender replica is correct and therefore vouches for the correctness of the message. Thus, (faulty) messages that were only sent by the up to f_s faulty sender replicas are not transmitted over the channel. Thereby, the IRMC allows a sender group to safely transmit decisions to a receiver group. In case correct sender replicas send different messages on the same subchannel position, then each receiver may receive none or any one of them. See Section 5.5 for a more detailed discussion of this behavior.

Flow Control

The limited subchannel capacity and the flow-control mechanism of an IRMC serve to prevent the senders from overwhelming the receivers. It works by letting the receivers decide how many messages the senders are allowed to transmit and thus limit the flow of messages. Once a sender has reached the flow-control limit, its `send()` operations will block. The mechanism manages each subchannel by maintaining an individual window that limits the range of positions for which messages may be transmitted. Each window is defined by its lower and upper bound; the lower bound controls the garbage collection of old requests, whereas the upper bound determines the flow-control limit.

A message slot at a position that is lower than the lower bound of the window is said to be *before* the window. Conversely, a message slot for a position higher than or equal to the upper bound of the window is said to be *after* the window. All other positions are *in* or *inside* the window. If there are no unused slots in a window, then it is *full*. In respect to a subchannel we will use *flow-control window* or just *window* interchangeably.

A sender can use `send()` to queue messages for transmission. The method returns immediately if there are remaining unused slots in the window. As a subchannel is modeled after a distributed queue, the sender has to call it for increasing positions in a subchannel. When trying to send a message at a position after the window, which means that the window is full, then the `send()` call will block and only return after the window has moved forward such that the position is now inside or before the window. From a flow-control perspective, the blocking `send()` call in case of a full window is the main ingredient, as it serves to propagate back pressure from the receivers to the senders and thereby limits the throughput to a rate that can be handled by the receivers.

Note that an implementation of the sender endpoint is only allowed to send messages to a receiver endpoint if the flow-control window reported by that individual endpoint includes the corresponding position. This allows each receiver to further restrict the influx of messages as necessary to maintain a bounded state. Messages that exceed an individual window are dropped by a receiver, as these are only sent by faulty replicas.

The `receive()` method exhibits a similar behavior. For positions inside the window, the call will return immediately if the corresponding message has already been received, or it waits until this is the case. A request for a position after the window blocks until the window moves to include the position and then the previous cases apply. Trying to `receive()` a request that is no longer inside the window will result in an exception informing the caller that the message for the request position is no longer available.

Garbage Collection

The subchannel flow-control windows also serve a second purpose: garbage collection. As just described, slots before the window cannot be retrieved and thus there is no need to retain them, which allows these slots to be garbage collected. The receiver endpoints set the lower bounds of their window for a subchannel by calling `move_window()` accordingly. This will immediately unblock all `receive()` calls that are now before the window. Once a sender endpoint learns about the updated window start, it will move its window start to the $f_r + 1$-highest position reported by the receiver group. Note that for a concrete IRMC implementation due to the necessary wide-area communication, it will take some time before the sender replicas learn of window movements. For example, in Figure 5.3 only sender S_3 has already received the latest window updates for subchannel B and updated its window to the $f_r + 1$-highest position. With the call to `move_window()`, a receiver replica guarantees that messages for earlier positions are no longer necessary or can be skipped by applying a newer checkpoint. Using the $f_r + 1$-highest position ensures that there is at least one correct receiver replica that has requested the garbage collection for that or a later position. Once a slot was garbage collected at a sender, this cannot be undone and therefore the window is only allowed to move forwards. Thus, an endpoint ignores calls to `move_window()` with a lower position than in a previous call.

Window Movement

The upper bound of the window, which serves as flow-control limit, is determined by the lower bound of the subchannel window plus the subchannel capacity. Using a fixed window size simplifies the implementation as it removes the necessity of maintaining separate lower and upper bounds for the subchannel window.

Endpoints on the sender side also offer a `move_window()` method, which allows senders to request moving the window start forward for cases when a receiver group has fallen behind or the sender group wants to skip sequence number gaps. Each receiver endpoint calculates the $f_s + 1$-highest window start position requested by the senders and if the position is after its window start, then the receiver endpoint calls `move_window()` internally. Here it is again guaranteed, that at least one correct sender replica has requested this or a later window start.

5.3.3. Application

An application must provide the interface shown in Figure 5.5, which is applicable for every application that can be represented as a deterministic state machine. The application must deterministically `execute(`m`)` a request m and always produce the same result when executed for the same application state. For read requests, `is_read_only(`m`)` must determine whether a request is actually read-only or not.

The application must also offer methods to create a `snapshot()` of the current application state and to `apply(`st`)` that snapshot of the state later on. This is necessary for garbage collection and to enable lagging replicas to catch up. To improve the efficiency of taking a snapshot of the application state, it is for example possible to use the fuzzy checkpointing approach as described in Chapter 6.

```
1  interface Application {
2      // Execute request and return a result
3      RESULT execute(REQUEST m)
4      // Verify that a request is readonly
5      BOOLEAN is_read_only(REQUEST m)
6      // Create application snapshot
7      APPSTATE snapshot()
8      // Apply application snapshot
9      VOID apply(APPSTATE st)
10 }
```

Figure 5.5: Interface of the application component

```
1  interface Checkpoint {
2      // Create a checkpoint
3      VOID generate(SEQNR s, STATE st)
4      // Sequence numbers for returned checkpoints must increase
5      // Checkpoints may be skipped
6      callback stable(SEQNR s, STATE st)
7      // Explicitly request the retrieval of a checkpoint
8      VOID fetch(SEQNR s)
9  }
```

Figure 5.6: Interface of the checkpoint transfer component

5.3.4. Checkpoint Transfer Component

Checkpointing [46, 57, 59, 62, 81] is used in SPIDER to periodically garbage collect old requests and slots in both agreement and execution groups to bound the size of the replica state. A checkpoint for a sequence number s contains the state st a replica has exactly after processing all messages up to and including sequence number s. If a replica has fallen behind, then it can use a checkpoint to update its state. For the checkpointing to be able to tolerate Byzantine faults, all replicas must reach the exact same state after processing messages up to a certain sequence number. In the following we describe the interface for a reusable component, that works for groups consisting of at least $2f + 1$ replicas and is employed by each group to agree on and transfer checkpoints between replicas. By default, each instance of the checkpoint transfer component only works locally within a group, that is, it only interacts with replicas from the same group.

The interface of the checkpoint transfer component is shown in Figure 5.6 and can be implemented as described in Section 2.3.3.5. Once a replica has captured its current state st, then it calls `generate(`s`, `st`)` with the sequence number s of the last processed request and the corresponding state st. The checkpoint component then distributes a signed checkpoint message containing only the hash of the checkpoint. A checkpoint

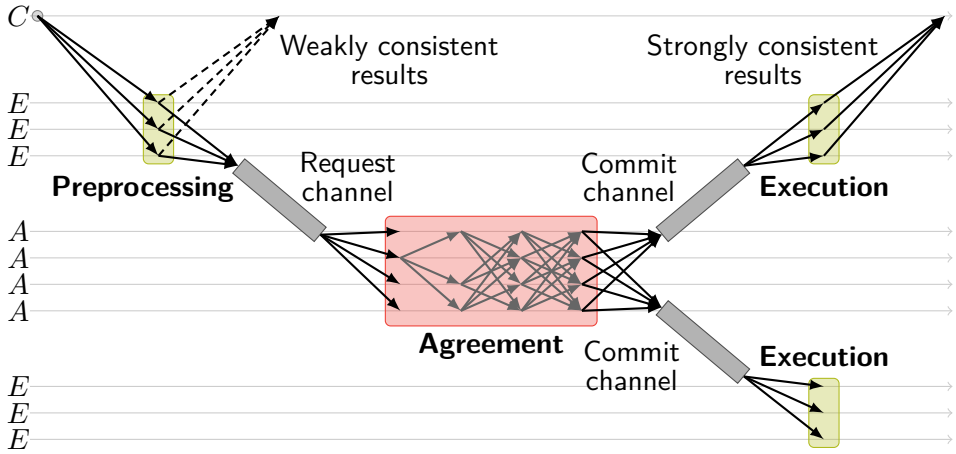


Figure 5.7: Overview of how SPIDER processes read and write requests. Read requests can choose between weak and strong consistency.

becomes *stable* at a replica, once it collects at least $f + 1$ matching checkpoint messages from different replicas, that is, either $f_a + 1$ in case of the agreement or $f_e + 1$ for the execution groups. These signed messages form a certificate that confirms the correctness of the checkpoint, as it includes a message from at least one correct replica that vouches for the correctness. As the checkpoint messages are signed, other replicas are later on able to verify the certificate and therefore also the correctness of the checkpoint.

Once the checkpoint transfer component has learned that a checkpoint is stable, it issues the `stable(`s`, `st`)` callback which informs the replica about the now stable checkpoint for sequence number s with state st. The callback is triggered whenever the checkpoint transfer component learns that a new checkpoint has become stable and the replica has either already generated a checkpoint for the corresponding sequence number itself or has received the corresponding checkpoint state from another replica. The `stable()` callback is only called for newer checkpoints, that is, the sequence numbers passed to the callback only increase over time. If a replica has learned about the existence of a newer checkpoint for a certain sequence number s via other means, then it can use the `fetch(`s`)` method to request a checkpoint to catch up. This method will also query other groups for the checkpoint. Once retrieved, the checkpoint is returned via the `stable(`s`, `st`)` callback.

The checkpoint component also has to ensure that all replicas of a group eventually learn about a checkpoint once it has become stable. This ensures that replicas that have fallen behind, eventually learn that this is the case and enables them to catch up.

5.4. Request Processing

SPIDER consists of an agreement group and multiple execution groups, which are connected using IRMCs. When using PBFT, the agreement group consists of $3f_a + 1$ replicas,

whereas the execution groups only require $2f_e + 1$ replicas each. The agreement group is connected to each execution group using two IRMCs as described below.

As shown in Figure 5.7, an execution group uses an IRMC instance called *request channel* to transmit client requests to the agreement group for ordering. For that, the execution group uses a client-specific subchannel to forward the requests of each client.

After ordering the requests at the agreement group, they are sent back to the execution groups using an IRMC instance called *commit channel*. Here a single subchannel is used to maintain the order decided by the agreement group. The replicas of an execution group execute the totally ordered request after receiving it via the commit channel. The execution group connected to the client also sends the execution result to the client.

The just described request processing flow is used to process write requests, that is, requests that modify the application state. These requests must be replicated to and executed at all execution groups to maintain a consistent application state. In contrast, read requests do not modify the application state and thus allow several optimizations. To speed up read requests that only require a weaker level of consistency, SPIDER offers a shortcut that allows an execution group to answer the request locally without requiring communication with the agreement group. A strongly consistent read request still passes through the agreement group, but after ordering it, the full request is only sent back to the execution group that will reply to the client. As a read request does not modify the application state, the other execution groups can safely skip the request.

5.4.1. Replica Registry

Before a client can issue a read or write request to SPIDER it has to determine which execution group to use. This is necessary as the execution groups can be reconfigured over time, which we describe in Section 5.4.7 in more detail. The agreement replicas provide a read-only registry that contains the contact information of all execution groups for use by the clients. More specifically, the system configuration $\mathcal{C}$ consists of a list that contains all execution groups used by the system, their replicas and all cryptographic keys necessary to set up communication with the replicas. It must be signed by a trusted admin client, which is responsible for making changes to the configuration of SPIDER. The system configuration $\mathcal{C}$ stored in the registry also contains a version number that increases every time the contact information changes.

Registry Querying

We assume that clients are configured with the addresses of the replicas in the agreement group and with the necessary cryptographic keys to exchange authenticated messages with them. A client then sends a $\langle \text{REGISTRYQUERY}, no \rangle_{\mu_{c,r_i}}$ request to all agreement replicas, which contains a nonce no that must be included in a reply to the client. The client authenticates the message individually for each replica r_i using a MAC. The agreement replicas only process correctly authenticated queries. The nonce must use a new value for each query and serves to prevent replay attacks where an attacker would provide an old reply to the client in response to later registry queries. The agreement replicas then

reply with a $\langle \text{RegistryReply}, no, \mathcal{C} \rangle_{\mu_{r_i,c}}$ message, which contains the client's nonce no and the latest configuration $\mathcal{C}$ known to the replica.

A client waits until it has received $2f_a + 1$ correctly authenticated replies from different replicas that contain the expected nonce and a valid configuration $\mathcal{C}$ signed by the admin client. If the client does not receive enough valid replies within a timeout, then it resends its query messages.

The client uses the version number included in the configuration to determine the latest version. As the configuration is signed, faulty replicas cannot create fake configurations with too new versions, but instead can only report the current or outdated registry information. System configuration changes pass through the agreement like regular requests and are thus kept consistent between agreement replicas, see Section 5.4.7 for more details. Once a configuration change is agreed upon, then at least one of the replies is from a correct agreement replica that has applied the current configuration. During a configuration update, a faulty replica could return the new configuration before it has been fully applied. This is not much of a problem, as the new configuration will become active eventually.

Execution Group Selection

The system configuration contains the location of each group and a list of addresses and public keys sufficient to set up communication with the execution group. Using this information, the clients and execution group replicas exchange cryptographic keys to authenticate messages; for example, by using a mechanism described by Castro et al. [58]. Alternatively a membership service can be used to set up the communication [26].

A client picks the execution group in the same region if one is available or falls back to selecting a nearby one based on ping times. That is, the client measures the ping times to the different execution groups and selects the one offering the lowest communication latency. This allows a client to optimize for weakly consistent read requests which only require communication with the execution group.

5.4.2. Write Requests

We now describe the necessary steps to process write requests. Messages exchanged between the client and execution replicas must be authenticated using a MAC and in case of a Write request, it must also contain a signature by the client wrapped with a MAC. The signature allows the agreement replicas to verify the correctness of the client request. As this is an expensive operation, the additional MAC allows the execution replicas to quickly verify that a request was indeed sent by a client c [63]. If the verification fails, then a replica can permanently ignore the faulty client. In general, messages that are formatted incorrectly or that are not correctly authenticated are dropped immediately by a replica without further processing.

In the following, line numbers refer to the pseudocode for the request processing shown in Figures 5.8 and 5.9. Pseudocode for the client is available in Appendix B.2.

```
1  s_n := 0                                          // Sequence number for last executed request
2  t[c] := 0                                         // Counter of latest forwarded client request
3  u[c] := ∅                                         // Reply cache ⟨Reply, u_c, t_c⟩
4  app = application, cp = checkpoint transfer component
5  E := execution group with |E| = 2f_e + 1
6  // Subchannel 0 is used as commit channel, any other subchannel could also be used
7  r_E = request IRMC sender                         // Each subchannel has a capacity of 2
8  c_E = commit IRMC receiver                        // Commit subchannel capacity must be ≥ k_e
9  on receive(m = ⟨Write, w, c, t_c⟩ from c):
10     if !valid_mac_{c,E}(m): return                // Ignore invalid requests
11     if t_c ≤ t[c]:
12         if u[c] = ⟨Reply, *, t'_c⟩ ∧ t'_c = t_c: // Check if a reply is available for the request
13             send mac_{r_e,c}(u[c]) to c
14         return                                    // Silently return on retry with no result yet
15     if !valid_sig_c(unwrap_mac(m)): return
16     // Each execution replica must forward a request once, even already executed ones
17     t[c] := t_c
18     r_E.move_window(c, t_c)                       // Notify agreement of new request
19     r_E.send(c, t_c, ⟨Request, unwrap_mac(m), E⟩)

21 main loop:
22     while true:
23         m := c_E.receive(0, s_n + 1)
24         if m = ⟨TooOld, s'⟩:
25             // Executor missed committed requests → fetch checkpoint
26             cp.fetch(s')                          // Ask other groups if necessary
27         else: // m = ⟨Execute, ⟨Request, ⟨Write, w, c, t_c⟩, E'⟩, s_n + 1⟩
28             s_n := s_n + 1
29             // Only execute new requests
30             if (u[c] = ⟨Reply, *, t'_c⟩ ∧ t_c > t'_c) ∨ u[c] = ∅:
31                 u_c := app.execute(m)
32                 u[c] := ⟨Reply, u_c, t_c⟩          // Store reply
33                 if E = E': // Only the local execution group sends the reply to the client
34                     send mac_{r_e,c}(u[c]) to c
35             if s_n ≡ 0 mod k_e:                   // Periodically create a checkpoint
36                 cp.generate(s_n, (u, app.snapshot()))

38 on cp.stable(s, st = (u', app')):
39     c_E.move_window(0, s + 1)                     // Allow garbage collection of commit channel
40     if s ≥ s_n:
41         s_n := s; app.apply(app'); u := u'
```

Figure 5.8: Pseudocode for an execution replica r_e. This is a variant of [86].

```
42 s_n := 0                                                    // Last ordered sequence number
43 win := [1,AG-WIN]                          // Range with [lower, upper] bound, both inclusive
44 AG-WIN ≥ k_a                                                // Size of agreement window
45 t[c] := 0                                  // Counter values of latest ordered request per client
46 t⁺[c] := 0                                                  // Counter values for next expected request
47 n_e := number of execution groups; z := limit on slow execution groups
48 hist := last |c_{E,0}| EXECUTEs
49 ag = agreement protocol black box, cp = checkpoint transfer component
50 A := agreement group with |A| = 3f_a + 1
51 for each execution group E:
52     r_E = request IRMC receiver                          // Each subchannel has a capacity of 2
53     c_E = commit IRMC sender                             // Commit subchannel capacity must be ≥ k_e
54 parallel for each client c and execution group E:
55     while true:
56         m := r_E.receive(c, t⁺[c])
57         if m = ⟨TOOOLD, t_c⟩: t⁺[c] := t_c
58         else: // m = ⟨REQUEST, ⟨WRITE, w, c, t_c⟩, E⟩
59             ag.order_request(c, t_c, m)                  // Returns once request is ordered
60             t⁺[c] := t_c + 1

62 // Delivered in-order, agreement must timeout if blocked for too long
63 on ag.ordered(s, r = ⟨REQUEST, ⟨WRITE, w, c, t_c⟩, E⟩):
64     sleep until s ≤ max(win)          // Force agreement to periodically create a checkpoint
65     // Update state with new request
66     t[c] := t_c; t⁺[c] := max(t_c + 1, t⁺[c]); hist.add(⟨EXECUTE, r, s⟩)
67     s_n := s
68     parallel for each execution group E:
69         c_E.send(0, s, ⟨EXECUTE, r, s⟩)
70     sleep until completed for n_e − z groups     // Send calls continue in the background
71     if s_n ≡ 0 mod k_a: cp.generate(s_n, (t, hist))   // Create checkpoint periodically

73 on cp.stable(s, st = (t', hist')):
74     parallel for each execution group E:
75         c_E.move_window(0, s − |hist'| + 1)              // Move commit window forward
76     ag.collect_garbage_before(s + 1, t')
77     if s > s_n:
78         tmp := s_n; s_n := s; t := t'; hist := hist'
79         parallel for each execution group E:
80             for x = ⟨EXECUTE, r, s'⟩ ∈ hist, s' ∈ [tmp + 1, s]:
81                 c_E.send(0, s', x)     // Add missing requests from hist to commit channel
82         sleep until completed for n_e − z groups
83     win := [s+1, s+AG-WIN]
```

Figure 5.9: Pseudocode for an agreement replica r_a. This is a variant of [86].

Request Forwarding

A client c wanting to modify the application state sends a signed $\langle\langle \textsc{Write}, w, c, t_c\rangle_{\sigma_c}\rangle_{\mu_{c,r_i}}$ request to all replicas of its execution group $\mathcal{E}$. The request contains the actual command w to execute and a client-specific counter t_c. The counter must be incremented by one for each new request. After receiving the WRITE request (Line 9) an execution replica uses the counter t_c to check whether a request is new and should be forwarded to the agreement group via the request channel (Line 15 - 19) or whether it is a retransmission of an already received request (Line 11 - 14). For a retransmitted request that was already executed, the replica sends the cached reply to the client (Line 13).

The check for a new request is only based on whether the replica has already forwarded it previously or not (Line 11 and 17). Even if a request was already executed by the execution replica, it will still be forwarded once. This guarantees that enough execution replicas forward a request to the agreement group to ensure liveness.

Before forwarding the request, an execution replica adjusts the flow-control window of the request channel to begin with position t_c (Line 18). For a correct client this is usually a no-op as the execution replica has already forwarded all previous requests. However, explicitly moving the window allows execution replicas that have missed a request from a correct client to skip the missing request(s) and catch up with the latest request. This scenario also applies when a client switches execution groups.

Then the execution replica wraps the WRITE request m in a $\langle \textsc{Request}, m, \mathcal{E}\rangle$ message and sends it to the agreement group using the client's subchannel c at position t_c (Line 19). If a client correctly distributes its request to all replicas within its execution group, then at least $f_e + 1$ correct execution replicas will forward the same request at the same position along the client-specific subchannel c. This ensures that the request is successfully transmitted to the agreement group.

If a faulty client distributes different requests for the same counter values or uses different counter values for one request, then the request will become stuck and not arrive at the agreement group. Note that this only affects the faulty client's subchannel, but not those of other clients. That is, a faulty client can only prevent that its own requests are processed. We will discuss this attack in more detail in Section 5.5.

Request Ordering

Replicas in the agreement group wait for requests forwarded by the execution group (Line 56). Once an agreement replica receives a request, it passes the request to the agreement protocol (Line 59). After the request is ordered, the replica updates the client request counter (Line 60) and waits for the next client request. In case a client skips a counter value, the channel will eventually return an exception $\langle \textsc{TooOld}, s\rangle$ to inform the replica about the next position s in the channel that can be received (Line 57). Afterwards the replica updates the counter and waits for the next request.

Once a request has been ordered (Line 63), it is wrapped into an $\langle \textsc{Execute}, r, s\rangle$ message, containing the REQUEST r and the assigned sequence number s. The replica then updates the counters for the latest ordered request and the next request position (Line 66). Afterwards the agreement replicas forward the request to all execution groups via the commit channel at position s (Line 69) to update their state.

Request Execution

Once at least $f_a + 1$ correct replicas of the agreement group have finished ordering and forwarded the request, this allows the execution replicas to receive the EXECUTE message (Line 23 and 27). The request's counter value is used to detect and ignore old or duplicate requests from a client (Line 30). Otherwise, an execution replica executes the request and stores a $\langle \text{REPLY}, u_c, t_c \rangle$ message containing the result u_c and counter value t_c (Line 31 and 32) to allow request retransmissions to retrieve the result. If the replica belongs to the execution group $\mathcal{E}$ contacted by the client, then it also sends the authenticated reply to the client (Line 34). The client then accepts the result after receiving $f_e + 1$ replies from different replicas of its execution group with matching result u_c and the expected counter value t_c. This ensures that at least one of the replies is from a correct replica and therefore must be correct. If the client does not collect a valid result within a timeout then it retransmits its request to its execution group. We defer the explanation of checkpointing to Sections 5.4.5 and 5.4.6.

5.4.3. Read Requests

For read requests, SPIDER distinguishes between reads with strong or weak consistency. For strong consistency the read request is ordered by the agreement group similar to a write request, whereas for a read request with weak consistency the request is only processed by the local execution group and thus does not require wide-area communication.

Strongly Consistent Read Requests

For a strongly consistent read request, a client c sends a signed $\langle\langle \text{READ}, w, c, t_c \rangle_{\sigma_c} \rangle_{\mu_{c,r_i}}$ request to its execution group with command w and counter value t_c. The READ is then processed similarly to a WRITE request. The main difference is that only the EXECUTE message sent to the client's execution group includes the full request. All other execution groups only receive a placeholder $\langle \text{READ}, \perp, c, t_c \rangle$ containing just the client c and its counter value t_c. As read requests do not modify the application state, they can be safely skipped, which allows this optimization to reduce the processing and data transfer overhead. The decision which group receives the full request and which only a placeholder request, is made deterministically based on the execution group $\mathcal{E}$ that is included in the REQUEST message delivered by the agreement protocol. The placeholder request is necessary as it allows the execution groups to update the reply cache entry for the client and store a placeholder entry. That way, every execution group still learns the latest executed client counter value.

Weakly Consistent Read Requests

For now we only describe how to guarantee prefix consistency for weakly consistent read requests and defer sequential consistency to Section 5.7.4. In case a client only requires correct but potentially outdated replies to its read requests, then it can send a MAC-authenticated $\langle \text{READWEAK}, w, c, t_c \rangle_{\mu_{c,r_i}}$ message to all replicas in its execution group. The request is only authenticated, as it is not forwarded between replicas. The replicas will then execute the request immediately and send a reply to the client. Once a client receives $f_e + 1$ identical replies, then it accepts the result as at least one reply must

be from a correct replica. This allows a client to receive a reply with low latency as only local communication is required. Note that the reply might be based on an outdated state in case a correct execution replica has fallen behind. In this situation faulty replicas could provide replies supporting the outdated result. However, a weak read request will receive an up-to-date result once all correct execution replicas have updated their state.

If a client does not receive enough matching replies within a timeout, then it can repeat its weakly consistent read request or fall back to a strongly consistent read request, which is guaranteed to eventually succeed. Similar to read optimizations in PBFT [57] or WHEAT [189], the former case may arise because the read processing is not coordinated across replicas and thus the replies can be based on different application states.

5.4.4. Group Coordination

The flow of requests between execution and agreement group requires coordination to prevent them from overwhelming each other. SPIDER uses the flow-control mechanism offered by the IRMC to control the message flow between two individual groups and thereby enables the agreement group to coordinate the global message flow by adjusting the IRMCs' flow-control windows as appropriate.

An execution group forwards client requests by sending them on their client's subchannel of the request channel. For that each execution replica first requests a move of the subchannel window such that it begins with the position for the new request (Line 18). As the window move has to be confirmed by the agreement group this throttles how many new requests a client can forward. Each subchannel in the request channel has a capacity of two to allow the execution replicas to forward a new request for its client, while the window update is still pending and therefore avoids stalls during normal execution. On the side of the agreement group, for each client subchannel a replica proposes one request at a time for ordering (Line 59), which limits the influx of new requests at the agreement replicas. If a (faulty) client sends new requests before previous requests were ordered, then some older requests may be dropped.

For the communication via the commit channel, the agreement group has to wait for enough execution groups to accept the EXECUTE messages. The agreement group waits until it has sent these messages to at least $n_e - z$ execution groups (Line 69) where n_e is the total number of execution groups and z is the number of execution groups that are allowed to fall behind. z must be within $0 \leq z < n_e$. This prevents up to z slow execution groups from slowing down the whole system. The `send()` calls to slow execution groups continue in the background, until the execution group either catches up far enough to allow the calls to complete or the corresponding positions in the subchannel window are garbage collected. To catch up with other groups, a slow group can request an execution checkpoint as described in Section 5.4.6.

5.4.5. Agreement Checkpointing

SPIDER uses checkpointing to allow fallen-behind replicas to catch up quickly and also to bound the size of the replica state. For this, each replica creates a checkpoint after

processing the request associated with a certain sequence number. The state also includes all requests which on the sender side of an IRMC are available inside the window of each subchannel. This allows a different replica to reconstruct the internal state of its IRMC.

An agreement replica creates a checkpoint after processing every k_ath sequence number (Line 71). The checkpoint consists of *hist*, containing the EXECUTEs that could still be inside the commit channel window, and the vector t, which includes the latest executed client counter values. The size of *hist* matches the commit channel capacity, which ensures that all agreement replicas will create identical checkpoints.

Once the checkpoint component has collected f_a+1 identical checkpoints, the checkpoint becomes stable (Line 73). This allows the agreement group to adjust the commit channel window to be at least as recent as the oldest message contained in *hist* (Line 75). Note that this only has an effect on lagging execution groups or replicas. When trying to receive an older message from the commit channel, these will receive an exception which informs them to fetch a current checkpoint from the local or another execution group to catch up (Line 26). The agreement protocol black box also garbage collects old slots and timeouts for client requests (Line 76). If the stable checkpoint is newer than the state of the agreement replica, then the replica also updates its state and fills in missing messages from the *hist* variable into the commit channel.

An agreement replica also manages an additional agreement window *win* to limit the active sequence numbers. Its size is defined by `AG-WIN` and must contain at least k_a sequence numbers. This window only moves forward when a checkpoint becomes stable and thus forces the agreement group to create checkpoints in regular intervals, otherwise the agreement protocol will block (Line 64). Without the agreement window, faulty agreement replicas could create a situation where a single correct agreement replica together with f faulty agreement replicas continues to forward EXECUTEs along the commit channel without ever creating a new checkpoint. This would either result in unlimited memory usage or prevent replicas from retrieving a checkpoint if necessary.

$t^+[c]$ is not part of the checkpoint as its content can differ between replicas. It is updated immediately when receiving a new client request at which point the replicas have not yet agreed on an order. However, as execution replicas explicitly request a move of the flow-control window for a client subchannel (Line 18), each agreement replica will eventually update the flow-control window accordingly and thereby also $t^+[c]$.

5.4.6. Execution Checkpointing

A checkpoint at an execution group consists of a snapshot of the application state and the reply cache, which for each client contains the latest REPLY. The reply cache must be included in the checkpoint as in other protocols [72, 84, 210] that only use $2f_e+1$ replicas for their execution. Both parts of the replica state contained in the checkpoint are only modified while processing messages received from the commit channel, which ensures that all replicas of a group arrive at the same state. The replicas of each execution group create a checkpoint every k_eth sequence number. For each execution group the checkpoint component then exchanges messages only among the group's replicas to stabilize the checkpoint. That is, checkpoints are created on a per-group basis.

Once an execution checkpoint is stable (Line 38), a replica moves its commit channel window forward (Line 39), which allows the garbage collection of the channel as soon as at least $f_e + 1$ replicas of the execution group have done the same. If the checkpoint contains a newer state, then the replica updates its state accordingly.

The capacity of the commit channel must be larger than the checkpoint interval k_e and should be large enough to shadow the time it takes for a checkpoint to become stable and for the updated flow-control window to propagate to the agreement group. Depending on k_e a commit channel capacity of $2k_e$ can be sufficient.

A replica or a group that has fallen behind might be unable to retrieve the next EXECUTE message from its commit channel. In this case, the replica queries its group and other groups for the current checkpoint (Line 26). When processing strongly consistent read requests it is possible for the reply cache to differ between execution groups as only a single group has executed the read request. All other groups store a placeholder instead. This allows the replicas in that case to inform the client that the reply is not available. If a client does not receive sufficient replies to its read request, then it has to repeat the request. This is unproblematic as read requests have no side effects and therefore can be executed repeatedly.

An execution replica does not include the request channel's state in its checkpoint. Messages on the request channel are not totally ordered yet and thus can differ between execution replicas. However, as a client has to resend its request until it has received $f_e + 1$ matching replies, a replica will eventually receive the client's current request and thereby synchronize the state of the client's subchannel. As execution replicas move the window of the request channel forward before sending a request (Line 18), this allows replicas to skip missing requests and just send the client's latest request.

5.4.7. Adaptability

SPIDER is able to adapt to changes in the workload at runtime by adding and removing execution groups. To adapt the system configuration, for example in response to clients started at a new location, a privileged admin client has to update the registry information and adjust the running execution groups as well.

More specifically, in order to add a new execution group, the admin client c_a first has to start replicas at the new location. Then it sends a signed $\langle \text{CONFIG}, t_{c_a}, \mathcal{C} \rangle_{\sigma_{c_a}}$ message to an execution group; the message is then ordered like a WRITE request. $\mathcal{C}$ is the system configuration signed by the admin client and includes the new execution group $\mathcal{E}$. Once the request has been ordered by the agreement group, then its replicas update their registry with the new, signed configuration $\mathcal{C}$. The agreement group also establishes a request and commit channel to the new execution group. The replicas in the execution group then try to receive their first EXECUTE via the commit channel, learn that they have fallen behind and query other groups for a current execution checkpoint.

If the execution group has to retrieve a large application state, then one possible solution is to add replicas to the system, which initially do not actively participate in the system but only fetch the application state [167, 180]. SPIDER can support a similar behavior with its global coordination mechanism described in Section 5.4.4 by temporarily

increasing the limit on fallen behind execution groups z by one. Once the state transfer has completed, then the admin client can reset z to its original value.

To remove an execution group $\mathcal{E}$, the admin client first issues an updated CONFIG request, which no longer contains the execution group $\mathcal{E}$. The agreement group shuts down the request and commit channel once the configuration has been updated. Afterwards the admin client can stop the replicas belonging to the old execution group.

5.5. Fault Handling

In the following we will discuss various ways in which faulty replicas may try to compromise the correctness or liveness and how SPIDER prevents such attacks by relying on the IRMC properties, which require a correct replica to vouch for data before forwarding it, and the individual building blocks.

Faulty Clients

A faulty client can try to attack the system in different ways. It could send different valid requests for the same client counter value to its execution group. As $f_e + 1$ execution replicas have to send the same value on the client's IRMC subchannel, this simply causes the subchannel to become stuck and prevents forwarding of the request to the agreement group. This only affects the subchannel of the faulty client, which therefore just prevents its own requests from being ordered, but does not affect other clients.

A faulty client collaborating with faulty execution replicas can send different requests to different agreement replicas for the same subchannel position. The faulty client sends a different request $m_1, \ldots, m_{f+1}$ to each correct execution replica $r_{e_1}, \ldots, r_{e_{f+1}}$, respectively, which then each forward the received request via the request channel. The client also sends all requests to the faulty execution replicas which send each request on the IRMC. Thereby, each of the requests $m_1, \ldots, m_{f+1}$ was sent by $f_e + 1$ replicas, which allows any one of it to be received by the agreement replicas.

However, the agreement protocol must be prepared to handle this situation, as a faulty client can also send diverging requests for the same client counter to the agreement protocol if it is used standalone. Duplicate requests are typically either filtered out during agreement or skipped during execution, such that the replicas only execute the first request for each client counter value [57, 63, 84, 210].

Faulty Agreement Replicas

On the agreement side, a faulty replica could try to send the wrong requests or the right requests in the wrong order to the execution groups. As sending a message across an IRMC requires at least one correct replica to vouch for it, such a faulty message is never delivered to the execution replicas. All correct replicas only send the correct result of the ordering process via the IRMC and therefore send identical messages. Thus, only the correct ordering result can be delivered to execution replicas.

A faulty leader in the agreement group could attempt to prevent a request from being ordered. As a correct client repeats a request until it receives a reply, the request will eventually arrive at all correct replicas of the client's execution group. These adjust

the window of the client's subchannel and send the request via the IRMC such that eventually all agreement replicas receive the request. This allows the agreement replicas to monitor the agreement process and to initiate a view change to replace the leader replica if it did not propose the client request within a timeout [22, 57, 63, 188].

A faulty agreement replica could attempt to erroneously skip a client request by calling `move_window` on the client's subchannel before the request was actually ordered. To garbage collect messages from an IRMC at least $f_a + 1$ replicas, that is, at least one correct replica, have to request that operation. However, correct agreement replicas will only do that after a request has been ordered and is included in a checkpoint.

Faulty Execution Replicas

A faulty execution replica could send a client's request at the wrong position on the client's subchannel or request that the subchannel is moved to a too high position. Neither attack has an effect, as at least $f_e + 1$ replicas including one correct replica would have to call the same IRMC methods.

Regarding the commit channel, an execution replica could prematurely request messages to be garbage collected. As correct execution replicas only request garbage collection after having created or obtained a checkpoint themselves, this ensures that messages are only garbage collected after a stable checkpoint exists at a correct replica.

Similar to a standalone agreement protocol, it is possible for faulty replicas to provide a client with faulty replies. However, as before a faulty reply is only sent by up to f_e faulty replicas and therefore does not reach the threshold of $f_e + 1$ matching replies that are necessary for a client to accept the result.

For weakly consistent read requests, the same constraint ensures that a client will only accept a reply that is supported by at least one correct replica. If a correct replica has fallen behind, then it is possible for the faulty replicas to support the reply provided by the lagging replica. However, this does not affect correctness as weakly consistent reads do not guarantee that a client receives the latest result.

Proof of Safety and Liveness

A detailed proof for the safety and liveness of write requests in SPIDER and their interaction with checkpoints is available in Appendix B.

5.6. IRMC Implementations

In this section we first explain supporting infrastructure relevant for the different IRMC implementations. This includes an event-based IRMC interface in Section 5.6.1 and the outbox abstraction in Section 5.6.2, which helps with maintaining a bounded state.

The interface of an IRMC is abstract enough to allow for different implementations with different trade-offs. One main decision is between the overhead for transmitting a message compared to the complexity of the implementation and the required error handling to properly tolerate malicious replicas. For example, a channel variant in which each sender replica just forwards each message to every receiver replica does not require special cases to handle the failure of any specific sender replica. In fact, it does not require

```
1  // Sender endpoint
2  interface IRMC_Sender_Event {
3      event channel_send(SubChannel sc, Position p, Message m)
4      callback channel_window_moved(SubChannel sc, Position p, Number limit)
5      event channel_skip(SubChannel sc, Position p)
6  }

8  // Receiver endpoint
9  interface IRMC_Receiver_Event {
10     callback channel_received(SubChannel sc, Position p, Message m)
11     event channel_move_window(SubChannel sc, Position p, Number tokens)
12     callback channel_skipped(SubChannel sc, Position p)
13 }
```

Figure 5.10: Event-based interface of sender- and receiver-side endpoints of an Inter-Regional Message Channel (IRMC)

any timing assumptions except that correct replicas are eventually able to communicate with each other. Therefore, this variant can even work over an asynchronous network which allows messages to be arbitrarily delayed as long as they arrive eventually. However, the weak network assumptions come at the price of a considerable message transmission overhead. The Inter-Regional Message Channel with Receiver-side Collection (IRMC-RC) variant is based on this design and is described in Section 5.6.3.

By using the timing assumptions of *partial synchrony* [80] (cf. Section 2.1.2), we can reduce the number of repeated message transmissions required. The partial synchrony model assumes that the network alternates at arbitrary points in time between synchronous and asynchronous phases, that is, with or without time bounds until messages are delivered. To use this in an IRMC, a receiver replica can then select a sender replica responsible for forwarding messages. If forwarded messages are invalid or are not forwarded within a certain timeout, then the receiver switches to a different sender. The corresponding Inter-Regional Message Channel with Sender-side Collection (IRMC-SC) variant is described in Section 5.6.4.

5.6.1. Event-Based Interface

Directly implementing the synchronous interface for an IRMC, as presented in Figure 5.4, requires one execution thread for each client and execution group. As our prototype implementation is written in Java 11, which does not natively support user-level threads, the interface would require the use of kernel-level threads whose overhead [41, 206] becomes prohibitive for thousands of clients and multiple execution groups. Instead, our implementation uses an event-based IRMC interface that is presented in Figure 5.10. Handling an event requires a small amount of processing, but never blocks the current

thread, such that a single thread can process events from many different subchannels to avoid the overhead of using many kernel-level threads.

In the following we refer to the component of a replica that interacts with an IRMC endpoint as *caller*. Each event method in the interface is triggered when the caller sends the corresponding event to the endpoint or in the case of a callback the event is sent by the endpoint to the caller. We will only discuss how the event-based interface relates to the synchronous interface, refer to Section 5.3.2 for a general description. To distinguish both interfaces, all methods of the event-based interface are prefixed with `channel_`.

Sender Endpoint

The `send()` method of the synchronous interface is equal to a combination of the event `channel_send(`sc`, `p`, `m`)` and the callback `channel_window_moved(`sc`, `p`, `$limit$`)`. `channel_send(`sc`, `p`, `m`)` sends a message m at position p of subchannel sc and must be called in-order for each position. A message m sent at a position p before the flow-control window is silently discarded. Sending at a position p after the subchannel window is not allowed. The `channel_window_moved(`sc`, `p`, `$limit$`)` callback informs the caller that the window for subchannel s now starts at position p and ends before position $p + limit$. It therefore replaces the blocking behavior of `send()` by instead explicitly propagating the flow-control window to the caller of the sender endpoint. `channel_skip(`sc`, `p`)` is directly equivalent to the `move_window(`sc`, `p`)` method of the sender endpoint, which requests that the flow-control window for subchannel sc is moved forward to position p.

Receiver Endpoint

The `receive()` method of the synchronous interface returns either message m for position p in subchannel sc, which maps to the `channel_received(`sc`, `p`, `m`)` callback, or a TooOld message, which is replaced by the `channel_skipped(`sc`, `p`)` callback.

The caller can use the event `channel_move_window(`sc`, `p`, `$tokens$`)` to move the start of the flow-control window for a subchannel sc to position p, which corresponds to the `move_window()` method of the receiver endpoint in the synchronous interface. The parameter $tokens$ specifies that only messages at a position starting from p, up to and excluding $p + tokens$ may be forwarded to the caller. $tokens$ is independent of the flow-control window and instead enables the caller to regulate or even stop message delivery by the receiver endpoint to ensure the replica is not overwhelmed. This has a similar effect as not calling the `receive()` method of the synchronous interface.

5.6.2. Bounded State

The core idea to ensure a bounded state at the IRMC endpoints is to only store a fixed number of messages by making use of the flow-control mechanism. Thus, endpoints only keep messages (a) that belong to a bounded window of slots and discard all other messages or (b) that are used to update a fixed amount of state. Forwarded message slots fall into the former category, whereas auxiliary messages like those used to update the flow-control windows, which can be stored in aggregate, fall into the latter.

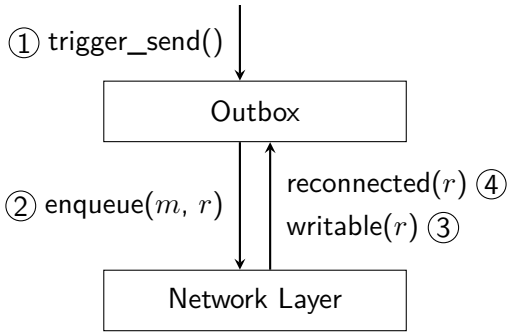


Figure 5.11: General interaction between an outbox and the network layer.

Outbox

We introduce an abstraction called *outbox* to encapsulate most of the implementation complexity on the sender side. Although the description here is focused on IRMCs, we expect the concept to also be applicable to other parts of a replica implementation.

An outbox buffers information to be sent to other replicas either in the form of already assembled messages or as raw data in which case the outbox can generate messages on demand. This is especially beneficial for summable data, like monotonic counters, as the generated messages can contain an aggregation of the latest data right before sending these messages. By delaying the transmission of new information for a short time, it can be possible to increase the amount of data that can be combined into a single message.

The general interface of an outbox, shown in Figure 5.11, only defines the interaction with the network layer, but not how messages or data are added to the outbox. That way, outboxes can store different types of data without affecting the network layer. After an outbox is marked as ready to transmit data ①, it calls `enqueue(`m`, `r`)` to queue messages m for transmission to replica r ②. The network layer in our implementation collects a small number of messages in a send queue to each replica, before passing them on to the operating system to reduce the overhead of executing the syscalls [187]. If the network layer does not have sufficient capacity in the send queue to a certain replica, then the enqueued message is rejected. This resembles the interface of the non-blocking send syscall [123], which only accepts some or no new bytes if the buffers within the operating system have filled up. In case a message is rejected, once the connection has free space available, the network layer notifies the outbox that the connection to a replica r is `writable(`r`)` again ③, causing the outbox to retry sending its messages.

An outbox must offer a flow-control mechanism to ensure that it only sends messages if the receiving replica is ready to handle them in order to not overwhelm the receiver and avoid wasted network traffic. This is either the case when the receiving replica has sent a matching flow-control window or when the outbox is sending aggregate messages which do not require additional state for storage.

Retransmissions

To ensure reliable message delivery, we use transmission control protocol (TCP) connections between replicas, which handle most cases of packet loss. However, in case of

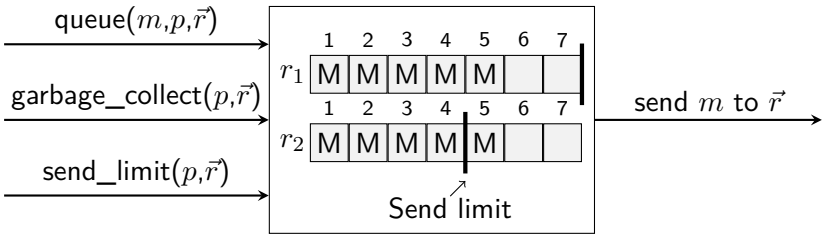


Figure 5.12: A message outbox manages a queue of messages for each target replica. The outbox allows queuing of new messages, garbage collecting old messages and configuring the send limits for one or multiple replicas $\vec{r}$.

interrupted TCP connections all messages currently queued in the TCP transmission buffers are lost. To avoid the need to collect fine-grained feedback on which messages were lost exactly, once the outbox receives a `reconnected(`r`)` notification that the connection to a replica r is reestablished ④, it must retransmit all messages that were not garbage collected in the meantime.

PBFT approaches retransmissions from a different angle: it first broadcasts messages to all replicas and then exchanges status messages in which replicas inform each other about received messages to handle message loss [56]. Missing messages are then resent by the corresponding replica. This approach relies on efficient multicast primitives; however, these are either not available [102, 158] in today's cloud environments or incur additional costs [17]. Without multicast support messages would have to be sent to each replica individually, making it costly to forward requests speculatively.

Message Outbox

The *message outbox* manages sending a sequence of messages. As shown in Figure 5.12, a message can be sent using `queue(`m`, `p`, `$\vec{r}$`)`, which enqueues a message m at position p for sending to the replicas in $\vec{r}$. Internally, the outbox maintains a FIFO queue of messages for each target replica. Messages in the queue are ordered by their position number, which also determines the transmission order.

The capacity of the queues is fixed such that messages eventually have to be garbage collected. The outbox allows garbage collecting all messages below a certain position p using the `garbage_collect(`p`, `$\vec{r}$`)` method. Depending on the list of replicas $\vec{r}$ it applies to all or only individual replicas. For example, once a target replica has confirmed the receipt of a message, this allows garbage collecting all earlier messages for this replica.

The flow control is managed using `send_limit(`p`, `$\vec{r}$`)` to set per-receiver send limits as shown in Figure 5.12. The outbox will for each replica only send messages up to the individual limit, which must be set to the highest sequence number the receiver is ready to handle. That way, individual slow receivers are not overwhelmed with messages.

For an IRMC, our implementation uses a separate message outbox for each subchannel to manage the messages queued inside the subchannel window. The garbage collection at the lower bound of the flow-control window of an IRMC subchannel is managed via the `garbage_collect()` method. And `send_limit()` provides the mechanism to implement the per-receiver flow control limits of an IRMC.

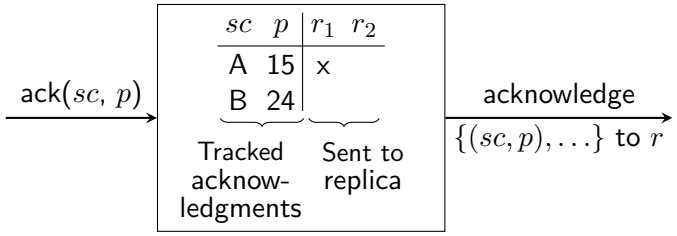


Figure 5.13: An acknowledgement outbox tracks the acknowledged position p for each subchannel sc. It tracks which replica has been informed about which acknowledgement and constructs messages accordingly.

Acknowledgement Outbox

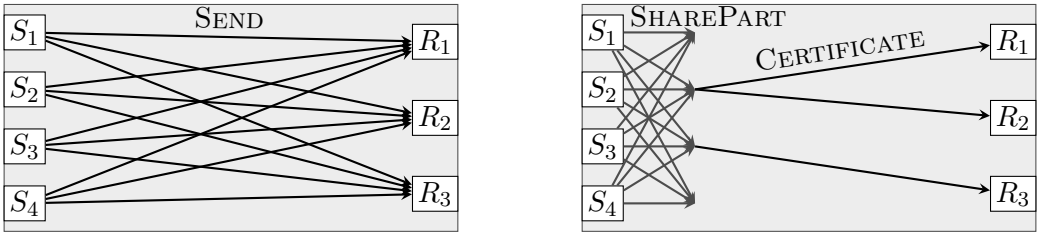
The *acknowledgement outbox*, which is shown in Figure 5.13, stores for each subchannel sc one counter or position p whose value must increase monotonically and is set using the `ack(`*sc*`, `*p*`)` method. As only a limited number of subchannels exists, this guarantees a bounded state size for the outbox. The outbox can, for example, be used to communicate the lower bound for a subchannel window after it was changed by a call to `move_window`. An acknowledgement outbox generates messages on demand, that is, right before passing the data to the network layer. It always sends all changed positions to each target replica and thereby batches as many positions together as possible. For that the outbox also tracks to which replicas it has already forwarded the current value of a position.

If a subchannel position increases multiple times before sending, then the outbox only stores the latest position and thereby merges these updates. The sender can (slightly) delay sending the acknowledgements to allow for further updates to accumulate.

In detail, the position transmission works as follows: The outbox at a replica r_i sends an authenticated message $\langle \textsc{Move}, l, \vec{w} \rangle_{\mu_{r_i,r_j}}$ to replica r_j which contains a transmission counter l that is increased by one for every MOVE message sent to r_j to prevent message replays or reordering. The receiver must ignore message with old transmission counter values. A MOVE message is only authenticated for r_j as it is constructed individually for each receiver. The vector $\vec{w}$ consists of pending tuples (sc, p) for replica r_j, where sc is a subchannel and p the latest position. The outbox then records which tuples were sent. This transmission process is repeated for every replica to which tuples are pending.

5.6.3. Inter-Regional Message Channel with Receiver-side Collection

An Inter-Regional Message Channel with Receiver-side Collection (IRMC-RC) has a simple design. Each message passed to the IRMC is sent by every sender endpoint to every receiver endpoint. The resulting message pattern is shown in Figure 5.14a; for pseudocode, please refer to Appendix B.4.1. Each receiver endpoint then collects a bundle of $f_s + 1$ matching messages, hence the name. More precisely, for a message m sent on subchannel sc at position p, each sender endpoint s_i creates a signed message $\langle \textsc{Send}, m, sc, p \rangle_{\sigma_{s_i}}$ and adds it to its message outbox for delivery to the receiver group $\mathcal{R}_*$. The messages are signed to prevent tampering and to allow the receivers to ascertain the identity of the sending endpoint. Once a receiver endpoint has received the



(a) IRMC-RC **(b)** IRMC-SC

Figure 5.14: Message patterns to transmit a message via two possible IRMC implementations.

exact same message m on the same subchannel sc and position p from at least $f_s + 1$ different endpoints of the sender group S_*, then it delivers the message locally. A receiver only delivers the first message m that arrives from $f_s + 1$ senders for each subchannel position, even if it receives another message m' that is also supported by $f_s + 1$ different senders. As discussed in Section 5.5 different receivers are allowed to deliver different messages in that case.

The movement of the flow-control windows is managed using acknowledgement outboxes that transfer the lower bounds of the flow-control window of a subchannel. The sender endpoints store the highest received acknowledgement for each subchannel and receiver endpoint. For each subchannel, the $f_r + 1$ highest value is then used as the new start position for the flow-control window.

The same approach applies to `move_window` calls at the sender endpoints. However, there a receiver endpoint computes the combined value based on the $f_s + 1$-highest value and automatically calls `move_window` internally every time the value changes. If the combined value is lower than the current window start, then the `move_window` call ignores the value. Otherwise, the window moves forward.

The IRMC-RC does not rely on timing assumptions, instead it will make progress if sufficient Send messages are transmitted before the receiver side garbage collects them. The implementation itself is still subject to timeouts in the communication protocol, in our case in TCP. After network problems, progress is possible as soon as the connections between a sufficient number of replicas have been reestablished. Therefore, the IRMC-RC implementation removes the need for Spider to make timing assumptions for its wide-area communication. However, this comes at the cost of requiring $|\mathcal{S}_*| \cdot |\mathcal{R}_*|$ transmissions of the message forwarded via the IRMC.

5.6.4. Inter-Regional Message Channel with Sender-side Collection

The Inter-Regional Message Channel with Sender-side Collection (IRMC-SC) variant is a more complex variant that transmits fewer data than an IRMC-RC. For pseudocode, please refer to Appendix B.4.2. Even more network-efficient variants could be based on the approach of BLink [20]. As shown in Figure 5.14b, an IRMC-SC adds a collection step at the sender side, where the sender endpoints act as collectors [24, 108] and assemble a certificate proving the correctness of the transmitted message. This certificate allows

an individual sender endpoint to forward the certificate to the receiver endpoints. Those can select the sender endpoint from which they want to receive the certificate.

Signature Exchange

In order to assemble a certificate, for each subchannel sc and position p, the sender endpoints exchange $\langle \textsc{SharePart}, sc, p, h(sm) \rangle_{\sigma_{s_i}}$ messages with each other, containing the hash $h(sm)$ of the Send message sm that should be forwarded. The exchange of the SharePart messages is handled using message outboxes that are configured to only accept messages for positions within the current flow-control window of the subchannel. As sender endpoints can learn about window movements at different times, the sender endpoints use an additional acknowledgement outbox to inform each other about the lowest position for which they still have to collect a certificate. The other sender endpoints thereby learn which SharePart messages are still required by an endpoint and whether it is ready to receive further SharePart messages. This serves as a flow-control mechanism for SharePart messages between the sender endpoints.

Certificate Forwarding

As soon as a sender endpoint s_i has collected a vector $\vec{v}$ consisting of $f_s + 1$ valid signatures from different senders for the same message sm, then it assembles these into a certificate $\langle \textsc{Certificate}, sm, \vec{v} \rangle_{\alpha_{s_i,R_*}}$. The Certificate is authenticated using a MAC authenticator [57] to prevent tampering during transmission. The vector $\vec{v}$ only includes the signer identities and the signatures itself; the remaining parts of the SharePart messages can be reconstructed from the message sm. To create the certificate, the sender endpoint needs the full message content of the Send message sm and not just the hash. This will be the case once the `send()` method of the specific sender endpoint has been called with the corresponding message m for subchannel sc at position p. A sender endpoint then forwards the certificate to one or more receivers, as selected by them. Each receiver verifies the authenticator's correctness, that each contained signature matches message sm and that the signatures originate from $f_s + 1$ different sender endpoints. If these are valid, then the receiver has proof that at least one correct sender endpoint confirmed the validity of message sm and is allowed to deliver sm.

Sender Selection

Each receiver endpoint selects the sender endpoint from which it wants to receive the certificates for a subchannel. To communicate the selection, a receiver attaches its sender selection for a subchannel to the messages sent by the acknowledgement outbox. For this, it sends the identifier of the selected sender as the position value for a pseudo subchannel $-sc - 1$. Contrary to values for regular subchannels, the value is not required to increase monotonically, that is, the sender endpoints must accept a lower received value for a pseudo subchannel. The sender endpoint now configures the per-receiver send limit for the message outbox that transmits the Certificate messages accordingly.

A faulty receiver replica can use the selection mechanism to cause multiple sender endpoints to each transmit the certificates. To detect such misbehavior, the amount of data received by an endpoint can be compared with that of other endpoints of the same

group. If it is much higher than that of the other replicas, then the replica should be considered as possibly faulty and be replaced with a new one.

To prevent faulty senders from suppressing messages, each sender endpoint periodically transmits $\langle \text{PROGRESS}, l, \vec{p} \rangle_{\mu_{s_i,r_j}}$ messages that contain a vector $\vec{p}$ to inform the receivers about the highest position in each subchannel for which the sender has collected a certificate. As the slots in an IRMC are filled sequentially and without gaps, this also automatically means that the sender endpoint has a certificate for each slot up to and including that position. The messages are generated by an acknowledgement outbox modified to send PROGRESS messages.

The receiver endpoints then calculate the $f_s + 1$ highest received position for each subchannel and use that to monitor the progress of the selected sender endpoint for that subchannel. If the selected sender endpoint does not transmit a valid certificate for the position within a timeout, then the receiver endpoint switches to another one of the endpoints that according to their PROGRESS message claim to have the certificate. While the network stays in a synchronous phase, this ensures that a receiver endpoint only has to switch between $f_s + 1$ different sender endpoints to find one that is actually able to supply the certificate, as only up to f_s sender endpoints can be faulty.

As long as the selected sender replicas work correctly, the IRMC-SC variant only requires $|\mathcal{R}_*|$ wide-area transmissions of the sent message.

5.7. Optimizations

As described so far, each IRMC variant requires creating and verifying multiple signatures for each message and IRMC. In the following we start with an optimization in Section 5.7.1 that allows reusing certificates at the agreement group, followed by two further optimizations to amortize the signature creation and verification costs over many messages in Section 5.7.2, and to offload the signature verification step from the agreement group to the execution groups in Section 5.7.3. We finish this section with another two optimizations to strengthen the consistency guarantees offered for weakly consistent read requests in Section 5.7.4 and to ensure that these can be answered by execution groups even if the agreement group is temporarily not reachable in Section 5.7.5.

5.7.1. Signature Sharing between IRMCs

The agreement group sends the exact same ordered requests to every execution group. The only deviations are caused by the optimization for strongly consistent read requests described in Section 5.4, which only sends the full request to the client's execution group. By structuring the message that is forwarded along the commit channel as described in the following, the message always has the same message hash, even if a read request is omitted for some groups. This allows the IRMC to reuse signatures for the SEND messages created by the agreement group's sender endpoints. This also applies to the corresponding SHAREPART messages used by the IRMC-SC variant, which are now also identical across sender endpoints. Thus, the sender endpoints of the commit channels at a replica are able to share these signed messages and perform all signature processing only once. Each

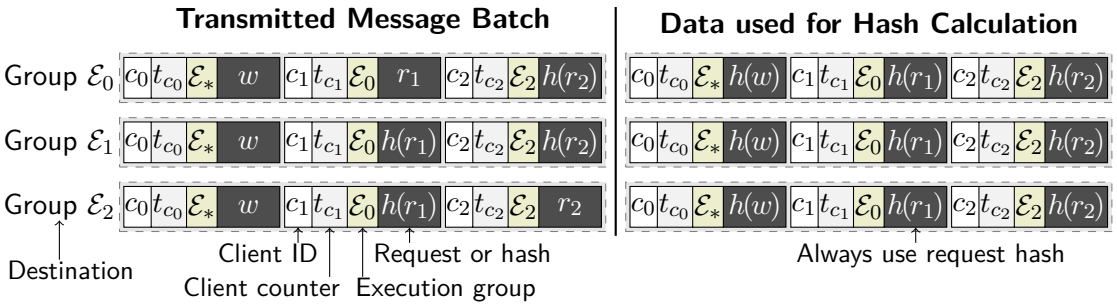


Figure 5.15: Format of a batch request m that allows omitting read requests to certain execution groups without changing the message hash $h(m)$, which is calculated using the representation on the right. The write request w is sent to all groups, which is indicated by $\mathcal{E}_*$, read request r_1 only to execution group $\mathcal{E}_0$ and r_2 to $\mathcal{E}_2$.

agreement replica then only has to calculate a single signature for each forwarded message independent of the number of execution groups.

Handling Strongly Consistent Read Requests

When computing the message hash over the complete content of an ordered request, then replacing a strongly consistent read request with a placeholder would change the message hash and thus require a different signature. As the agreement protocol typically bundles together multiple requests into a batch [57], which is then ordered instead of the individual messages, this can in the worst case require a different signature for each execution group and prevent the signature sharing between IRMCs.

SPIDER therefore uses a modified batch structure that still allows sharing the signature between IRMCs. As shown on the left side of Figure 5.15, the batch transmitted to a group contains for each request the client identifier c, its counter value t_c, the group identifier $\mathcal{E}$ and the request or its hash. If the group identifier matches that of the destination group, then the full request is included, otherwise only its hash. The identifier $\mathcal{E}_*$ matches every group. To ensure identical hashes for the batch message sent to each group, the hash calculation only considers request hashes as shown on the right side of Figure 5.15, independent of whether the full request or only its hash is transmitted. Thus, the hash of the batch message sent to the different groups is always identical.

The modified hash calculation introduces the risk that a faulty replica could suppress a certain request in the batch by only sending its hash. In order to prevent such tampering, a receiver must verify that the batch message contains the full request if and only if the group identifier is $\mathcal{E}_*$ or matches the receiver's own group identifier. This verification must be implemented in the receiver endpoint to maintain the guarantee that the receiving side only delivers messages vouched for by $f_s + 1$ replicas. For example, in Figure 5.15 the batch message sent to group $\mathcal{E}_0$ must include the full messages w and r_1, but only the hash of request r_2. This ensures that group $\mathcal{E}_0$ can correctly process the batch message and only receives as much data as necessary.

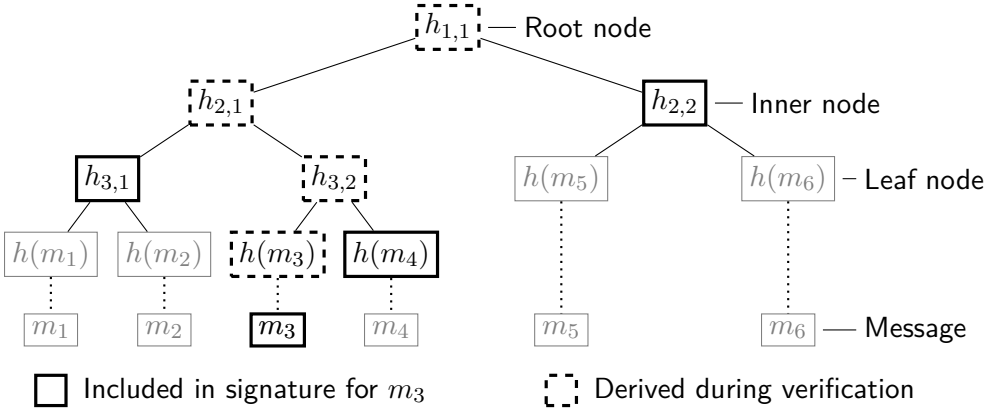


Figure 5.16: Structure of a Merkle tree used to verify six messages. The leaf nodes contain the hash of the corresponding messages, whereas all inner nodes and the root node use the hash of their child nodes as value. The highlighted nodes are considered when verifying that m_3 is part of the tree.

5.7.2. Signature Batching

Calculating a signature at each sender endpoint for every request transmission on an IRMC is still expensive. For the commit channels, the costs can easily be amortized over multiple requests by using the batching optimization typically available in agreement protocols [22, 39, 47, 57, 108, 130], which packs multiple requests together and then only agrees on the batch instead of individual requests. However, this does not work for transmitting client requests on the request channel, where no total order exists in which requests from different clients are handed over to the IRMC.

Merkle Tree

Instead, it is possible for each endpoint to individually group messages in a Merkle tree [154] and only sign the tree root to amortize the signature costs over all messages in the tree. The tree provides proof that a certain message is part of it and the root node signature thereby indirectly proves the validity of each message contained in the tree. This approach is used similarly to sign unordered responses in Basil [194].

A Merkle tree [154] is a binary tree whose leafs contain hashes of the contained messages, whereas the value of an inner node is determined by the hash of its two child nodes. An example of such a tree is shown in Figure 5.16. The use of a collision-resistant hash function [153] is required to guarantee that an attacker cannot construct alternative tree nodes with the same hash. By applying the hash function recursively, this results in a root node containing the root hash of the tree. To create a proof, it is sufficient to sign the root hash and include enough intermediate hashes to allow a verifier to confirm that the tree contains a certain message. The verifier then starts with the hash of the message it wants to verify and follows the path through the tree upwards to the root node. As the value of each inner node is the hash of the left and right child node, the signer at each level only has to include the child node that is not part of the path followed by the

verifier. Note that the root hash is also not included, as it is calculated while verifying the path through the tree. The final step is to verify that the signature matches the computed root hash.

For example, to prove in Figure 5.16 that message m_3 is part of the tree, the signer would send message m_3, $h(m_4)$, $h_{3,1}$ and $h_{2,2}$ together with a signature of the tree root to the verifier. For the verification, it has to follow the path from m_3 to the root node. First $h_{3,2} = h(h(m_3)||h(m_4))$ is calculated, followed by $h_{2,1} = h(h_{3,1}||h_{3,2})$ and so on. As last step, the verifier checks that the signature is for $h_{1,1}$ and is valid.

Tree Signatures

An IRMC uses Merkle trees as follows: A sender endpoint collects multiple messages that should be sent on any of the subchannels of the IRMC, assembles these into a Merkle tree and then signs the root hash. The sender now attaches $\langle mc, i, \vec{h}, sig \rangle$ as signature to each message, where mc is the number of messages included in the Merkle tree, i is the message index, $\vec{h}$ the list of intermediate hashes necessary to verify the path from the message to the tree root and sig is the signature of the root hash. We refer to this type of signature as *tree signature* in the following. As each tree signature is independently verifiable from the other messages in the same Merkle tree, the tree signature can be used as a drop-in replacement for normal signatures. This also simplifies garbage collection of old messages, as there are no dependencies across multiple requests.

To ensure that the combination of the message index i and the message count mc uniquely defines the path from a message to the tree root, we define the shape of the tree to only depend on the message count. As shown in Figure 5.16, the messages are included in a complete binary tree, which means that all levels except the last are completely filled, and the last level is filled from left to right. Each node is either a leaf node or has two children. The messages are then assigned to leaf nodes of the tree.

The receiver side has to verify that the message is part of the Merkle tree and that there is a valid signature for the tree root. The result of the signature check should be cached to avoid multiple verifications of the same signature when different messages belonging to the same Merkle tree arrive. The size of the signature-check cache for each sender replica should be limited to the number of slots an IRMC can store at a time.

For optimal efficiency, it is necessary to apply tree signatures to a sufficiently large number of messages. For this purpose, an IRMC should - like with request batching - wait a short amount of time between creating two signatures unless enough messages have arrived to reach a predefined size limit for the Merkle tree.

5.7.3. Client Request Verification Offloading

The agreement protocol has to verify that a client request is correctly authenticated before allowing it to continue through the agreement process. Otherwise, a faulty client or replica could issue requests pretending to originate from a different client and thereby bypass access control mechanisms [57]. Spider uses signatures to authenticate client requests, which prevents corner cases where a faulty client could create a request that is only regarded as valid by some but not all replicas [57, 62, 63].

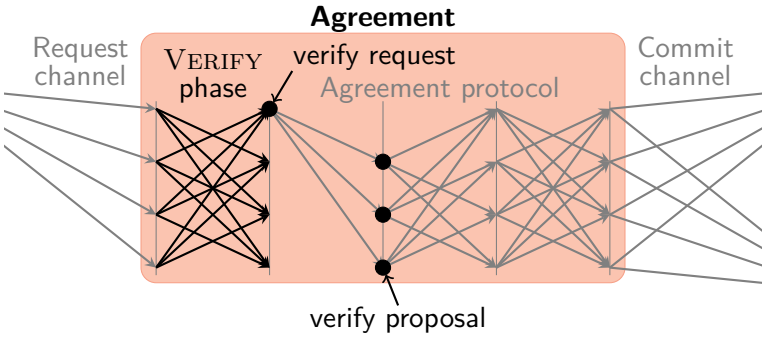


Figure 5.17: Agreement group including the VERIFY phase necessary to offload the request verification. Parts drawn in black are modified in comparison to an unoptimized agreement group.

Signature verifications have a large computational overhead. For RSA-based signatures, as we show in Section 5.8.6.1, the signature verification takes on average 14 µs (microseconds) in our testbed. The performance impact is limited as the verification is more than an order of magnitude faster than creating a signature. However, with modern elliptic curve based signatures that ratio changes and verifying a signature can be slower than creating it. For example, it takes on average 80 µs to create an ed25519 [139] signature and 173 µs to verify it. That is, the verification costs can become a computational bottleneck at a throughput of several thousand requests per seconds.

To avoid these costs at the agreement replicas, it is possible to use the fact that execution groups already verify all requests before forwarding them via the request channel. Instead of verifying the request signature a second time at the agreement group it is therefore sufficient to verify that a request was transferred via the IRMC, which guarantees that at least one correct execution replica has successfully verified the request. Thereby, the agreement group can offload the request verification to the execution groups.

Proof of Transfer

This requires the agreement replicas to obtain a proof that a request was forwarded via the request channel, in the following referred to as *proof of transfer*. It is constructed as follows. We require that the agreement group consists of at least $3f_a + 1$ replicas. After an agreement replica r_i receives requests $r = \langle \text{REQUEST}, m, \mathcal{E} \rangle$ via the request channel, instead of passing them directly to the agreement protocol using `ag.order_request()` (Line 59 in Figure 5.9), it first runs a verification phase. For this the replica sends a $\langle \text{VERIFY}, \mathcal{D} \rangle_{\mu_{r_i,r_j}}$ message to each other agreement replica r_j as shown in Figure 5.17. $\mathcal{D}$ contains a set of message descriptors $\langle c, t_c, h(r) \rangle$. Each descriptor states that the replica knows a request from client c with counter value t_c and hash $h(r)$ that has arrived via the request channel. For efficiency reasons, multiple descriptors for requests that arrive in a short time interval are batched together. The verification nevertheless works at the granularity of individual descriptors.

Once a replica has received $f_a + 1$ matching descriptors from different replicas for the same request, then it also sends a VERIFY message of its own. After a replica has

received $2f_a + 1$ matching descriptors, these form a proof of transfer and thus allow the request to be ordered. Each replica that has received the request via a request channel then passes the request on to the agreement protocol.

The construction of a proof of transfer ensures that once a correct replica has obtained it then eventually all other correct replicas will obtain one too. Collecting a proof of transfer requires at least VERIFY messages from f_a+1 correct replicas. Thus, every correct replica will receive $f_a + 1$ VERIFYs and send one of its own. As at least $2f_a + 1$ replicas are correct, this allows every correct replica to obtain a proof of transfer.

The agreement protocol must only act on requests for which the replica has obtained a proof of transfer. If the agreement protocol receives a request via the request channel or a replica forwards a request to the current leader replica, then, as shown in Figure 5.17, the agreement protocol at the leader must *verify* the *request* before proposing it. This is necessary to ensure that the request can be verified by other correct replicas. When a replica receives a request proposal (i.e., a PREPREPARE message when using PBFT) in the agreement protocol from the leader replica, then it has to *verify* the *proposal* to prevent the ordering of unverified requests. If no proof of transfer is available, then the agreement on the proposal is blocked, until the replicas obtain the proof. Both cases require the agreement protocol to provide a hook to allow verifying that a proof of transfer exists for the request in question.

If multiple descriptors with the same client c and counter value t_c obtain a proof of transfer, then this proves misbehavior on part of the client. The replicas could then agree to ignore the faulty client.

Garbage Collecting Proofs

To keep the set of request descriptors from growing without bounds, it has to be garbage collected from time to time. Whenever a checkpoint at the agreement replicas becomes stable, all descriptors for each client c with a smaller client counter t_c than the value in the checkpoint ($t[c]$ at Line 71 in Figure 5.9) can be garbage collected, as a replica is able to update its state by applying the checkpoint and no longer needs the old proof of transfer. To properly handle old requests proposed in the agreement protocol, a client request with a timestamp that was garbage collected is always considered as valid.

Without this exception, the agreement protocol could become stuck. Assume that the leader replica proposes a valid, but old request that was already garbage collected at the other replicas. Then the agreement protocol would block while trying to verify the request and would eventually replace the leader replica. In contrast, with the rule to accept all garbage-collected requests, the agreement will be able to continue. Note that this does not affect correctness as client requests with old counter values will be ignored during execution (Line 30 in Figure 5.8).

Bounded State

The fields in a request descriptor $\langle c, t_c, h(r)\rangle$ are not self-verifying, that is, based on a single descriptor, it is not possible to tell whether their values are correct or not. To prevent faulty replicas from flooding the system with invalid descriptors, we limit the number of descriptors which a replica has to store.

Each correct replica for each unique *client request identifier* (c, t_c), with client c and counter value t_c, must only accept the first request it receives via the request channel. As a faulty client that sends diverging requests to a sufficient number of execution groups could cause each correct agreement replica to receive a different client request, $N_a - f_a$ valid descriptors can legitimately exist. That is, it is sufficient if an individual replica is able to report $N_a - f_a$ different descriptors for each client request identifier (c, t_c). All further descriptors from the replica for this client request identifier must be ignored.

The maximum number of request descriptors, which a replica may have to store is calculated as follows. As a correct replica only sends a descriptor when receiving the request itself or after observing that $f_a + 1$ replicas support a descriptor, among which at least one replica must be correct, this yields a limit of up to $N_a - f_a$ valid descriptors that can be sent by all correct agreement replicas together. As each of the f faulty replicas can also report $N_a - f_a$ different descriptors, this results in at most $(f_a + 1) \cdot (N_a - f_a)$ descriptors that have to be stored for a client request identifier.

To prevent an individual client c from flooding the system by continuously proposing new requests, the agreement replicas only read a new request from a client's request subchannel after having completed the processing of the previous request. That is, the call to `ag.order_request()` (Line 59 in Figure 5.9) must not return before the current request or a newer one has been ordered. Additionally, only a limited number of requests per client may be ordered between two agreement checkpoints. For each client only the next k_a positions in its subchannel may be processed, starting from the counter value $t[c]$ included in the last checkpoint, where k_a is the checkpoint interval of the agreement group. This is not a limitation for correct clients, as these only send a new request after the previous one was ordered and executed.

Descriptors may only be exchanged between replicas if the sender knows that the receiver is ready to accept them. For this, a replica that receives a request descriptor $\langle c, t_c, h(r) \rangle$ interprets it as an implicit permit to send descriptors to this sender for this client up to counter value $t'_c < t_c + k_d$, where k_d is a small constant, which we set to 2. This slows down flooding request descriptors without affecting correct clients.

If applying a checkpoint causes a replica to skip some counter values, then it has to broadcast a special descriptor value $\langle c, t_c, \perp \rangle$ using the updated counter value t_c. As this special value only updates the flow-control limits, but does not result in a descriptor that has to be stored, it is not subject to the normal limits for exchanging descriptors and can always be sent immediately.

5.7.4. Reading with Sequential Consistency

Weakly consistent read requests as described in Section 5.4.3 only ensure prefix consistency. SPIDER guarantees that the returned result is correct as a client waits for $f_e + 1$ matching replies and therefore at least one of the replies is from a correct execution replica. However, there is no guarantee on how recent the state of the answering replicas is. In the worst case, the $f_e + 1$ replies consist of one reply from a correct but outdated replica and of f_e replies from faulty replicas supporting the old reply.

All correct replicas answer a client's read request based on their local state. However, there are no checks on whether the state of a replica is up-to-date or not. Even though all replicas in principle process the same sequence of write requests and thus proceed through the same states, they may execute the read requests at different logical points in time and therefore answer based on different states. This allows faulty replicas to provide enough support for any of the replies provided by correct replicas to let the client accept that reply. Regarding consistency this has the problem, that a client can alternatingly receive replies based on an older or more recent application state.

The property that different correct replicas can execute a request at different points in time and provide different replies is specific to weakly consistent read requests as these are executed directly by the execution replicas and thereby bypass the agreement process. In contrast, for write requests, which must be ordered prior to execution, a replica is only able to execute the request after learning the agreement result and thus by construction all correct replicas execute the request at the same sequence number.

Sequential Consistency

In the following we strengthen weakly consistent read requests to offer sequential consistency. It guarantees that all operations appear to happen according to a single total order, such that for an *individual* client's operations these execute in the order they were issued by the client [31, 204]. However, it is still possible for one client to complete a write request but for another client to still read an older state afterwards. In comparison to strong consistency, or more formally linearizability [118], sequential consistency only requires the order of operations of each individual client to match the order in which the operations were issued and does not require that order to hold across operations from different clients.

As read requests, which by definition have no side effects, are only relevant for the requesting client, it is sufficient to totally order only the write requests, as these define the states through which the system progresses. The only restriction from the view point of a client is that after reading a value, it must always see the same or a later state. Write requests in SPIDER are already ordered with strong consistency, therefore we only have to guarantee that for a client weakly consistent read requests are processed after all earlier write requests of that client (*read-my-writes*) and that read requests access the same or a newer state as the previous read request (*monotonic reads*) [204].

Part 1: Read my Writes

In order to solve the first requirement, similar to WEAVE [89], we add a minimum sequence number a to $\langle \text{READWEAK}, w, c, t_c, a \rangle_{\mu_{c,r_i}}$, which specifies the sequence number a replica must have executed before executing the read request. The reply to a write request $\langle \text{REPLY}, u_c, t_c, s_n \rangle_{\mu_{r_i,c}}$ is extended with the sequence number s_n at which the request was executed. A client now has to wait for $f_e + 1$ replies to its WRITE containing the same result u_c and sequence number s_n. As all correct replicas execute an operation at the same sequence number, both their result and sequence number match, which guarantees that a client will be able to receive $f_e + 1$ matching replies. After accepting a result for a write request, the client then updates its minimum sequence number a to s_n.

This guarantees that correct replicas will only reply after they have executed the write request and therefore also prevents faulty replicas from injecting older replies.

Part 2: Monotonic Reads

The second requirement is harder to satisfy. For it a client has to include the sequence number of the system state it has already read from in its later requests. As weakly consistent read requests are not ordered, there is no fixed sequence number at which the request will be executed, and consequently it is unclear which sequence number included in the individual replies is correct and which not. Therefore, we modify the reply for a ReadWeak request to include the sequence number of the latest state change that had an influence on the reply. In the case of a key-value store this could be the sequence number at which a key was last written. Similar to write requests, a client then waits for $f_e + 1$ matching replies with identical results and sequence numbers. The client then only updates the minimum sequence number a if the received sequence number s_n is larger. If a client does not accept a reply within a timeout, then the client issues the request again as a strongly consistent read request.

If the last change occurred at an older sequence number than the requested minimum sequence number a specified by the client, then the replies prove that the result has not changed at least up to sequence number a. Using the sequence number of the last change has the benefit that it will very likely change much less frequently than the latest sequence number that a replica has processed and thus increases the chance that all correct replicas return the same sequence number.

5.7.5. Reading with Interrupted Wide-Area Communication

As long as the agreement group and an execution group can communicate with each other, then the execution group will eventually receive every message that was forwarded to it across an IRMC. And therefore all replicas in the execution group will eventually reach the same state, which is necessary to guarantee that a client can get enough matching replies to complete a weakly consistent read request. However, this is not the case if the communication is interrupted. In that case, each execution replica remains in the state it had when the communication was interrupted. Note that liveness in general only guarantees progress during synchronous phases.

In order to ensure that clients receive enough matching replies to still complete weakly consistent read requests even if the wide-area communication is interrupted, all execution replicas have to execute the same set of write requests to reach the same state. To guarantee this, we extend the receiver endpoints for the commit channel as follows. The endpoints regularly exchange status messages about the highest position in each subchannel up to which they have received valid certificates for this and all earlier slots. These status messages are constructed using acknowledgement outboxes and are exchanged between replicas of the same group. Another receiver endpoint that has already received certificates for a newer slot can then forward the missing certificates. As the certificates are signed, the receiver is able to verify them and therefore the certificate can be shared between endpoints. To avoid duplicate message transfers, an endpoint will wait

for a short timeout before forwarding the certificate. If the other endpoint has received the certificate in the meantime and reports so in a status message, then no forwarding takes place. In total, a receiver endpoint will eventually learn about certificates for all slots delivered by any other correct receiver endpoint.

5.8. Evaluation

In the following we evaluate the performance of SPIDER's architecture against that of two other protocols. The experiments are conducted in parts on Amazon EC2 and in parts using an emulated cloud testbed. We start by analyzing the response times for write and read requests, followed by two microbenchmarks that focus on IRMCs. The section concludes with an analysis of SPIDER's adaptability to workload changes and an analysis of two optimizations for SPIDER and the IRMCs.

5.8.1. Setup

For the evaluation we compare three protocols representing different system architectures. Firstly, **BFT** represents the PBFT [57] protocol, which serves as a representative of classical BFT protocols and is configured to use MACs for its protocol messages for optimal performance. Secondly, **HFT** is an implementation of the hierarchical fault-tolerant protocol Steward [24] and like it consists of two layers of protocols to coordinate replicas within and across sites. And finally SPIDER is our protocol, which has been described in the previous sections. It is configured to use PBFT with MACs to order requests at the central agreement group. SPIDER uses the IRMC-SC variant together with the signature sharing optimization described in Section 5.7.1.

Replicas

Unless stated otherwise the replicas are distributed world-wide and run in the Amazon EC2 regions in Northern Virginia, Oregon, Ireland and Tokyo. Each replica of a group in a region is located in a different availability zone. In general, the systems are configured to make use of all four regions and to tolerate a single fault $f = 1$. SPIDER requires four replicas for its agreement group, which is hosted in Northern Virginia, and an execution group consisting of three replicas in each region, resulting in a total of 16 replicas. Note that Northern Virginia contains both an execution group and the agreement group. For HFT, each site consists of four replicas and the sites are mapped to regions, which also results in a total of 16 replicas. As HFT requires four replicas at each site, whereas most regions only provide three availability zones, we have placed the replicas such that one availability zone contains two replicas. The setup for BFT consists of only a single replica in each region. Using more replicas to tolerate more faults would increase the amount of communication necessary between replicas and generally results in a lower throughput [1, 121]. The agreement protocols are configured to build batches containing up to 10 requests. The batching mechanism assembles a batch once it is full or after 2 ms depending on what happens first.

All protocols are implemented as part of a single codebase written in Java to allow for better comparability between protocols. The implementation uses HMACs [133] with SHA256 [164] to authenticate messages, 1024-bit RSA PKCS1 signatures [163], to which we will also refer as RSA-1024, for signing and for HFT the threshold signature scheme described by Shoup [182] also using 1024-bit keys.

We use small virtual machines of type t3.small (2 vCPU, 2 GiB RAM) running Ubuntu 18.04.4 LTS and OpenJDK 11 to host our replicas. The, by default too small, TCP buffer sizes in the Linux kernel are tuned to allow for maximum throughput instead of being limited by the round-trip time between datacenters as shown by Lai et al. [136]. The buffers are adjusted to be large enough that the receiving side of the connection can acknowledge the transmitted data before the buffer runs out of space.

Later experiments will use a local testbed that is described in Section 5.8.6.

Clients

The clients in a region run in a single separate virtual machine and by default send their requests to replicas in the same region. We run 50 client instances at each location running in an open loop such that the clients in a region submit up to 100 requests per second. A client only issues a new request after receiving a response to a previous request, however, the number of clients is high enough to ensure open loop behavior. The request rate is limited via a token bucket to which tokens are added every 10 ms and whose capacity is sufficient to allow bursts equivalent to sending half a second worth of requests at the normal request rate. We use rate-limited clients, as otherwise the different response times for clients in different regions result in a highly skewed workload between regions. For example, for Spider Northern Virginia would contribute the vast majority of requests.

The clients issue read or write requests for random entries in a key-value store and either retrieve or set a 200 byte payload, similar to the request size used for Steward [24]. All client requests are signed to prevent tampering and to ensure that all replicas agree on the validity of a request.

Experiments

Each measurement runs for 180 seconds of which a warm-up period of 50 seconds and a shutdown time of 10 seconds is cut of. We record the timings of each individual request and use these as the basis to calculate averages and percentiles.

We want to answer the following questions in our evaluation:

1. How does the performance for read and write requests of Spider compare to that of the other systems?

2. What are the costs of modularity in Spider?

3. What are the performance characteristics of the IRMC variants?

4. Which benefits does adaptability offer?

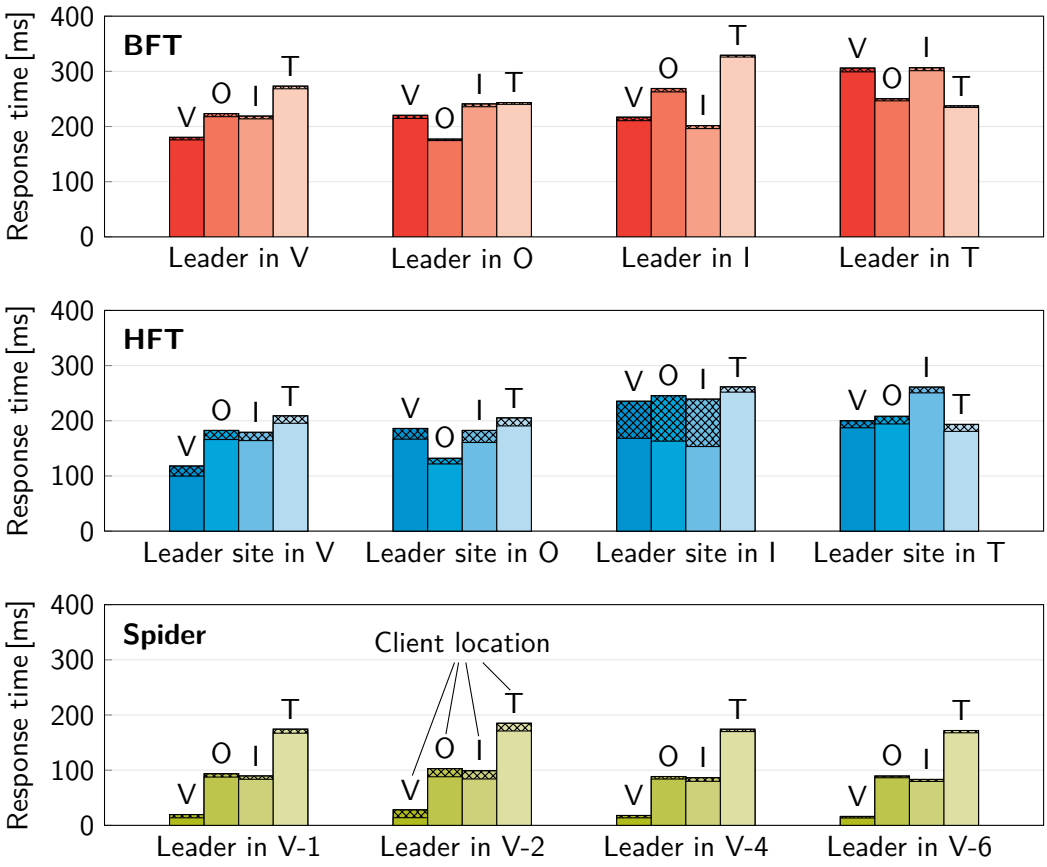


Figure 5.18: Median □ and 90th percentile ▩ of the response times for write requests depending on the client and leader location. The locations are Northern Virginia (V), Oregon (O), Ireland (I) and Tokyo (T). For SPIDER the suffix after the region name refers to the availability zone ID which is identical across AWS accounts [14]. For example, V-1 maps to the availability zone with ID 1 in Northern Virginia.

5.8.2. Latency

To answer the first question, we analyze the response times in wide-area networks of each protocol. For this we separately measure write and read requests.

5.8.2.1. Write Requests

In our first experiment, we measure the response time for clients issuing write requests to the system. We evaluate different leader locations, for BFT and HFT the leader is placed into different regions, whereas for SPIDER the leader moves within the region of the agreement group. For each system we report the median and 90th percentile depending on the leader location and group the results by client location.

Based on the results shown in Figure 5.18 we make the following observations: Firstly, the response time for a client strongly depends on the client's geographic location. For example, with the leader replica located in Northern Virginia, the response times are much lower for clients in Northern Virginia than for those in Tokyo. This is a result of the client being located near the protocol's leader replica or in case of SPIDER the agreement group.

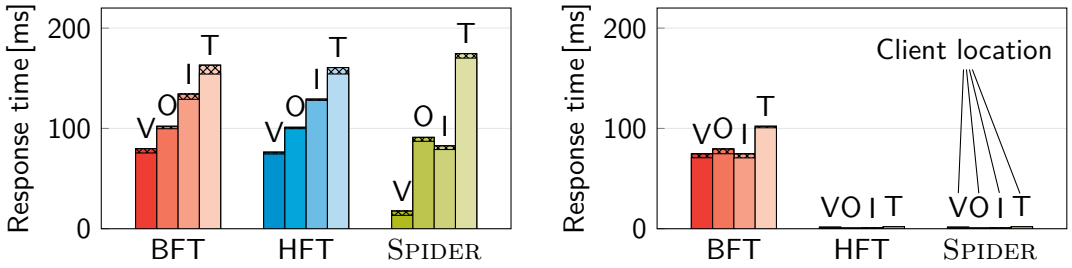
Secondly, the response times differ strongly between the different protocols. For the best leader location the median response time, for example, for a client in Northern Virginia ranges between 176 ms using BFT, 100 ms for HFT and 13 ms for SPIDER. That is, SPIDER provides 87% and 92% lower responses times for clients in Northern Virginia, respectively. Even for less favorable client locations, SPIDER offers lower response times of only 167 ms for clients in Tokyo compared to 181 ms using HFT and 234 ms using BFT. Similarly, SPIDER also provides the lowest response times for all other client locations.

The performance difference is a result of the communication patterns used by the protocols. BFT and HFT use a wide-area communication pattern that first has to forward a client request to the current leader, which then has to contact at least a majority of regions before a client can get a reply. In contrast, in SPIDER only a single wide-area roundtrip from the client's execution group to the central agreement group is necessary to process a write request. This also explains the especially low response times of the execution group in Northern Virginia, which only has to communicate locally with the agreement group. The execution groups at other locations update their state concurrently but are not involved in providing a reply to the clients in Northern Virginia.

The response times in BFT and HFT are also highly dependent on the current location of the leader replica or site. For example, for a client located in Ireland the median response times can increase by up to 53% for BFT and 64% for HFT if the location of the leader changes from Ireland to Tokyo. In general, the response times are lowest for clients in the region of the current leader. This can lead to significant performance variability if the location of the leader replica changes. SPIDER on the other hand provides stable response times with only a small variability of a few milliseconds depending on the current leader location, as its leader replica in the agreement group only moves between the availability zones of a single region.

5.8.2.2. Read Requests

Our next experiment measures the response times for strongly and weakly consistent read requests. We start with the results for clients issuing strongly consistent read requests, shown in Figure 5.19a, which presents the response times for each protocol depending on the client location. There is no distinction by leader location as BFT and HFT directly query all replicas and SPIDER provides stable response times across leader locations. SPIDER offers lower median (and 90th percentile) response times than BFT and HFT for all client locations except for Tokyo. There the response time for clients is about 10% higher for SPIDER than with BFT and HFT.



(a) Strongly consistent read requests

(b) Weakly consistent read requests

Figure 5.19: Median ☐ and 90th percentile ▨ of the response times for read requests with strong or weak consistency depending on the client and leader location.

Clients using the read optimization in BFT and HFT[2] can directly query replicas from a (Byzantine) majority of regions, which explains the response times. For SPIDER in comparison the strongly consistent read request is processed similarly to a write request and has to pass through the agreement, which leads to response times similar to those of write request. The response time for a client in Tokyo is determined by the latency to Northern Virginia for all protocols. BFT and HFT use replies from Tokyo, Oregon, and Northern Virginia, which latency-wise is the farthest away region. A client in SPIDER has to wait until the request is processed by the agreement group in Northern Virginia resulting in the 16 ms higher response time. For all other client locations SPIDER nevertheless provides the lowest response times.

Note that strongly consistent read requests in SPIDER always succeed whereas for BFT and HFT it may become necessary to retry the request when too many replicas return diverging replies.

To answer weakly consistent read requests, as presented in Figure 5.19b, HFT and SPIDER only require less than 2 ms. This low response time is possible as both protocols can answer the request locally, without requiring any wide-area communication. BFT on the other hand takes 71 to 101 ms to answer weakly consistent read requests as a client still has to query $f + 1$ replicas of which f are located in a different region.

5.8.3. Tolerating More Faults

For the following experiment, we investigate the effects of a higher fault tolerance level. We configure the systems to tolerate $f = 2$ faults. This applies to both agreement and execution groups of SPIDER, that is $f_a = 2$ and $f_e = 2$. The additional replicas for each region are located in nearby EC2 regions in Ohio, California, London and Seoul. This ensures that even if a complete availability zone becomes unavailable, the systems are still able to tolerate another fault. As BFT only requires $3f + 1 = 7$ replicas, it does not

[2]The evaluation follows the description for strongly consistent read requests in Steward [24], which claims it is sufficient to query replicas at a majority of sites. However, as described in Section 5.1.2 a correct result actually requires querying each site. Therefore, a correct implementation would incur higher response times than those reported here.

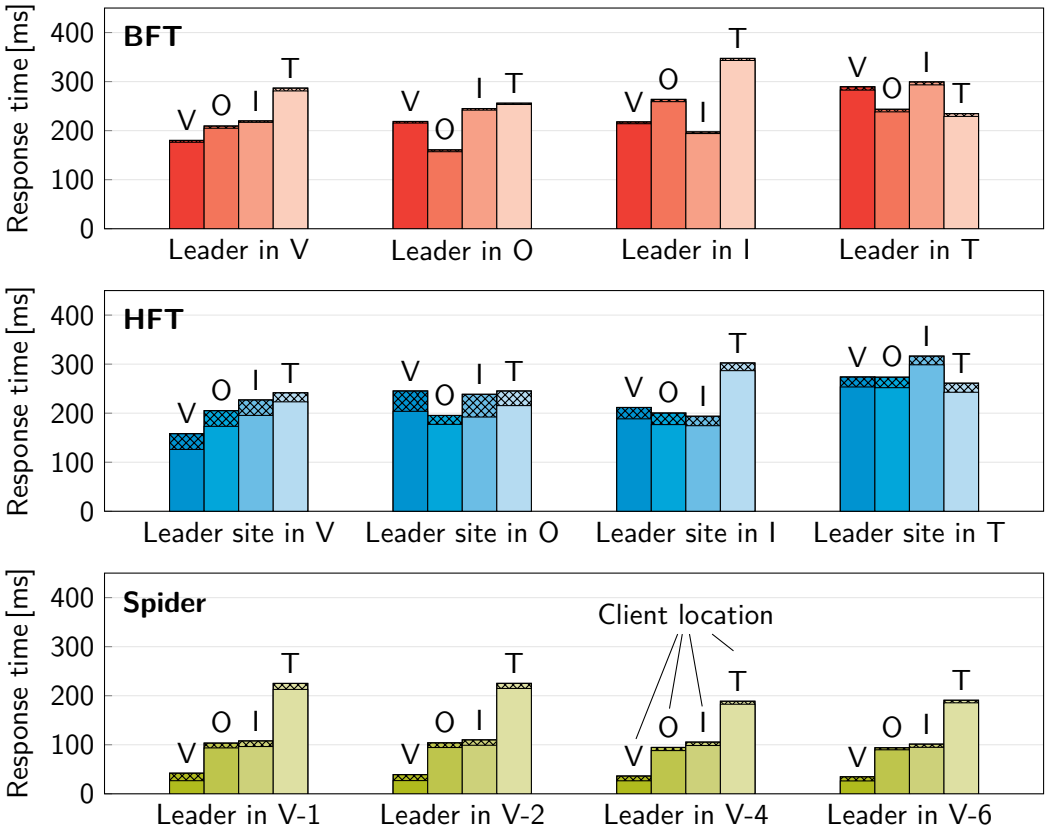


Figure 5.20: Median ☐ and 90th percentile ▩ of the response times for write requests with $f = 2$ depending on the client and leader location.

use an additional replica in Seoul. Otherwise, the experiment matches the write request setting from Section 5.8.2.1. Figure 5.20 shows the resulting response times.

The response times for BFT improve slightly for clients in Oregon or when using the leader in Tokyo. As the additional replicas for BFT are located near Ireland, Northern Virginia and Oregon, these replicas add further possibilities to gather messages from a Byzantine majority quorum required to proceed through the protocol.

The other configurations either achieve the same or higher response times than before. For HFT the response time increases in every configuration, this is especially clear when the leader site is located in Tokyo, where clients in different regions see a response time increase by 48 to 66 ms. For SPIDER, the median response time generally increases by 3 to 19 ms, except for two outliers where the response time for two of the leader replicas increase by up to 46 ms for clients in Tokyo.

The results can be explained by the higher communication latency within a group, which now has to communicate with replicas in the nearby region to order requests. In HFT this applies to every site, whereas for SPIDER with its smaller execution groups, a client can still receive enough replies just from the execution replicas in its region.

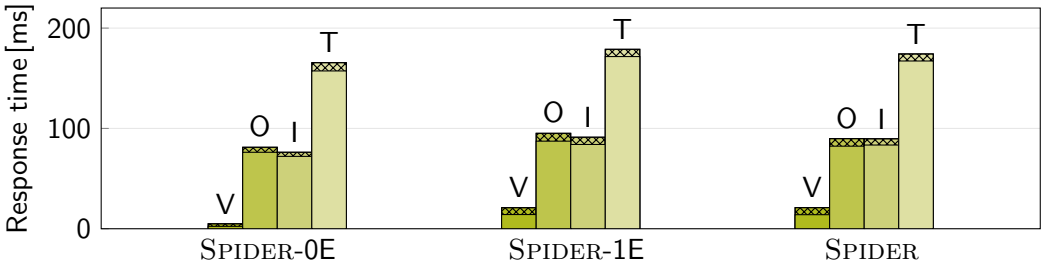


Figure 5.21: Response time comparison of SPIDER running the execution directly at the agreement (SPIDER-0E), using a single central execution group (SPIDER-1E) and the standard SPIDER setup.

Despite the response time increase, SPIDER continues to offer lower response times at each client location than both BFT and HFT.

5.8.4. Microbenchmarks

In the following we address our second and third question by first investigating the cost of modularity and then the performance characteristics of IRMCs.

5.8.4.1. Modularity

Our next experiment analyzes the response time costs associated with modularizing SPIDER into agreement and execution groups and coupling them loosely using IRMCs. For this we compare the following three variants of SPIDER:

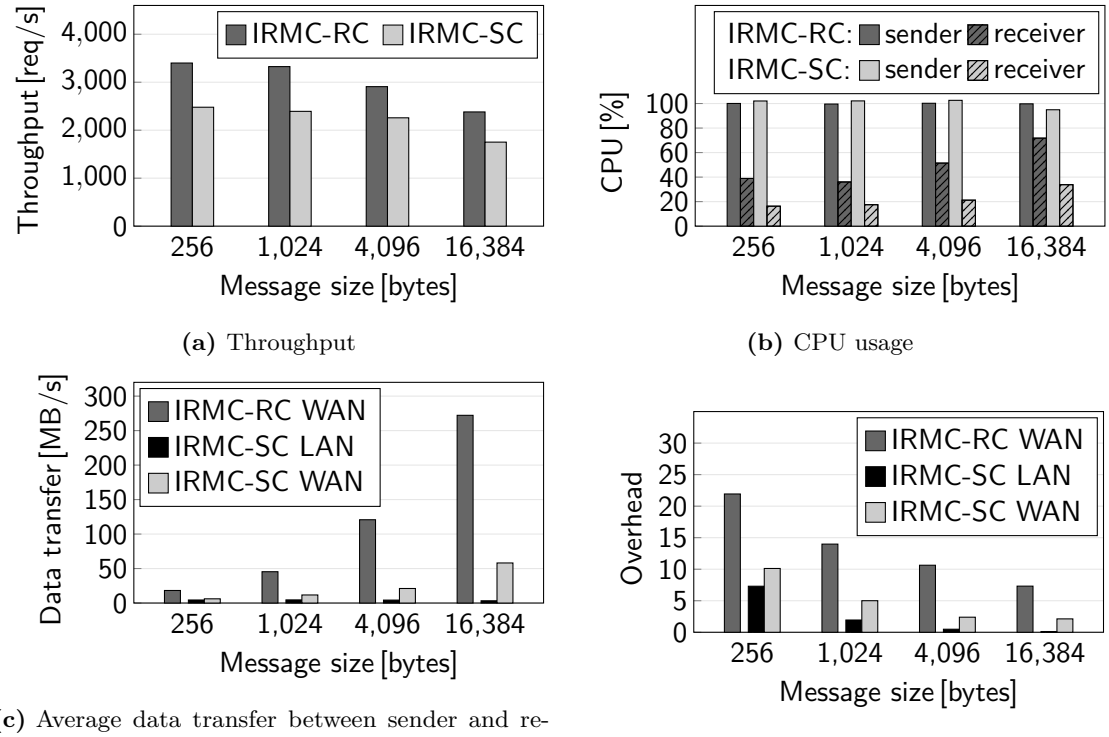
(a) A variant called SPIDER-0E that directly runs the execution within the agreement replicas and thus does not use IRMCs and execution groups.

(b) A variant called SPIDER-1E that uses a single central execution group located in the region of the agreement group.

(c) The standard SPIDER setup.

The workload is otherwise identical to the write requests experiment in Section 5.8.2.1 using the leader replica in V-4. SPIDER-0E shows the minimal costs for processing requests, whereas SPIDER-1E measures the cost of adding an IRMC that does not require wide-area communication.

The results in Figure 5.21 show the same structure of response times dependent on the latency between a client's and the agreement group's location. The response time provided by SPIDER-0E is about 11 to 14 ms lower than for SPIDER-1E, which can be attributed to the small overhead introduced by using IRMCs to transmit requests between the execution and agreement group. SPIDER-1E and SPIDER on the other hand offer roughly equal response times. However, when using only a central execution group as in the SPIDER-1E variant, then it is not possible to benefit from low response times for weakly consistent read requests for which a nearby execution group is necessary.



(a) Throughput

(b) CPU usage

(c) Average data transfer between sender and receiver group

(d) Ratio between bytes sent to message size

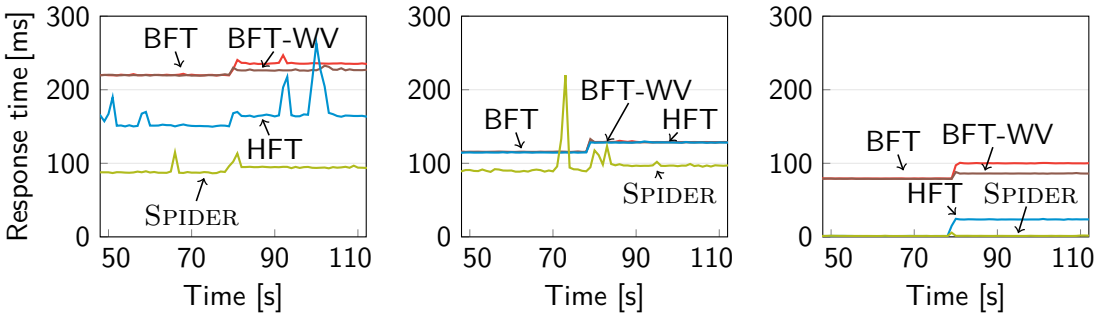
Figure 5.22: CPU, throughput and network usage of the IRMC variants.

5.8.4.2. IRMC Implementations

The following experiment focuses on the IRMC abstraction to analyze its throughput, CPU usage and amount of transferred data depending on the message size and variant. We use a configuration corresponding to that of a commit channel in Spider, that is, a group of four replicas in Northern Virginia that resemble an agreement group sends requests to a group of three replicas located in Tokyo that represent an execution group. The agreement group sends a continuous stream of messages containing a payload of a specific size to the execution group. The measurements evaluate both the IRMC-RC and IRMC-SC variant.

The results are presented in Figures 5.22a to 5.22d. The CPU usage on the sender side is shown for the first replica, which is used as sender by IRMC-SC, and on the receiver side we show the replica that has processed the largest number of messages. This ensures that we do not report values for a replica that has fallen behind. The data transfer figure shows the aggregate amount of transmitted data. For the IRMC-SC variant, local and wide-area traffic are reported separately.

For small message sizes, both IRMC-RC and IRMC-SC are CPU-bound at the sender side while sending up to 3,400 and 2,479 requests per second, respectively. The lower throughput of the IRMC-SC is caused by the additional signature verifications necessary at the sender side to assemble the Certificate message. On the other hand, on the



(a) Write requests

(b) Read requests with strong consistency

(c) Read requests with weak consistency

Figure 5.23: Average response time when adding a new region after 80 seconds.

receiver side the IRMC-RC requires more CPU than the IRMC-SC, the most probable explanation is that this is a result of the higher number of received messages when using the IRMC-RC variant.

The wide-area network traffic required for IRMC-SC is lower than for IRMC-RC, for example, for 256 byte messages by a factor of 3.0. For IRMC-SC only one of the four sender replicas forwards requests to the receivers, such that the traffic could reduce by up to a factor four. However, the actual traffic reduction is lower, as the sender replica forwards CERTIFICATE messages, which include $f_a + 1$ signatures and thus are larger than the individual SEND messages used by IRMC-RC, which only contain a single signature.

For requests with 4,096 bytes or more, the IRMC-SC sends 5.7 times fewer data than the IRMC-RC. This higher than expected traffic reduction can be explained by looking at the transmission overhead, which falls to 10.6 and 2.4 for IRMC-RC and IRMC-SC, respectively. As the IRMC only has to wait until $f_e + 1 = 2$ receivers allow the garbage collection of old messages, this can cause up to f_e replicas to fall behind and therefore lead to a lower overhead than when transmitting to all receivers.

In order to reduce the message transmission overhead, it is preferable to forward fewer, large messages with a size of at least 1 KiB instead of many small messages. By using batching at the agreement group, SPIDER can combine multiple client requests into messages with a size of a few kilobytes and thereby distribute requests to execution groups with a moderate traffic overhead and thus reduce the costs for traffic.

5.8.5. Adaptability

To answer the fourth question, we now analyze the effect of adding a new client location while the system is running. For this we start with our usual setting and add a new region in São Paulo with 50 clients that submit requests and join after 80 seconds. For SPIDER, which is able to adapt its configuration at runtime, we also start a new execution group in São Paulo.

With five regions, it becomes possible to apply the weighted-voting approach used by WHEAT [189], which we use to create a BFT variant called BFT-WV. A subgroup of

well-connected replicas is assigned a higher weight, which allows the agreement to still work with a quorum size of three if enough replicas of that subgroup are included and can lead to reduced response times. We assign the higher weight to the replicas in Northern Virginia and Oregon, which provide the lowest average response times. AWARE [43], which enhances WHEAT with an automatic selection of the best system configuration, would achieve similar results, as we have manually determined the optimal configuration.

The Figures 5.23a to 5.23c show the influence of starting clients in São Paulo on the average response time across all clients for write requests and strongly consistent as well as weakly consistent read requests. For clarity, for each protocol the graphs only show the results for the leader location providing the lowest average response times.

We observe that the response times for writes and strongly consistent reads increase for each system, once the new clients in São Paulo join. For example, for BFT these clients experience a response time of 298 ms and 124 ms for SPIDER, which results in an increase of the average response times. The benefit of BFT-WV is limited to clients in São Paulo that issue write and weakly consistent read requests. There the weighted voting approach allows reaching a quorum using replicas in Northern Virginia, Oregon and São Paulo without having to wait for further regions. Clients in other locations do not benefit significantly from the BFT-WV variant, which shows that the effectiveness provided by weighted voting depends a lot on the overall system structure.

For weakly consistent read requests, shown in Figure 5.23c, the response times for all protocols except for SPIDER increase. This is especially pronounced for HFT where the new clients in São Paulo cannot benefit from low latency for their weakly consistent read requests. HFT to our knowledge does not support reconfiguration and thus is not able to start a new region at runtime to adapt to changes in the workload. SPIDER on the other hand can activate a new region, which allows it to continue responding to weakly consistent read requests with low latency.

5.8.6. Emulated Cloud Testbed

We conduct additional experiments to assess the optimizations that reduce the number of signature computations required by SPIDER at the agreement group and to forward messages over IRMCs in regard to the questions one to three. More specifically, we analyze the effects of the signature batching optimization described in Section 5.7.2 and the client request verification offloading described in Section 5.7.3.

Testbed Setup

For the following experiments, we use a local testbed in order to allow us to conduct more extensive experiments with high message throughput. The testbed resembles the cloud setup used in the previous experiments and consists of four servers, which each emulate an Amazon EC2 region including the latency between the regions. Each server runs all clients and replicas belonging to its region. The servers run Ubuntu 20.04.4 LTS using OpenJDK 11, have an Intel Xeon E3-1275 v5 CPU with 4 physical cores with two hyper-threads each and 16 GiB RAM. The servers use netem [117, 144] provided by the Linux kernel to emulate the wide-area latencies between the cloud regions by delaying

	Northern Virginia	Oregon	Ireland	Tokyo
Northern Virginia	0.74			
Oregon	66.93	0.61		
Ireland	66.48	122.38	0.51	
Tokyo	143.76	96.85	200.08	1.40

Table 5.1: Average ping times in milliseconds between replicas in the used EC2 regions measured on February 27, 2022.

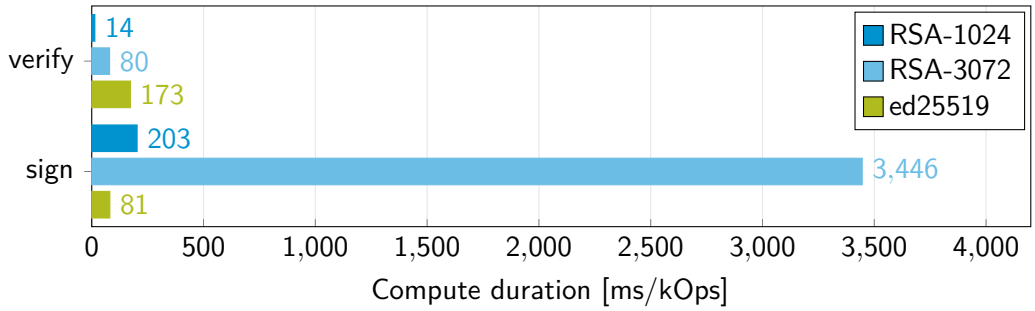


Figure 5.24: Time to run 1,000 sign or verify operations.

packets sent via Ethernet. We add a one-way communication latency between regions that is set to half the ping times in Table 5.1, based on the assumption that communication times between a pair of regions are symmetrical. For communication within a region, we inject one-way delays of 0.2 ms, which results in ping times of approximately 0.56 ms. The servers are attached to the same switch via 1 Gbit Ethernet interfaces.

System Configuration

The following experiments use an enhanced version of the prototype that includes the additional protocol optimizations as well as improvements to efficiently handle a larger number of clients. To account for the higher expected throughput, we instantiate 2,000 clients at each location, yielding a total of 8,000 clients, which is enough to saturate the system. In addition, we increase the maximum agreement batch size to 128. The large batch size amortizes the overhead of computing the expensive threshold signatures in HFT and signatures in SPIDER over a larger number of messages. For HFT this has a similar effect as using a Merkle tree as described by Amir et al. [20] to spread the cost for the threshold signature computation, but is much simpler to implement.

5.8.6.1. Stronger Cryptography

Looking forward to 2030 and beyond, the RSA key size should increase to at least 3072 bits to remain secure [37]. However, this has a huge downside: the signature size grows to 384 bytes thereby drastically increasing the overhead for each signed message. For our requests with a 200 byte payload, this results in a signature that is larger than the

payload itself. In addition, the sign and verify operations for larger keys are much slower. Figure 5.24 shows the average time to sign and verify one thousand messages with empty payload in our testbed. When switching from RSA-1024 to RSA-3072 the results show a slowdown of a factor 17 for signing and 5.7 for verification. Creating a single RSA-3072 signature will then on average take 3,400 µs and 80 µs to verify it.

As a modern alternative, we use the elliptic curve based algorithm ed25519 [139] implemented in pure Java[3]. It provides a similar security level as RSA-3072, but offers signatures that are only 64 bytes in size. Compared to RSA, the algorithm has different characteristics. As shown in Figure 5.24, verifying a signature in ed25519 takes 173 µs, whereas creating a signature only takes 81 µs, such that verifying is roughly twice as slow as signing. This is in contrast to the ratio between signing and verifying for RSA-1024, which takes 203 µs to create a signature compared to 14 µs to verify it, resulting in 14.5 times lower costs for verification.

To analyze the effects of the algorithm's different characteristics on the performance, we will use both RSA-1024 and ed25519 for our following experiments.

5.8.6.2. Write Throughput

In this experiment, we compare the throughput and response time for BFT, HFT and Spider. For Spider, we use a configuration corresponding to the previous experiments as baseline. We measure the effect of adding the IRMC signature batching optimization, the client request verification offloading and both optimizations together.

The experiment focuses on write requests, as read requests either only require MACs for authentication and thus the optimizations are not relevant for them, or in case of the strongly consistent read requests in Spider are expected to behave similarly to write requests as shown in Section 5.8.2.2.

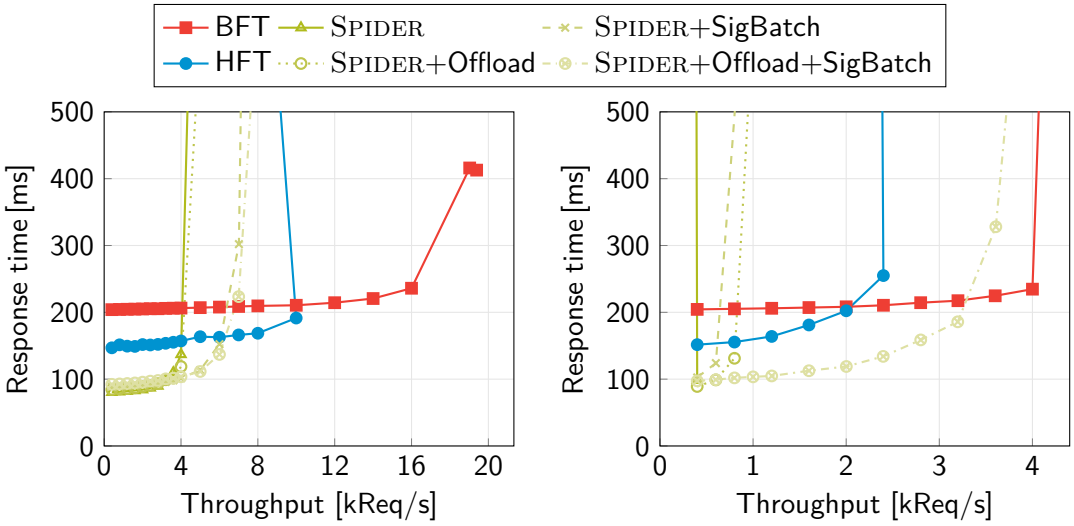
For the individual measurements, we gradually increase the client request rate in steps of a few hundred to a few thousand requests until the system is saturated. We report as maximum throughput the highest throughput up to and including the point at which a system is no longer able to achieve the requested throughput. The results of the measurements are shown in Figure 5.25a. The graph shows the average response time for requests from all regions at a certain throughput. The goal is to reach a high throughput while maintaining low response times.

For a throughput of 400 requests per second, the response time is slightly lower than that presented in earlier experiments. We assume that this is a result of the network latencies between regions being slightly lower than during the previous measurements and, in addition, our local testbed emulates slightly lower latencies between some availability zones within a region.

Throughput using RSA-1024

We first analyze the results for RSA-1024. BFT, HFT and Spider with both optimizations achieve a maximum throughput of 19.0, 10.0 and 8.4 thousand requests per second, respectively. BFT only has to verify client request signatures, whereas the agreement

[3] https://github.com/str4d/ed25519-java/releases/tag/v0.3.0



(a) Using RSA with 1024-bit keys

(b) Using ed25519

Figure 5.25: Average response time and throughput under increasing request loads. *Offload* refers to client request verification offloading and *SigBatch* to the IRMC signature batching optimization.

process only relies on (much cheaper) MACs. Thus, the replicas only have to verify signatures, which for RSA is much faster than creating signatures. In contrast, HFT and SPIDER also have to create and verify signatures to process requests. However, they still achieve a throughput of 53% and 44%, respectively, of that provided by BFT, showing that both systems are able to amortize the costs of the signature computations over multiple of requests.

SPIDER without optimizations, with the offload optimization, with signature batching and both optimizations achieves a maximum throughput of 5.1, 6.0, 7.4 and 8.4 thousand requests per second, respectively. As shown in Section 5.8.6.1, verifying signatures is much cheaper than creating them when using RSA. Thus, the offload optimization only offloads the relatively cheap signature verification for client requests from the agreement group, which explains the limited increase in throughput. The request channel normally has to verify $f_e + 1 = 2$ signatures when receiving a request from the execution groups. This overhead is reduced by the IRMC signature batching optimization which amortizes the cost over many requests.

Response Time using RSA-1024

The response time of each system grows slowly until the throughput nears saturation, at which point it drastically increases. The measurements show that SPIDER maintains its lower response times than BFT and HFT until the system becomes saturated. Compared to the base variant of SPIDER, each optimization adds a few milliseconds of latency, either due to the additional protocol phase for the offload optimization or the IRMC batching timeout of 10 ms used to batch signature calculations.

Throughput and Response Time using ed25519

Using ed25519, HFT reaches a maximum throughput of 2.4 thousand requests per second, compared to the throughput of 4.4 thousand requests per second for BFT. The higher costs to verify the signature of a client request now drastically reduce the overall throughput compared to using RSA-1024. Note that HFT benefits from still using RSA-based threshold signatures, which only offer a lower security level than ed25519.

Without optimizations, SPIDER achieves a throughput of only up to 0.4 thousand requests per second. Just using IRMC signature batching increases the throughput to 0.8 thousand requests per second, resulting in an increase by a factor 2. The client signature verification offloading alone is more effective and increases the throughput to 1.1 thousand requests per second. When combining both optimizations the throughput increases nearly five-fold to 5.4 thousand requests per second. As shown in our next experiment, the signature batching optimization allows forwarding more than 20 thousand requests per second via an IRMC, however, the overall throughput is still limited as the agreement group also has to verify the signature of each request. By offloading the signature verification to the execution groups, the agreement group is no longer the computational bottleneck. The high throughput comes with a high response time of 1.25 seconds, which is a result of saturating the CPU of the server hosting the agreement group, which also hosts an execution group and the clients of the region. Nevertheless, SPIDER is able to outperform BFT in both response time and throughput for most throughput levels.

By combining both optimizations SPIDER reaches 64% of its throughput when using RSA-1024, but also offers a much higher security level.

5.8.6.3. IRMC Microbenchmark

In our last experiment, we analyze the IRMC variants and the signature batching optimization in isolation. For this, we run the microbenchmarks from Section 5.8.4.2 on our local testbed. As the batching optimization primarily focuses on reducing the computational overhead, we only analyze the smaller message sizes of 256 and 1,024 bytes as larger message sizes already saturate the network of the testbed without optimizations.

The results are presented in Figure 5.26. We first check whether the Amazon EC2 and the testbed achieve similar results. For this we compare the unoptimized variant using RSA-1024 to the measurements run on Amazon EC2, which shows an up to 28% higher throughput for the IRMC-RC and up to 37% for the IRMC-SC on the testbed. The CPU usage and network overhead remain in a similar range. That is, the testbed produces results which are roughly comparable to those using Amazon EC2.

Performance using RSA-1024

For both message sizes, using the unoptimized IRMC variants, the sender side saturates its CPU, which limits the throughput. The signature-batching optimization increases the throughput by up to a factor 5.1 for the IRMC-RC variant and by up to a factor 8.5 for the IRMC-SC variant depending on the message size. The CPU usage on the sender side falls to 13% using an IRMC-RC and to 83% using an IRMC-SC such that the

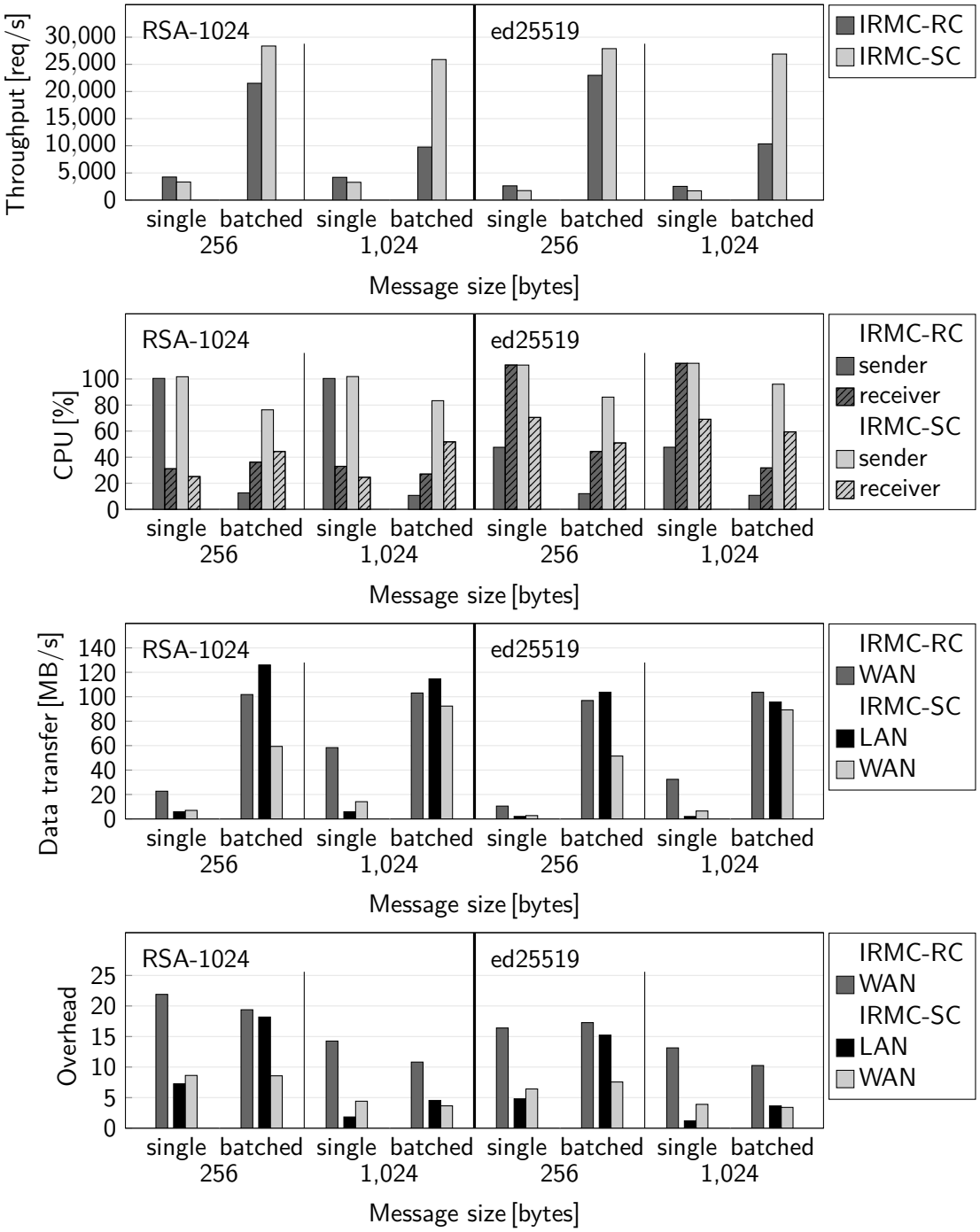


Figure 5.26: Throughput, CPU, network usage and network overhead of the IRMC variants. *single* refers to the unoptimized variant, which computes signatures for each message, whereas *batched* uses signature batching.

benchmark is no longer CPU bounded. The IRMC-SC has a much higher CPU usage on the sender side as it has to exchange and verify signatures while assembling certificates.

Using signature batching, the IRMC-SC now achieves a higher throughput than the IRMC-RC variant. As neither variant is CPU bounded by the signature verifications, this throughput is a result of the IRMC-RC variant incurring a higher data transmission overhead that has to be transmitted between regions. The throughput of the IRMC-RC for 1,024 byte messages is limited by the network, which according to `iperf3` can transmit at most 111 MiB/s in total between regions in our testbed.

Performance using ed25519

We now analyze the influence of using ed25519 signatures on the IRMC performance. The throughput of the unoptimized IRMC-RC is lower at about 2.6 thousand requests per second and is 48% to 50% higher than that of the unoptimized IRMC-SC, resulting in a larger performance difference than when using RSA signatures. In addition, the CPU usage at the receiver side of an IRMC-RC drastically increases and becomes the bottleneck. This is a result of ed25519 signatures being computationally more expensive to verify than to create. For the IRMC-SC, the CPU usage at the receiver side increases to around 70%. As the sender side also has to verify signatures while assembling a CERTIFICATE message, it reaches its CPU limit before the receiver side.

Using signature batching, the throughput increases by up to a factor 8.8 for the IRMC-RC and a factor 16.0 for the IRMC-SC. The CPU usage is no longer the bottleneck, such that the throughput reaches a similar level as the RSA-1024 variant.

Compared to RSA-1024 without batching, the overhead for 256 byte requests shrinks by approximately 25% when using ed25519 signatures as a result of the smaller signature size. The usage of the signature batching optimization increases the overhead again, which is especially visible for the IRMC-SC variant. There the local traffic overhead using RSA-1024 and ed25519 increases by roughly a factor 2.5 and 3.2, respectively. The reason for this is that the locally exchanged messages primarily consist of the signature, whose size grows as tree signatures contain the path through the Merkle tree in addition to the 64 bytes for the ed25519 signature itself. At the used batch limit of 128, this results in adding up to seven SHA256 hashes, that are equivalent to 224 bytes.

The messages sent over the wide-area network also include the actual requests such that the larger tree signatures only lead to a smaller relative increase of the message size. In fact, when comparing the wide-area traffic between the unoptimized and the signature batching variants, contrary to the previous explanations the overhead appears to shrink in most cases. However, this is a result of one of the receiver replicas falling behind such that it no longer receives messages, which results in a lower than expected overhead.

Transmitting the whole Merkle tree only once and referring to it later on would remove most of this overhead in addition to only require transmitting the signature once. We leave optimizing the transmission overhead of the tree signatures as future work. Nevertheless, signature batching provides a large increase in throughput with potential for further improvements by reducing the transmission overhead.

5.9. Related Work

In the following we discuss related works in relation to different aspects of SPIDER.

Replica Discovery

In systems which support reconfiguration, clients need a way to discover the currently active replicas to take part in a system. A common way is to enable the clients to query all known replicas for the current configuration that was generated and signed by a trusted administrator client [47, 149]. As a fallback, in case the client does not know any of the currently active replicas, a name service such as a secure variant of the domain name system (DNS) [34, 53] can be used to discover replicas that can be queried for the current configuration [146, 149]. As SPIDER only reconfigures its execution replicas, it uses the agreement group as a central configuration repository to provide the clients with the current configuration.

Reconfiguration

In addition to the reconfiguration mechanism that is provided by SPIDER, orthogonal approaches can be used to reconfigure the agreement protocol itself. Abstract [32] and ADAPT [36] allow composing different protocols by providing a mechanism to abort a protocol execution and transfer its state to a successor. Carvalho et al. [55] investigated the performance of different reconfiguration mechanisms using BFT-SMaRt [47]. These approaches could be used to reconfigure the agreement protocol used by SPIDER or even to change the composition of the agreement group.

ReBFT [73] and CheapBFT [126] allow switching between a resource-efficient mode for fault-free intervals in which only $2f + 1$ replicas actively process request to save resources and a fallback mode in which all replicas are active. ZZ [209] modifies the request execution to use only $2f + 1$ replicas of which f are paused during normal, fault-free executions. In case of disagreement on the execution result, the additional replicas are woken up. A similar reduction of the resource footprint can be achieved using passive virtual machines as in SPARE [75]. These approaches are orthogonal to SPIDER and could be used to improve the resource efficiency of the execution groups by pausing f_e of the only $2f_e + 1$ replicas in an execution group during fault-free executions.

To support the administrator with modifying the execution groups, an adaption manager like FITCH [64] could be used. Reconfiguration mechanisms for systems like MongoDB [180] and ZooKeeper [183] let new replicas first retrieve the current state before adding them to the system. SPIDER provides a similar mechanism by allowing new execution groups to fall behind without affecting the rest of the system.

Adaption

Besides changing the protocols and replicas themselves, it is possible to adapt system parameters dynamically. To improve performance, the number of requests to combine into a batch and their timeout can be adjusted at runtime [177]. For systems using weighted voting, AWARE [43] measures latencies between replicas and uses those to determine optimized weights for the replicas. As shown in our evaluation in Section 5.8.5 the potential for latency improvements depends on the actual system structure. By

configuring a system to be more or less resilient, it is possible to trade resource usage with resiliency based on the current threat level [185]. These approaches are orthogonal to the reconfiguration mechanism in SPIDER. The batch size adaption could allow SPIDER to automatically select optimal batch sizes to increase the performance of the agreement group and the IRMCs.

Reducing Communication Costs

BLinks [20] offer the means to transmit an ordered sequence of requests between replica sites. A BLink requires the sender side to submit ordered requests, which makes it necessary to run an agreement protocol at each sender and adds overhead. Requests are efficiently forwarded between sites using a single wide-area message that is afterwards distributed within the receiver site. To forward client requests to the site hosting the leader replica, Amir et al. describe an additional subprotocol called CLink, which is able to forward the not yet ordered client requests. In contrast, an IRMC supports multiple subchannels which can forward client requests without requiring prior ordering or additional mechanisms. This also enables SPIDER to use fewer replicas for the execution groups than for the agreement. The forwarding approach of BLinks can serve as a blueprint for more network-efficient IRMC variants.

Other approaches to reduce the communication costs are the usage of erasure coding to reduce the amount of redundancy between transmission from different replicas [116] or separating the request ordering from fault handling by making use of an external reconfiguration service [173]. In SPIDER the agreement group could host such a service.

Geo-Replication Architecture

GeoBFT [110] evenly partitions the agreement sequence numbers in round-robin manner across different replica sites. Each site runs a full agreement protocol to locally order and certify request assignments to sequence numbers. The leader replica of each site is then responsible for distributing the agreement result to $f + 1$ replicas at each of the other sites, which further distribute the result internally. These assignments are then combined into a single total order and are executed at each site. To replace a faulty leader replica, it can become necessary to trigger a view change across the wide-area communication links. Compared to SPIDER, each site must contain $3f + 1$ replicas to run an agreement protocol instead of the $2f_e + 1$ replicas required by SPIDER for an execution group. For GeoBFT it is also necessary that every site stays available to allow the protocol to make progress, whereas SPIDER is able to tolerate individual slow execution groups. Due to the round-robin assignment of sequence numbers in GeoBFT, it must fill gaps between sequence numbers if necessary with a placeholder request such that the slowest group can determine the overall response time.

Signature Offloading

Signature verification is considered a performance bottleneck in Byzantine fault-tolerant protocols [57, 62, 130, 150]. However, not using signatures for client requests can result in complex corner cases [63] or even allow clients to trigger the replacement of a correct leader replica [56], which is the reason why SPIDER requires requests processed by the agreement group to be signed. UpRight [62] introduces a separate layer of replicas which

assemble MAC-based matrix signatures [9] to serve as a more efficient replacement for signatures. The signature verification offloading in SPIDER serves a similar purpose but minimizes the overhead by only exchanging the information that at least one correct replica considers a request to be valid, which avoids the need to distribute a matrix of authenticators. Mir-BFT [192] selects $f + 1$ replicas per request that during fault-free executions are responsible for verifying the client signatures. In SPIDER when using signature verification offloading, clients can achieve a similar result by initially sending their requests only to $f_e + 1$ replicas in the local execution group and waiting for a timeout, before sending it to the remaining execution replicas.

Client Communication

Efficiently providing a client with a reply is important for the overall performance of a protocol. Several approaches have been suggested besides the typical way of providing $f+1$ matching replies from different replicas to a client as done by PBFT [57]. SBFT [108] adds a protocol phase in which the replicas use threshold signatures to aggregate a certified reply for the client, which enables a single replica to provide a reply with a corresponding signature to the client. Troxy [141] uses replicas equipped with a trusted component that proxies the client request and enables clients to access the system without knowing details of the replication protocol. The trusted component to which a client is connected submits the request, collects and verifies replies, and provides the client with a single reply. In SPIDER clients are typically located in the same region as the execution replicas and thus allow the replicas to efficiently communicate the response to the client over low-latency links.

To prevent clients from flooding the system with requests, protocols like Aardvark [63] only accept a new client request after processing the previous one has finished. Mir-BFT [192] allows clients to issue a limited number of requests concurrently. The limit is replenished after reaching the next checkpoint. SPIDER includes a mechanism to limit the influx of client request by letting agreement replicas only read a new request from the request channel after ordering the previous one of a client.

State Synchronization

Reliably sending data queued for submission at an IRMC to the receiver side requires handling message loss. One possible solution is the use of a message store combined with periodic status messages to resend missing messages as in PBFT [57]. However, this essentially results in reimplementing the reliable transmission guarantees offered by TCP [78]. The approach used by the IRMC distinguishes between faults already handled by TCP (i.e., message loss) and those that are not (i.e., interrupted connections). Only the latter require special handling. After reestablishing the connection, the sender and receiver side essentially have to synchronize their message store. Currently, this works by sending the full state to the receiver side, similar to synchronization in state-based conflict-free replicated data types (CRDTs) [181]. However, a more efficient synchronization would also be possible by only transferring the missing delta [10].

5.10. Summary

The communication latency between the replicas running the agreement protocol contributes a significant fraction of the overall response time for processing a request. SPIDER groups its replicas into one agreement group and multiple execution groups. The replicas in each group are typically located within a region, which allows local communication with low latency, with replicas spread across multiple availability zones to reduce the risk of correlated failures. The communication between groups is managed using the IRMC abstraction, which only transfers messages supported by at least $f_s + 1$ replicas and thereby prevents faulty replicas from sending messages. This loose coupling also enables SPIDER to only use $2f_e + 1$ replicas for each execution group, thus simplifying the mapping of replicas to the three availability zones that are typically offered in a region. To process a request, a client submits it to the local execution group, which forwards the request over a client-specific subchannel to the agreement group, which in turn orders the request and distributes it via the commit channel to all execution groups. After executing the request, the client receives the result from its local execution group. Additionally, SPIDER allows clients to issue weakly consistent read requests, which can be processed locally and therefore offer low latency.

Our evaluation shows that SPIDER is able to provide significantly lower response times for write requests than PBFT and HFT for each region. For strongly and weakly consistent read requests, it also achieves similar or lower response times. When messages are authenticated using ed25519 signatures, an optimized variant of SPIDER achieves a similar maximum throughput as PBFT.

6
Concurrent Checkpointing

For applications with a large state, creating checkpoints can take a long time. It requires capturing a snapshot of the application state during which the request execution must be paused, which can result in response-time spikes. Our approach Deterministic Fuzzy Checkpointing (DFC) lets the replicas create the snapshot concurrently to processing requests, thereby drastically reducing the pause duration. However, the resulting snapshot is fuzzy. Together with additional information about the modifications made by requests executed in the meantime, the snapshot is turned into a deterministic fuzzy checkpoint that is again identical on all replicas.

Section 6.1 analyzes the shortcomings of existing approaches to create checkpoints regarding the aspects efficiency, resilience and flexibility. Afterwards Section 6.2 presents the different steps to create a deterministic fuzzy checkpoint on a conceptual level. The mapping to two concrete application interfaces is given in Section 6.3. In Section 6.4 our checkpointing approach is extended to only capture data that has changed between two checkpoints. Further optimizations are then described in Section 6.5. Section 6.6 investigates the impact on throughput and response time of DFC in comparison to traditional checkpointing methods. Finally, we discuss related work in Section 6.7 and conclude the chapter in Section 6.8.

6.1. Problem Statement

Replicas running a state-machine replication protocol have to create checkpoints in regular intervals. This allows replicas that have fallen behind to catch up by retrieving such a checkpoint and also enables the recovery [58] of replicas which after restarting the replica require the current state. In addition, checkpoints enable the replicas to garbage collect older requests to ensure that they do not run out of memory.

We start with a short repetition of the checkpointing component discussed in Section 2.3.3.5. Replication protocols typically create a new checkpoint after executing a

sequence number divisible by the checkpoint interval k. This ensures that the application has executed the same requests on all replicas. A replica captures a *snapshot* consisting of the application state and some replication-protocol–specific data, which it uses to create a checkpoint. As described in Section 2.1.6, we assume that the application state consists of several objects $\mathcal{O} = \{o_1, o_2, ..., o_n\}$, which are uniquely identified by an object identifier. The replicas then exchange signed CHECKPOINT messages containing the hash of the checkpoint. Each replica tries to collect a *checkpoint certificate* consisting of $2f + 1$ matching checkpoint messages, which requires that the replicas create identical checkpoints and which proves the checkpoint's correctness at which point the checkpoint becomes *stable*. This triggers the garbage collection of old requests and allows other replicas to retrieve and verify the checkpoint.

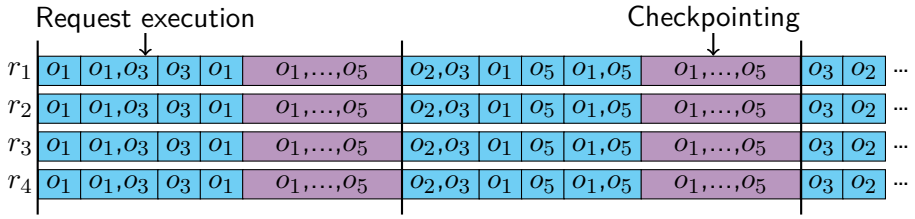
Section 6.1.1 discusses how to efficiently create checkpoints, followed by resiliency considerations in Section 6.1.2. Afterwards we present in Section 6.1.3 the challenge of providing flexibility regarding which data is captured by a checkpoint and the number of required replicas.

6.1.1. Increasing the Efficiency

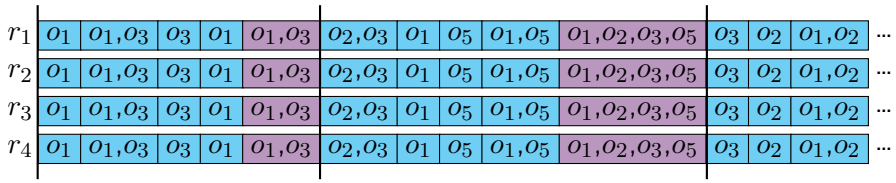
As checkpoints must be created periodically, it is important that their creation only has a small performance impact. However, for applications with several gigabytes of state or more, just capturing a snapshot of the application state can take multiple seconds [46], which causes noticeable delays even in a wide-area environment. To create a consistent snapshot that is identical on all correct replicas, during this time the request execution at the replica has to be paused. Otherwise, concurrent modifications of the application state could cause some replicas to capture a state object before a certain request was executed and others to capture it afterwards, thus collecting different states.

The most basic approach is *full checkpointing*, which creates a checkpoint by capturing a full snapshot of the application state, as shown in Figure 6.1a, that is, it contains a copy of every object [46, 62]. The replication library pauses the request execution right after executing the checkpoint sequence number and then instructs the application to export its whole state. Once the export is complete, the request execution continues. As all replicas pause at the same logical point in time and no requests are executed during the export, this results in identical snapshots on all replicas. Thus, the corresponding checkpoints are identical and can be compared across replicas. Full snapshots are relatively easy to implement, as the application just has to export its whole state. However, this comes at the price that creating a full snapshot can cause substantial delays of multiple seconds [46], which are also confirmed by our evaluation in Section 6.6.2.

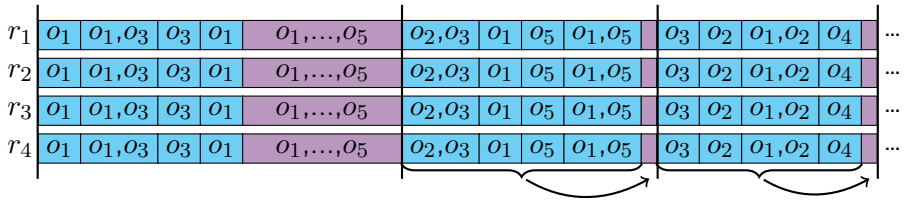
PBFT [57] uses *differential checkpointing*, shown in Figure 6.1b, which captures *differential snapshots* containing only state objects that have changed since the previous snapshot. This requires an application interface that allows the replication library to learn about changed state objects and to selectively export those. By combining those with the objects from the last checkpoint, the library can assemble an up-to-date checkpoint. In more detail, PBFT uses a copy-on-write (COW)-like approach in which the application notifies the library before modifying an object, which allows the library to back up the



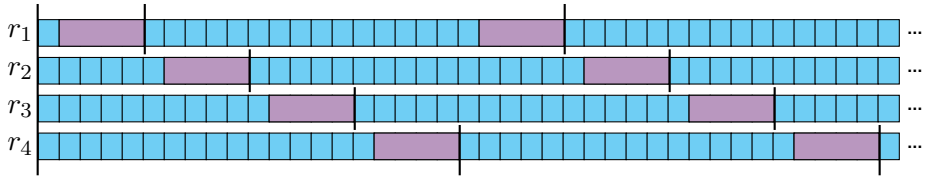
(a) *Full checkpointing* always captures a snapshot of the whole application state.



(b) *Differential checkpointing* [57, 59] only captures objects that have changed since the last checkpoint.



(c) *Hybrid checkpointing* [62] alternates between creating full checkpoints and hybrid checkpoints, which additionally contain the list of ordered requests since the last full checkpoint.



(d) Dura-SMaRt [46] sequentially schedules the creation of full checkpoints such that only up to f replicas at a time pause their request execution to create checkpoints.

Figure 6.1: Schematic overview of different existing approaches to create checkpoints.

current state of the object and store it in the latest snapshot if necessary. Retrieving an object from a snapshot then works by selecting the object copy in this snapshot, or if no such copy exists, then iteratively checking all newer snapshots and finally falling back to retrieving the object from the current application state. Taking a snapshot only requires creating a new, initially empty, snapshot and resetting the tracking of changed objects. However, to create the actual checkpoint, the library also has to calculate the current checkpoint hash. This works by computing a hash combining the hashes of all state objects. For objects which are unchanged since the last snapshot, the previously calculated hash is reused, but for changed objects the calculation requires exporting these objects. To guarantee a consistent state, this step cannot run concurrently to the request execution and thus delays the resumption of the request execution. Differential checkpointing processes fewer data than full checkpointing, however, as we show in our

evaluation in Section 6.6.3, there can still be substantial delays when large numbers of objects have changed, or the export of some objects takes a long time.

BASE [59] extends PBFT's *differential checkpointing* by distinguishing between an abstract state representation that is shared between replicas and a concrete variant at each individual replica. This allows replicas to rely on different service implementations, for example by using different databases, and thereby enables the usage of existing commercial of-the-shelf (COTS) components to achieve fault independence between replicas. However, this also requires a translation layer to convert between the abstract and concrete representation. This conversion increases the costs of exporting data from the application and thereby prolongs the pause of the request execution.

Clement et al. [62] suggest using a helper process that also processes all requests and thus can be used to capture a full application snapshot without pausing the request processing at a replica. This approach doubles the required resources to process requests and requires that the request execution in the helper process can catch up to the replicas before having to create the next checkpoint. If this cannot be guaranteed, then the request execution at a replica may still have to pause in order to let the helper process catch up. Another suggestion is to integrate COW directly into the application [62]. While this approach may be effective to reduce delays, it can require significant modifications to the application's data structures. In addition, this requires applications to implement the management of the snapshot data itself, which can no longer be offered in a generic manner by the replication library.

Approach of this Thesis

Instead of creating a snapshot after the execution processed a certain sequence number, our approach DFC starts capturing the application state a short time earlier on and does so concurrently to the request execution. Thereby, DFC does not have to pause the request execution, but this will result in a fuzzy snapshot. By additionally capturing intermediate changes to the state objects, DFC has enough information to transform the fuzzy snapshot into a regular checkpoint. This latter step also runs concurrently to the request execution to avoid delays.

6.1.2. Maintaining Resilience

Replicas have to verify the content of a checkpoint before applying it in order to guarantee that the application state has not been tampered with. This requirement is easily fulfilled for checkpoints created by full and differential checkpointing, as a replica only retrieves and applies a checkpoint after obtaining a certificate proving its correctness.

Clement et al. [62] propose an approach called *hybrid checkpointing*, shown in Figure 6.1c, which combines infrequent full snapshots with a list of requests that have been ordered in the meantime to create a hybrid checkpoint. This reduces the number of full snapshots necessary and thus also the checkpointing delays, but does not avoid them completely. The verification of such a checkpoint then has to check that both the full snapshot and the list of requests are supported by at least $f + 1$ replicas, that is, one correct replica. Applying a hybrid checkpoint requires the application to first update its

state using the snapshot and then to execute all attached requests. This can introduce a substantial delay, as executing a request typically takes longer than only applying its effects to the application state. Additionally, when a checkpoint is used to recover a faulty replica [58], re-executing the requests can be a problem: if a request triggers a bug in the replica implementation, then the recovered replica could fail again when executing the problematic request, which consequently prevents replicas from recovering.

Dura-SMaRt [46] takes the idea of spreading the costs for creating checkpoints even further: replicas create their full snapshots in turns at different sequence numbers, as shown in Figure 6.1d, such that at most f replicas at a time are taking a full snapshot. As the system has enough redundancy in the normal case to tolerate up to f faulty replicas, the other replicas can fill in for the replicas that currently create their checkpoint to avoid service interruptions. However, in wide-area networks with their non-uniform latencies, as shown in the figures in Section 3.1, replies from different replicas arrive at different times, such that pausing the execution at some replica can cause increased reply times. With the replicas creating their full checkpoints at different points in time, the resulting checkpoints also cannot be compared directly. Instead, a replica that wants to apply a checkpoint has to retrieve the $f + 1$-newest full checkpoint and a log of the requests that were ordered after the checkpoint, apply it and then iteratively update the application state using the request log, until it reaches the next checkpoint and compare that against its current application state. In total, the replica has to reproduce $f + 1$ matching checkpoints (including the initial checkpoint) created by different replicas, which ensures that the current state is correct. If the verification fails, the replica selects a different full checkpoint as starting point and retries the verification procedure. A big downside of this approach, in addition to having to re-execute large amounts of requests and the already discussed associated problems, is that a replica has to apply an unverified checkpoint. If a replica implementation contains bugs that can be triggered by applying a manipulated checkpoint, this can allow an attacker to compromise a replica. As a recovering replica always has to update itself using a checkpoint, a freshly recovered replica may become faulty even before it restarts request processing.

Approach of this Thesis

After collecting a fuzzy snapshot, DFC runs an additional phase that deterministically merges the fuzzy snapshot and the list of state changes into a normal checkpoint. The latter can be used just like a traditional checkpoint and is identical across all replicas, which allows the replicas to verify its correctness *before* applying it.

6.1.3. Providing Flexibility

Depending on the application, it might be sufficient to use full snapshots, which are simpler to implement, or differential snapshots, which are more efficient as they only capture the changed state objects but require a more complex application interface. Thus, a checkpointing approach should be flexible enough to support both variants. Of the already discussed approaches, only the COW snapshots used by PBFT [57] and BASE [59] provide this choice, the other approaches only support full snapshots.

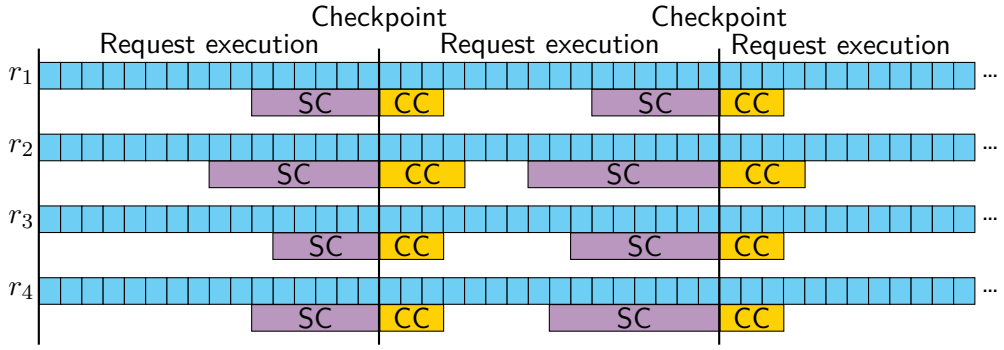


Figure 6.2: Schematic overview of creating a fuzzy checkpoint including the state capture (SC) and checkpoint completion (CC) phases.

Checkpointing should also be flexible enough to support both execution groups containing $2f + 1$ or $3f + 1$ replicas. For example, SPIDER (cf. Chapter 5) and several other approaches [40, 84, 130, 150, 203, 210] only use $2f + 1$ execution replicas. In that setting, it must be possible for a checkpoint to become stable once $f + 1$ replicas support it, which means that a single correct replica must be able to prove the checkpoint's correctness on its own, as the other f replicas might be faulty. With identical checkpoints, a checkpoint can become stable once a replica collects a certificate of $f + 1$ matching CHECKPOINT messages as this constitutes such a proof. All discussed approaches except for Dura-SMaRt create comparable checkpoints.

In contrast, with non-comparable checkpoints like in Dura-SMaRt a correct replica can collect $f + 1$ CHECKPOINT messages, but is unable to verify whether they are matching, without recreating the corresponding checkpoints itself[1]. Thus, up to f of the collected CHECKPOINT messages may be faulty, which can prevent other replicas from verifying the correctness of the checkpoint. With only $2f + 1$ execution replicas it is not possible to wait for more than $f + 1$ checkpoint messages while also guaranteeing liveness and therefore Dura-SMaRt cannot be used when relying on only $2f + 1$ execution replicas.

Approach of this Thesis

We present two variants of DFC that can either capture full or differential snapshots. The latter is then combined with the previous checkpoint to create an up-to-date checkpoint. This results in identical checkpoints on all replicas, such that DFC can also work with $2f + 1$ execution replicas.

6.2. Deterministic Fuzzy Checkpointing

The central idea of our approach Deterministic Fuzzy Checkpointing (DFC) is to first collect a fuzzy snapshot of the application state in a *state-capture* (SC) phase that runs concurrently to the normal request execution as shown in Figure 6.2. As each replica

[1] However, when verifying every checkpoint immediately, Dura-SMaRt would degrade to full checkpointing, which negates all its performance benefits.

creates a new checkpoint every k-th slot, it starts the state-capture phase a bit earlier such that it finishes the collection of the application state before reaching the sequence number for the next checkpoint. Additionally, the effects of all requests executed between the start of the state capture and the checkpoint's sequence number are collected and are combined with the snapshot during a *checkpoint-completion* (CC) phase to make the checkpoint deterministic again, hence yielding a deterministic fuzzy checkpoint. Both phases run concurrently to the request execution to minimize delays.

6.2.1. State Capture

In the following we describe how the application state is captured to create a checkpoint for *target sequence number* s. To ensure that the state capture is complete before reaching the target sequence number, each replica has to start the state-capture phase earlier on. For this, each replica r_i selects a sequence number p_i at which the state capture starts. This *start sequence number* is determined individually by each replica and does not require coordination, as the final checkpoint will be identical on all replicas independent of the start sequence number.

The library uses a separate thread to capture the application state $\mathcal{O} = \{o_1, o_2, ..., o_n\}$ by copying the objects o_i one after another. We assume that replication protocol-specific data like the client counter values and cached replies are captured along with the application state. In addition, once the state capture has started, all changes made to the application state have to be tracked until the execution reaches the target sequence number s. The details depend on the application interface for which we discuss two variants in Section 6.3, including the necessary synchronization for a consistent result.

After the state-capture phase, this results in a *fuzzy checkpoint* $\mathcal{F} = (S_{[p_i,s]}, M_{p_i,...,s})$. The *fuzzy snapshot* $S_{[p_i,s]}$ contains each object of the application state, which each were copied at some point between p_i and s, and the *modifications list* $M_{p_i,...,s}$ contains information about each change of the application state during the state-capture phase. Together these contain enough information to turn the fuzzy checkpoint into a regular full checkpoint.

6.2.2. Checkpoint Completion

A just created fuzzy snapshot cannot be directly compared between replicas. As each replica can choose the start sequence number independently, different fuzzy checkpoints can be captured across different sequence number ranges, and consequently the fuzzy snapshot $S_{[p_i,s]}$ can cover different objects. In addition, the state capture and the request execution can interleave differently at each replica, such that one replica copies an object before executing a certain request and another one copies it afterwards. Both reasons cause replicas to collect different fuzzy snapshots.

The modifications list $M_{p_i,...,s}$ for a snapshot at the different replicas varies in length if different starting points p_i are used. However, as each request execution results in the same state changes on every replica, for any sequence number the replicas would still

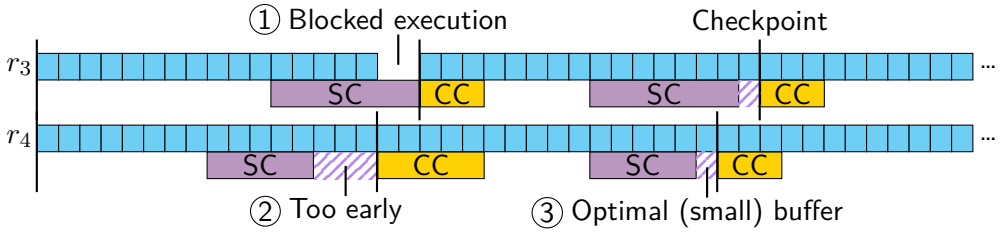


Figure 6.3: Replicas adapt the starting point of a state capture (SC) such that it completes a short time before reaching the target sequence number. A too short or too long time buffer (▨) can cause execution pauses or increase the overhead.

collect the same modifications. Thus, the modification lists at each replica are suffixes of each other.

To make the checkpoints comparable, the checkpoint-completion phase applies the modifications list $M_{p_i,\ldots,s}$ in order onto the fuzzy snapshot. Once completed, this yields the same checkpoint on all replicas. The latest version of each state object is either contained in the fuzzy snapshot or if it was modified during the state capture, then the modification list contains the corresponding change. Thus, after merging both, the resulting checkpoint contains the application state at target sequence number s. Like the state capture, the checkpoint completion runs in a separate thread and therefore allows the request execution to continue in the meantime.

6.2.3. Capture Timing

Replicas in DFC periodically create a new checkpoint after executing a sequence number divisible by k. The state capture for a checkpoint at *target sequence number* s has to start earlier on, which requires each replica r_i to determine a suitable *start sequence number* p_i. As shown in Figure 6.3, it should be selected such that it is early enough to allow the state capture to complete before reaching the target sequence number. If this is not the case ①, then the execution must be paused to ensure that a consistent snapshot can be created. In the opposite direction ②, the state capture should start as late as possible to reduce the overhead of collecting and applying a large modifications list. Ideally, the state capture completes in time with only a small buffer left ③.

To achieve a good balance, after capturing a snapshot, DFC dynamically selects the next start sequence number p_i based on the capture time of the last snapshot. It is calculated as $p_i = \max(s - \lambda \cdot d_{sc} - \delta, \tilde{s} + 1)$, where s is the next target sequence number, $\tilde{s}$ the last target sequence number, δ and λ are parameters used to control the snapshot timing by adding a constant and proportional amount of buffer time, respectively. The duration d_{sc} is the number of sequence numbers that were processed during the last state capture. This dynamic capture-start reduces the overhead when compared to a static offset, which would have to be early enough to cover all situations. The formula also prevents the checkpoint intervals from overlapping, by ensuring that a new state-capture phase starts after the previous checkpoint was captured. In the other extreme, when the state capture does not start before reaching the target sequence number, that is, $p_i = s$,

```
1  // Application interface
2  interface CAW_Application {
3      // Request execution
4      RESULT invoke(REQUEST r);

6      // Notify replication library
7      callback modified(OBJECTID oid);

9      // Checkpointing
10     BYTE[] object(OBJECTID oid);
11     VOID apply(OBJECTID[] oids, BYTE[][] objects);
12 }
```

Figure 6.4: Application interface for the DFC_{caw} variant.

then DFC essentially behaves like full checkpointing. By using the previous state-capture duration, the timing automatically adapts to heterogeneous replicas [84], which due to different hardware or software can require more or less time to capture a snapshot. As the start sequence number is selected independently for each replica, this allows DFC to work with replicas having differing performance characteristics.

6.3. Application Interface

In the following we present two variants of the application interface that can be used to implement deterministic fuzzy checkpointing. The copy-after-write variant DFC_{caw} works at the granularity of whole objects, which allows the replication library to generically manage the object handling and checkpoint completion. In contrast, the update-based variant DFC_{upd} uses application-specific updates to only capture the state parts that actually have changed. This makes the state capture more efficient but also requires an application-specific checkpoint capture and completion, and consequently results in a higher implementation complexity.

6.3.1. DFC_{caw}: Copy-after-Write Variant

When using the copy-after-write variant DFC_{caw}, the state collection works at the granularity of whole objects. We expect each object to have a size of at least a few kilobytes to limit the overhead for managing individual state objects in relation to the overall state size. For the replication library to know which objects have changed, the application has to send a notification whenever it modifies an object. While the state-capture phase is creating copies of every object in the application state, the library keeps track of objects that are modified in the meantime. Afterwards the library copies all of modified objects once the target sequence number s is reached. That is, modified objects

are copied after they were written. The final copy step is limited to objects modified during the state-capture phase and thus only results in a short delay.

Interface

The required application interface for DFC_{caw} is shown in Figure 6.4 and is similar to that used by BASE [59]. The `invoke(`r`)` method is used to execute a client request r and returns the execution result. If any of the state objects are modified while processing the request, the application has to call the `modified(`*oid*`)` callback to inform the replication library that an object with identifier *oid* has changed. The callback can be issued before or after modifying an object, as long as it is reliably called for each modified object. Note that it must always be called independent of whether the replica is currently in the state-capture phase or not, as the library has to learn about the identifiers of all existing objects. Using the `object(`*oid*`)` method, the replication library can retrieve the application-specific serialization of the state of the object with identifier *oid*. For non-existent objects, the method instead returns a special nil value $\perp$. In order to restore a checkpoint, the replication library calls `apply(`*oids*`,` *objects*`)` with a list of all *objects* that have changed, along with their identifiers *oids*.

Object Identifier Tracking

To capture a full checkpoint, the library must know the identifiers of all existing objects. Therefore, the library maintains a set $\mathcal{I}$ of existing object identifiers to which it adds all new identifiers returned by the `modified()` callback. If `object()` returns the nil value $\perp$, the identifier is removed from the set. The state-capture phase then uses the set $\mathcal{I}$ to collect a fuzzy snapshot $S_{[p_i,s]}$ of all these objects. The replication library also collects a set $\mathcal{I}_{sc}$ of objects modified during the state-capture phase. After reaching the target sequence number s, the library copies the current state of all objects in $\mathcal{I}_{sc}$. During this step the request execution must be paused. The resulting modification list $M_{p_i,\ldots,s}$ is then used to update the set $\mathcal{I}$ with the identifiers of added and removed objects.

Checkpoint Completion

The checkpoint completion step finally merges the fuzzy snapshot $S_{[p_i,s]}$ and the modification list $M_{p_i,\ldots,s}$ to create the checkpoint by replacing all modified objects with their latest version. If an object has value $\perp$, then it is removed from the checkpoint.

Correctness

The checkpoint is guaranteed to match the application state after executing sequence number s. The fuzzy snapshot includes a copy of each object that was referenced by $\mathcal{I}$ at sequence number p_i or $\perp$ if the object was removed in the meantime. If an object o_j was last modified before sequence number p_i, then the fuzzy state already includes its current state. Otherwise, the object's identifier was added to $\mathcal{I}_{sc}$ and thus the object is copied when reaching s. Therefore, the modification list contains an up-to-date copy, which is then merged into the final checkpoint. The above case also applies to newly created objects. Objects that were deleted by a request, yield $\perp$ when retrieving their state using `object()` and are removed while completing the checkpoint.

```
1  // Application interface
2  interface Upd_Application {
3      // Request execution
4      [Result, Update] invoke(Request r, boolean createUpdate);

6      // Checkpointing
7      Snapshot fuzzy();
8      Checkpoint complete(Snapshot S_[p_i,s], Update[] M_{p_i,...,s});
9      void apply(Checkpoint cp);
10 }
```

Figure 6.5: Application interface for the DFC_{upd} variant.

Object Retrieval

Retrieving an object copy from the application must be possible even for objects that are modified by a request that executes concurrently. For example, this can be achieved by employing per-object locking, such that the object copy is based on a consistent state before or after the object is modified. For legacy applications it is also possible to interleave request execution and state capturing, which avoids concurrent accesses to objects, but can slow down the request execution.

Comparison with BASE

Although the application interface is similar to that used by BASE [59], there are significant differences in the provided properties. BASE requires the application to always notify the replication library before modifying a state object, whereas DFC_{caw} is more flexible and supports notifications before, during or after an object was modified, giving developers the flexibility to choose the most suitable point in time to issue the callback. For each modified object, BASE has to synchronously back up its state which for large objects can lead to execution delays. In DFC_{caw} most objects can be captured asynchronously during the state-capture phase and thus allow the request execution to continue. The checkpoint completion in BASE has to happen synchronously when reaching the checkpoint sequence number. During this phase it becomes necessary to additionally export the current state of all modified objects for the hash calculation. DFC_{caw} instead allows the checkpoint completion to proceed asynchronously and only has to capture an object multiple times if it is modified during the state-capture phase. In Section 6.4 we discuss how to extend DFC_{caw} with differential checkpointing, which is also used by BASE, such that only objects that were modified since the last checkpoint are captured.

6.3.2. DFC_{upd}: Update Variant

The DFC_{upd} variant uses an update-based application interface that delegates more functionality to the application in exchange for a more efficient handling of state modi-

fications. The application becomes responsible for managing the capture of the fuzzy snapshot and has to generate application-specific updates that capture modifications of the application state. This allows tailoring the updates to the application's needs.

Interface

The interface of DFC_{upd} is presented in Figure 6.5. The `invoke(`r`,` *createUpdate*`)` method executes request r and offers the *createUpdate* option to enable or disable the creation of an update that captures the effects on the application state. That is, the update contains information on all modified objects. During the state-capture phase, the replication library collects them in a modification list $M_{p_i,\ldots,s}$. For efficiency reasons, update creation is only enabled between the start of the state capture and the target sequence number s for the checkpoint. The replication library calls the `fuzzy()` method from a separate thread during the state-capture phase, then the application is responsible for creating a fuzzy snapshot $S_{[p_i,s]}$. Merging the fuzzy snapshot $S_{[p_i,s]}$ and the modification list $M_{p_i,\ldots,s}$ is done using the `complete(`$S_{[p_i,s]}$`,` $M_{p_i,\ldots,s}$`)` method, which combines both into a deterministic fuzzy checkpoint. The application state can be updated by passing a checkpoint *cp* to the `apply(`*cp*`)` method.

Checkpoint Completion

For the checkpoint to be correct, updates must be applicable to the fuzzy snapshot independent of whether the affected objects were copied before or after creating the update. More precisely, for an object o_j applying all updates, which were created between sequence number p_i and s to the object state, which was captured at some point in the same timespan, must update the object to its latest state. In the following we describe two possible variants how updates can be structured to satisfy this requirement. As capturing a fuzzy snapshot has to copy objects one by one, different objects are copied at different points in time. Therefore, an update must always be applicable even if some of the objects relevant for the update already contain a later state and others still have to be updated. Consequently, an update should be a collection of modifications to individual objects in which each modification must be applicable independently. The update for an individual object can for example use one of the following constructions:

- Updates can be structured such that these are not applied when the state of an object is newer than expected. This can be checked by including version numbers into the objects and updates, and later on skipping old updates by comparing the version numbers. For this variant, the state capture must consistently capture objects at a state before or after these are modified by a request, to ensure that the version number is accurate.

- Another variant is to create only updates that can be applied without reading from the current state of the object in the fuzzy snapshot. For example, updates could blindly (over-)write selected parts of an object or delete the object, and just ignore errors if the object does not exist yet. These operations are then independent of the object's state. This behaves similarly to DFC_{caw}, but by being application-specific it is possible to apply modifications at a much finer granularity.

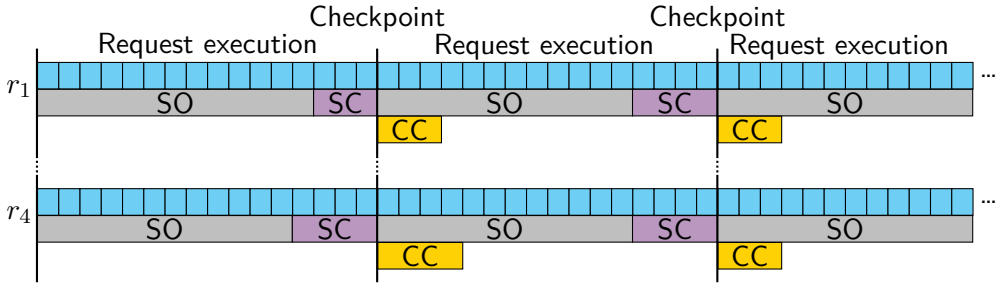


Figure 6.6: Schematic overview of creating a differential fuzzy checkpoint including the state observation (SO), state capture (SC) and checkpoint completion (CC) phases.

Correctness

The properties of the fuzzy snapshot and the updates ensure that the resulting checkpoint is correct. Objects that were last modified before sequence number p_i are already captured with their latest state by the `fuzzy()` method and the application by construction creates no updates for them. We now consider objects that are modified between sequence numbers p_i and s. As updates must be applicable to individual objects, we focus on an arbitrary but fixed object o_j in the following. Starting from sequence number p_i, modifications of object o_j result in the creation of an update. The fuzzy state-capture phase also retrieves a state after this starting point. Thus, the captured object version is either the version just before the first update, or some state in between or after the updates. As applying all updates for an object must yield its latest state, the resulting checkpoint contains the latest state of each object. Thus, the checkpoint is identical to a full checkpoint captured right after sequence number s.

6.4. Differential Deterministic Fuzzy Checkpointing (DDFC)

For applications with large states usually only a fraction of all objects are modified between two consecutive checkpoints. To create an up-to-date checkpoint, it is sufficient to capture only these modifications and then update the previous checkpoint. This can significantly reduce the number of state objects that have to be captured and thereby speeds up the snapshot creation. We add a *state observation* phase, which is included in Figure 6.6, and tracks which objects have changed between the last checkpoint and the start of the state-capture phase. The replication library then only captures these changed objects. Afterwards during the state-completion phase, this differential fuzzy checkpoint is combined with the previous full checkpoint into an up-to-date deterministic fuzzy checkpoint.

State Observation

The exact approach to collect a list of changed objects since the last checkpoint depends on the used variant of deterministic differential fuzzy checkpointing. The differential copy-after-write-based variant DDFC_{caw} maintains a set of identifiers $\mathcal{I}_o$ that contains all objects that were modified during the state-observation phase, that is, after the sequence

number $\tilde{s}$ of the last checkpoint and before the new state-capture phase starts at p_i. It replaces the set $\mathcal{I}$ containing the identifiers of all application objects, as this information is implicitly available by combining the objects in the last checkpoint and those in $\mathcal{I}_o$.

For the differential update-based variant DDFC_{upd}, the replication library instructs the application to create small meta updates when executing requests. The *meta updates* only contain the information which objects were modified, but not the modifications itself and thus are cheap to create. The library then collects these meta updates as a list $U_{\tilde{s}+1,\ldots,p_i-1}$. While it would be possible to always create full updates, this would lead to redundantly capturing all changes to objects that change more than once between checkpoints. Using meta updates during the state-observation phase, the generation of full updates remains limited to the state-capture phase.

State Capture

To create a differential fuzzy checkpoint $\Delta\mathcal{F} = (\Delta S_{[p_i,s]}, M_{p_i,\ldots,s})$, the state-capture phase collects a differential fuzzy snapshot $\Delta S_{[p_i,s]}$, which only contains objects modified during the state-observation phase. DDFC_{caw} now only copies the objects whose identifier is part of $\mathcal{I}_o$. In contrast, when using DDFC_{upd}, the replication library passes the meta updates $U_{\tilde{s}+1,\ldots,p_i-1}$ to the `fuzzy()` method to inform it that these objects should be exported. Like with full checkpoints, the modifications list $M_{p_i,\ldots,s}$ contains all modifications made to objects during the state capture.

Checkpoint Completion

To create a normal checkpoint, the differential fuzzy snapshot $\Delta S_{[p_i,s]}$ from the differential fuzzy checkpoint $\Delta\mathcal{F}$ is first combined with the previous checkpoint into a fuzzy checkpoint $\mathcal{F}$. Afterwards the modification list is applied to the fuzzy snapshot in the same way as for full checkpoints.

Correctness

Combining the previous checkpoint with the differential fuzzy snapshot $\Delta S_{[p_i,s]}$ yields a fuzzy snapshot with the following properties. An object o_j that was not modified during the state observation keeps its old state, thus its state is identical to the state the object still had at sequence number p_i. All other objects get the state that was captured as part of $\Delta S_{[p_i,s]}$. That is, the resulting fuzzy snapshot is indistinguishable from a full fuzzy snapshot captured between p_i and s. Then the correctness proofs for merging the modification list still apply.

6.5. Optimizations

In the following we discuss several possible optimizations that can further increase the efficiency of DFC or reduce its (performance) impact on the request execution.

Fine-Granular Modification Tracking

During the state-capture phase, the replication library tracks all modifications to any object. This ensures that all objects can be updated to their latest state. However, the modification tracking for an object is actually only necessary once the object has been

copied, earlier modifications are not required to update the object. Thus, it is sufficient to only collect modifications for objects which have already been copied.

Throttled State Capturing

Depending on the application, the state-capture phase and the request execution may have to compete for shared resources like CPU time, disk access, or locks. In this case, it can be beneficial to throttle the speed at which state objects are captured by introducing a short delay between capturing two state objects. Selecting a specific delay has to make a trade-off between an increased state-capture duration and a lower performance impact on the request execution.

Lazy Checkpoint Completion

Creating a checkpoint message, which includes the checkpoint hash, can be expensive, as it requires computing the hash of all changed objects. When using at least $3f+1$ replicas, it becomes possible to defer the calculation of the checkpoint hash until the checkpoint is requested by another replica. Instead, at least $f+1$ correct replicas provide a promise that the checkpoint exists and can be retrieved if necessary. This requires a replica to collect $2f+1$ promises such that $f+1$ of them must originate from correct replicas. These replicas can later on vouch for the correctness of the checkpoint if it becomes necessary for a replica to retrieve it. The lazy checkpoint hash calculation also allows replicas to lazily merge the captured fuzzy snapshot and the modification list.

Workload-Dependent Checkpoint Intervals

The amount of work to create a checkpoint can vary depending on the workload. For a fixed number of requests, for example, the size of the objects that must be captured can drastically differ depending on the size of the processed requests. To keep the delay introduced by checkpointing below a certain threshold, the interval at which checkpoints are created should adapt accordingly.

For this we first have to revisit the checkpoint interval k. As discussed in Section 4.5 for Isos, it is actually only necessary for all replicas to create checkpoints at the same logical point in time, that is at the same sequence numbers, but it is not necessary to use periodic intervals. Thus, the replicas can pick nearly arbitrary sequence numbers at which to create checkpoint, as long as the replicas always agree. To bound the size of the state that has to be kept, there should still be an upper limit for the number of agreement slots processed between two consecutive checkpoints. We therefore allow replicas to dynamically schedule the creation of a checkpoint for a sequence number s while limiting the distance to the checkpoint interval k such that $s \leq \tilde{s} + k$ where $\tilde{s}$ is the sequence number of the last checkpoint.

By using a deterministic measure to calculate the target sequence number, it is possible to do so without explicit communication between replicas. Possible criteria could be the number of write requests executed or the summed up request size in bytes since the last checkpoint. After reaching a certain threshold, the replicas schedule a checkpoint for sequence number $s = \min(s_t + \rho \cdot k, \tilde{s} + k)$ where s_t is the sequence number at which the threshold was reached and ρ specifies a fraction of the checkpoint interval k to reserve time to run the state-capture phase. Additionally, the sequence number s is bounded to

adhere to the maximum distance between two checkpoints. Like before, its precise start is determined locally at each replica.

6.6. Evaluation

In the following we evaluate the performance impact of checkpointing using the classical checkpointing methods and the different deterministic fuzzy checkpoint variants. We first compare the different methods for creating full checkpoints, before switching to differential checkpointing. The section concludes with a comparison of the copy-after-write and update-based variants of DFC.

6.6.1. Setup

We compare three different checkpointing approaches in our evaluation. Firstly, $\mathbf{BFT_{full}}$ serves as baseline and represents the classical approach of creating full checkpoints [47, 73, 84], which pauses the request execution while taking the application snapshot. Secondly, $\mathbf{DFC_{caw}}$ is the copy-after-write variant of DFC, which captures the application state at the granularity of objects concurrently to the request execution. Thirdly, $\mathbf{DFC_{upd}}$ uses application-specific updates to collect state updates more efficiently.

In addition, we separately evaluate the differential counterparts of each approach. $\mathbf{BFT_{diff}}$ [57, 59] only captures the state parts that have changed since the previous checkpoint in order to reduce the pause duration. It is compared with the differential variants of DFC, that is $\mathbf{DDFC_{caw}}$ and $\mathbf{DDFC_{upd}}$.

Our evaluation does not include Dura-SMaRt [46] as it requires replicas to apply unverified checkpoint state, which can be a security risk as discussed in Section 6.1.2. In addition, it is also incompatible with protocols like SPIDER (see Chapter 5) that use small execution groups consisting of only $2f + 1$ replicas.

All checkpointing approaches are implemented in a single codebase to allow for a meaningful comparison. The implementation is written in Java and uses PBFT [57] as agreement protocol. It uses SHA256 [164] to calculate the checkpoint hash and HMACs [133] with SHA256 to authenticate messages.

Our implementation of BFT_{full} and BFT_{diff} are optimized in comparison to the original PBFT implementation in that they, when reaching the checkpoint sequence number, only block to capture the application state for the checkpoint hash calculation, but run the more expensive checkpoint hash calculation asynchronously. As our evaluation in Section 6.6.1 shows, this significantly reduces the request execution delay as the hash calculation takes more time than retrieving the application state.

Replicas and Clients

The system is configured to tolerate one fault $f = 1$ and consequently consists of four replicas that each run on a server (Intel Xeon E3-1275 CPU, 3.6 GHz, 16 GiB RAM) in our local testbed using Ubuntu 18.04.2 LTS and OpenJDK 11. We run 100 client instances on a single server (Intel Xeon E5645 CPU, 2.4 GHz, 32 GiB RAM) using the same software. All servers are connected using Gigabit Ethernet.

The experiments run in our local testbed, as the focus is on the checkpoint creation and in particular the state capture which does not require communication between replicas and is therefore not affected by wide-area latencies. In addition, with the lower local area latencies we get a clearer view on the processing delays caused by checkpointing.

The agreement protocol creates batches of up to five requests with a combined total size of 10 KiB. Unless mentioned otherwise, we use a checkpoint interval $k = 100{,}000$.

Application

As application for our experiments we use a key-value store that stores a large number of key-value pairs with a value size of 4 KiB. Each key-value pair corresponds to a state object and is identified by its key. An object also contains metadata including a last-accessed timestamp which is updated each time the key is queried or written.

The dataset is stored in an in-memory SQLite database[2]. By keeping the dataset in-memory, this speeds up the data retrieval and thus benefits traditional approaches which have to pause the request execution to capture the application state.

As workload, unless stated otherwise, the clients issue an equal mix of query and write requests which return the state of an object and replace its data, respectively. The client randomly selects which key to access for each request. With a large application state typically only a part of all objects are actively used, thus we configure our clients to access and modify only a fraction of the application state. Unless noted otherwise, only half the objects in the application state are accessed.

In order to retrieve the state of a specific object when using BFT_{diff} or $\text{(D)DFC}_{\text{caw}}$, the application loads the data from the database and serializes it into a byte buffer. For $\text{(D)DFC}_{\text{upd}}$ the application creates updates only containing the object metadata or the whole object depending on the data changed by the request. For example, for a query request the update only includes the metadata change.

Experiments

All experiments run for 240 seconds, of which we remove a warm-up phase consisting of the first 230,000 sequence numbers. With our standard checkpoint interval of 100,000 sequence numbers, this removes the first three checkpoints at sequence numbers 0, 100,000 and 200,000 to give the system time to warm up. The warm-up phase also includes a short timespan after the third checkpoint to remove the corresponding checkpoint-completion phase and checkpoint certificate collection. Depending on the system's throughput, this warm-up phase results in removing roughly the first 75 to 100 seconds. We report the data measured during the 120 seconds immediately following the warm-up phase. The clients measure the timing of each request individually, based on which we determine the throughput and response time.

We want to answer the following questions in our evaluation:

1. What influence does BFT_{full} and the corresponding DFC variants have on the throughput and response times of the system?

2. Which improvements do BFT_{diff} and the differential DFC variants offer?

[2] https://sqlite.org/releaselog/3_23_1.html

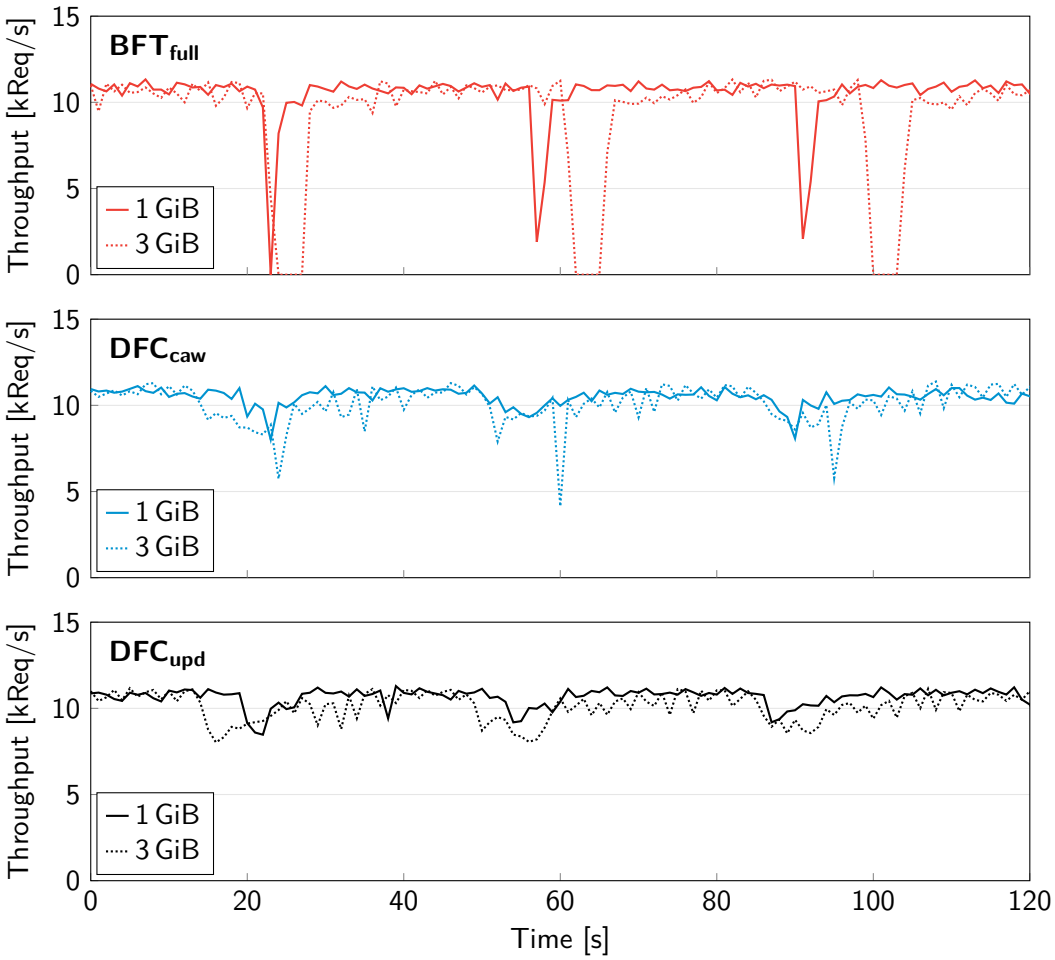


Figure 6.7: Full checkpointing: impact of BFT_{full}, DFC_{caw} and DFC_{upd} on the request processing throughput for a state size of 1 and 3 GiB.

3. How much influence does the number of changed objects have on the differential checkpointing approaches?

4. How efficient are $DDFC_{caw}$ and $DDFC_{upd}$ in comparison with each other?

6.6.2. Full Checkpointing

To answer the first question, we measure the impact of creating full checkpoints on the request execution. For this, we run the application using two different state sizes: once with 250,000 objects with 4 KiB of data each, resulting in a state size of about 1 GiB, and with 750,000 objects, which correspond to 3 GiB. We compare the BFT_{full}, DFC_{caw} and DFC_{upd} approaches with each other. Figure 6.7 shows the achieved throughput after the warm-up phase of all three approaches. With the checkpoint interval of 100,000

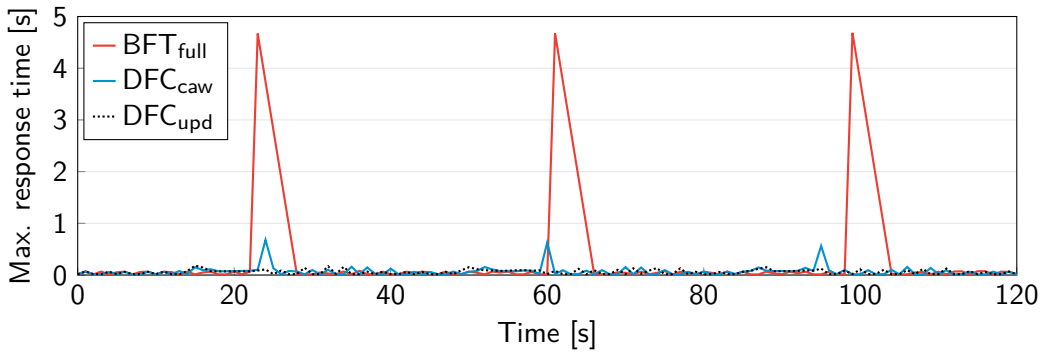


Figure 6.8: Response-time spikes caused by creating full checkpoints for a 3 GiB state

sequence numbers and the used batching configuration, the replicas process roughly 350,000 requests between two checkpoints. Depending on the achieved throughput this translates to creating a new checkpoint approximately every 33 to 38 seconds.

Response Time Increase due to Checkpointing

Our measurements show that creating a full checkpoint can lead to substantial delays during request processing. The throughput drops in Figure 6.7 for BFT_{full} are caused by delays of up to 1.3 seconds to snapshot a 1 GiB application state and up to 4.7 seconds for the 3 GiB setting. The latter setting is also shown in more detail in Figure 6.8.

In contrast, DFC_{caw} and DFC_{upd} capture the application state concurrently to the request execution, which drastically reduces the response time impact. For DFC_{caw}, the delay remains below 225 ms for the 1 GiB setting and 670 ms for 3 GiB. These shorter delays are a result of DFC_{caw} pausing the request execution while capturing the modification list, which only requires copying the state of all objects that have changed during the state-capture phase. As we discuss in Section 6.7, to reduce this delay further, it would be possible to repeatedly capture the modification list before reaching the checkpoint's target sequence number and thereby reduce the number of remaining modifications to collect. The throughput graphs also show that the state-capture phase, which runs before creating the checkpoint, only leads to a small decrease in throughput. This is despite the fact that the concurrent state capture and request execution have to coordinate with each other.

Using DFC_{upd} further decreases the maximum delays to 154 ms for 1 GiB of state and 181 ms using 3 GiB. DFC_{upd} already captures all required information during the state-capture phase such that there is no need to pause the request execution. A further analysis of the measured delays shows that the garbage collection pauses of the Java virtual machine are a significant contributor to them. The leader replica experiences such pauses of up to 63 ms and 144 ms for the different state sizes. Pauses of the leader replica directly translate into delayed request proposals and thereby are visible as increased response times to the clients. Therefore, a further decrease of the maximum response times requires reducing the garbage collection pauses.

Impact of Checkpointing on Throughput

For the 1 GiB setting, BFT_{full} and DFC_{caw} achieve a throughput of about 10.5 thousand requests per seconds, and DFC_{upd} reaches a slightly higher throughput of 10.7 thousand requests per second. With the larger 3 GiB setting, the throughput of BFT_{full} decreases to 9.3 thousand requests per second, whereas both variants of DFC sustain a nearly 9% higher throughput of 10.1 thousand requests per seconds. This confirms that deterministic fuzzy checkpointing is effective in significantly reducing the maximum response times while also achieving a throughput similar or higher than that of full checkpointing.

Asynchronous Checkpoint Hash Calculation

Different from PBFT, our implementation of full checkpointing first captures the whole application state while the request execution is paused and afterwards asynchronously computes the checkpoint hash instead of executing both steps during the execution pause. As previously discussed, for BFT_{full} the state capture takes 1.3 and 4.7 seconds. The duration until the corresponding checkpoint becomes stable is 2.7 and 8.9 seconds for a 1 and 3 GiB state size, respectively. During this time, a replica computes the checkpoint hash and exchanges the corresponding checkpoint messages with the other replicas to collect a checkpoint certificate. As our experiments run in a local area setting, the latter part only contributes a small amount of latency, such that the delay is nearly exclusively caused by the hash computation. Thus, BFT_{full} significantly benefits from computing the checkpoint hash asynchronously, as that takes longer than the state capture itself. This shows that our implementation variant of PBFT indeed reduces the response-time spikes while creating a checkpoint, resulting in a more competitive comparison.

6.6.3. Differential Checkpointing

To answer the second and third question, we investigate the performance impact of differential checkpointing and experiment with different checkpoint intervals. For this, we compare the differential checkpointing variants BFT_{diff}, DDFC_{caw} and DDFC_{upd}. As differential checkpointing is intended for large application states, we configure the application to use 750,000 objects, resulting in a state size of 3 GiB, and now use a checkpoint interval of 50,000 and 100,000.

Comparison to Full Checkpointing

The resulting throughput is shown in Figure 6.9 and the maximum response time in Figure 6.10. For a comparison with the full checkpointing experiments, we first focus on the checkpoint interval $k = 100{,}000$. Compared to creating full checkpoints, using BFT_{diff} only results in a maximum response time of slightly above 1.7 seconds. This shows that differential checkpointing indeed reduces the work necessary to create a checkpoint and thus is suitable for larger application states than full checkpointing. However, for large numbers of changed objects - in our experiment more than 200,000 objects change between two checkpoints - it still results in significant delays.

For DDFC_{caw}, the maximum delay visible to clients shrinks to 303 ms. As the state-capture phase now has to copy fewer data, the list of objects changed in the meantime shrinks as well and therefore the final step to create the modification list also has to

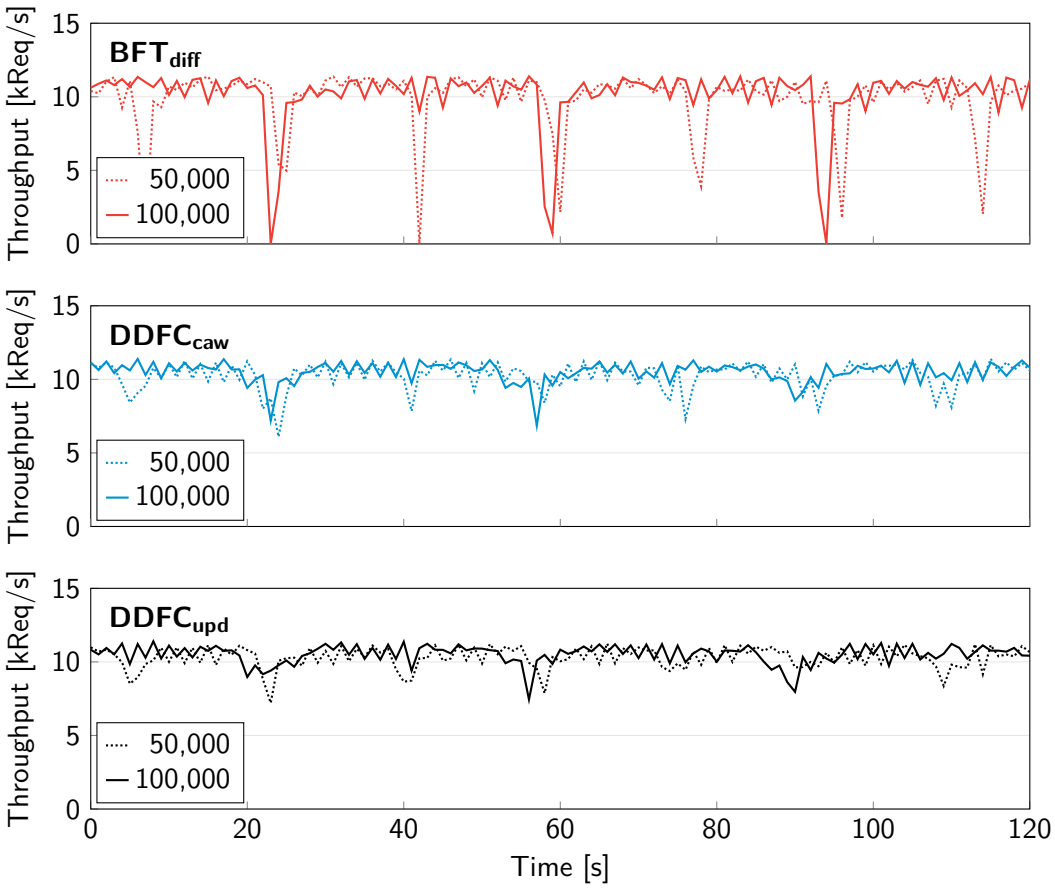


Figure 6.9: Differential checkpointing: impact of BFT_{diff}, DDFC_{caw} and DDFC_{upd} on the request processing throughput for a checkpoint interval of 50,000 and 100,000.

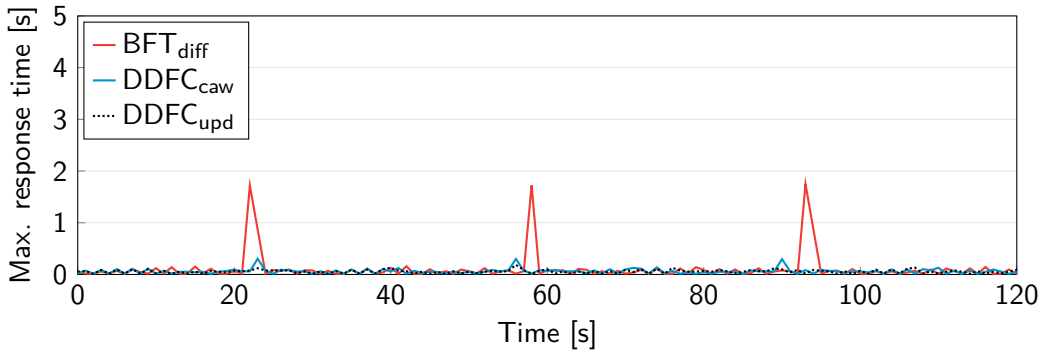


Figure 6.10: Response-time spikes caused by creating differential checkpoints for a 3 GiB state using a checkpoint interval of 100,000.

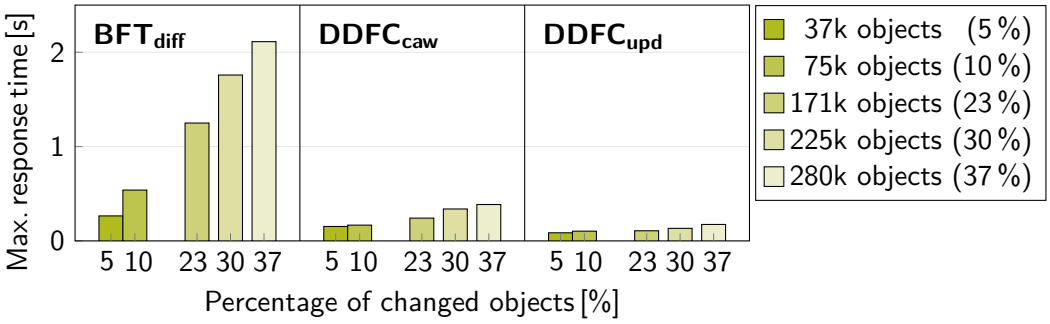


Figure 6.11: Maximum response times of differential checkpointing for a varying fraction of modified objects.

capture fewer objects, which results in a shorter execution pause. The delay times for DDFC_{upd} remain nearly unchanged, as it does not pause the request execution.

The reduced work to capture the application snapshot also yields an increased throughput. BFT_{diff} achieves a nearly 9% higher throughput than BFT_{full}, which is a result of the much shorter execution pauses. Both DDFC_{caw} and DDFC_{upd} increase the throughput compared to their full checkpoint variants by about 4%. For them the throughput improvement has a different cause. During the concurrent state-capture phase, the request processing throughput degrades slightly. With differential checkpointing that phase takes less time to complete and therefore results in a smaller performance impact.

Varying the Checkpoint Interval

We now analyze the effect of cutting the checkpoint interval in half, that is, down to 50,000. For BFT_{diff} this reduces the maximum response time by 27%, down to slightly below 1.3 seconds. This less than expected pause time reduction can be explained in parts by the number of changed state objects which only decreases by approximately 40%. For DDFC_{caw} and DDFC_{upd} the maximum response times remain similar to those for the larger checkpoint interval. Finally, the throughput for all variants is reduced by 2% as a consequence of creating checkpoints more frequently.

Varying the Amount of Changed Objects

To answer the third question more thoroughly, we modify our previous experiment such that clients only access a configurable fraction of the application state. This allows us to control the number of changed objects between two checkpoints. As before, the size of the application state is 3 GiB and the checkpoint interval is $k = 100{,}000$.

As shown by Figure 6.11, the processing delay that is caused by checkpointing approximately scales with the number of changed objects. In addition, DDFC_{caw} and DDFC_{upd} consistently provide lower maximum response times that BFT_{diff}, confirming the results of our previous experiments.

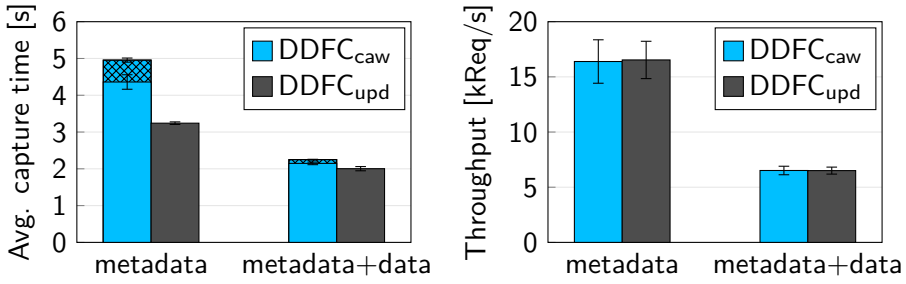


Figure 6.12: Average duration of the state-capture phase and throughput for both DDFC$_{caw}$ and DDFC$_{upd}$. Each bar shows the time to create the fuzzy snapshot ☐ and to capture the modification list ▨.

6.6.4. Comparing DDFC$_{caw}$ and DDFC$_{upd}$

In our final experiment, we investigate differences between DDFC$_{caw}$ and DDFC$_{upd}$ to answer question four. For this we compare two different workloads. The first one only changes the object metadata and consists of requests that query some random key. This results in an update of the last accessed timestamp in the corresponding object's metadata. In contrast, the second workload changes both metadata and data by issuing requests that overwrite the data stored for a key and thus both the object's data and metadata change.

Figure 6.12 shows the throughput for both workloads. The higher throughput and capture time of the metadata-only workload are largely a consequence of different batch sizes. For the metadata-only workload due to its smaller request sizes, the batches contain up to the maximum of five requests, whereas for the second workload with its larger write requests only two requests can be batched together.

When only metadata is changed, then DDFC$_{upd}$ takes 35% less time than DDFC$_{caw}$ to capture the application state. For DDFC$_{caw}$, we also include the time to capture the objects in the modification list. The improvement is twofold. The updates captured by DDFC$_{upd}$ only contain the object metadata and thus are cheaper to create than copying the full object. In contrast to DDFC$_{caw}$, there is also no need to later on copy each object that was modified during the state-capture phase to build the modifications list. When both metadata and data are modified, then the lead of DDFC$_{upd}$ shrinks to 11% as both variants now have to capture the full object data.

6.6.5. Discussion

To put the checkpointing delays reported in Section 6.6.2 into perspective, for a 1 GiB state Bessani et al. [46] report that capturing the snapshot takes multiple seconds when using full checkpointing. Our implementation compares favorably to that and only requires 1.3 seconds to capture a 1 GiB state. This benefits BFT$_{full}$ and BFT$_{diff}$, which pause the request execution while creating the snapshot. When storing the checkpoint

on HDD or an SSD, the reported delay increases to more than ten seconds [46]. This suggests that the performance benefits of DFC can be even larger in such a setting.

6.7. Related Work

This section discusses related approaches to DFC that are used in different contexts and reviews methods to efficiently transfer checkpoints.

Databases

In the context of crash faults, the approach of creating fuzzy checkpoints was originally developed for relational databases [113, 143, 178]. To limit the size of the database log, which is required for crash recovery, these from time to time create a fuzzy snapshot of the database. Together with a redo-log, the database can update the captured state to be consistent. In order to handle long-running transactions, which start before the snapshot creation, it may still be necessary to keep redo-log entries from before the start of the checkpoint creation. Due to handling only crash faults, it is sufficient to create a snapshot for an arbitrary point in time, whereas DFC has to guarantee that all replicas create the exact same checkpoints. Additionally, in DFC there are no transactions across multiple requests, thus the modification log begins with the state-capture phase.

The fuzzy checkpointing approach is also used in some replicated databases [213] or key-value stores [122]. These create a fuzzy snapshot by iterating over the stored data without locking and creating copies of all entries. This process can be triggered from time to time [122] or whenever a snapshot is requested to update a replica [213]. By reapplying the modification log since the start of the fuzzy checkpoint collection, the database state will return to a consistent state.

With crash-tolerant databases each replica can start and complete the snapshot creation at different times, as it is not necessary for the snapshots to be comparable. In fact, the fuzziness of the snapshot is only resolved when using the snapshot to recover the current state. To tolerate Byzantine faults, the snapshot creation must however adhere to more stringent criteria. The replicas require a proof that the content of a snapshot is correct, which in turn requires the creation of identical snapshots on all replicas. This forces the replicas to coordinate the checkpointing process.

Disk-Based Copy-on-Write

VM-FIT [76] and ZZ [209] use disk-based copy-on-write snapshots to create checkpoints. For this, the application has to persist all volatile state to disk and then request the virtual machine platform or the filesystem to create a copy-on-write snapshot. Compared to DFC, these approaches are only effective if most data is already stored on disk, as otherwise significant pause times can arise from writing the application state to disk. It is also impractical for applications that keep their state in memory. In addition, the implementation becomes dependent on specific features of the underlying filesystem or platform, which increases the amount of platform-specific code, hence requiring more effort to port the library. This complicates using a diverse set of platforms to improve

fault independence [96], as different platforms, for example, support different filesystem features. In contrast, DFC only requires standard operating system interfaces.

Virtual Machine Migration

The snapshot creation for DFC_{caw} bears a certain resemblance with virtual-machine live migration [61]. The latter allows moving virtual machines between hosts by copying their memory pages while the virtual machine is running. The hypervisor starts by transferring a copy of all memory pages of the virtual machine to the migration target while the virtual machine is still running. In a second step all pages modified in the meantime are transferred again. This is repeated until the remaining changes are small enough or a certain number of steps has been reached. As final step, the virtual machine is paused and the remaining pages are transferred. The iterative transmission reduces the amount of data to copy in the final step and thereby the duration for which a virtual machine is paused. A similar iterative approach would be possible for DFC_{caw}. Instead of just creating a fuzzy snapshot as first step and copying the remaining objects in the final step, it would be possible to also iteratively collect objects change in the meantime.

Checkpoint Transfer

For DFC we have focused on creating large checkpoints without causing large latency spikes. In order to apply such a checkpoint at a recovering replica, it is nevertheless important to be able to quickly transfer it. In the following we sketch approaches that can be used for a high-performance transfer of the checkpoints created by DFC.

PBFT [56] optimizes the checkpoint transfer by letting a replica only request the objects in a checkpoint for which the replica does not have the current state. To do so efficiently, a replica first retrieves the hashes of the state objects and then only requests the missing ones. By organizing the objects in a Merkle tree (cf. Section 5.7.2) this can be optimized further to quickly skip identical tree parts. Using specialized tree variants, the overhead for detecting missing objects can be further optimized [134].

To optimally adapt the object transfer to the available wide-area network throughput, Kapitza et al. [127] propose a pipelined object-transfer mechanism. A replica always requests objects in batches. Once the first object of a batch arrives, the replica requests a new batch whose size is selected to saturate the available network throughput for the duration of a round-trip time. That way, the network remains fully used while still being able to quickly adapt to changes in available bandwidth.

Besides optimizing the time to transfer the replica state, the time to recover a replica can also be optimized. For this the request execution can be modified to allow loading the application state incrementally [151, 209]. Only the parts necessary to execute a specific request have to be loaded, other parts of the application state can be recovered at a later time. This allows a replica to start executing requests while it is still recovering.

6.8. Summary

The periodic creation of checkpoints can cause large delays for the request execution. DFC aims to eliminate these delays by running the state-capture phase concurrently to

the normal request execution. The resulting fuzzy checkpoint consists of the captured fuzzy snapshot and a modification list containing all state changes that occurred in the meantime. Together these are asynchronously combined by a state-completion phase into a deterministic fuzzy checkpoint, which is identical on all replicas. We have presented a copy-after-write and an update-based variant, which trade off implementation complexity and efficiency. With the help of an optional state observation phase, DDFC only has to capture the changed state parts. As our evaluation shows, this enables our approach to reduce its execution delays even further compared to traditional checkpointing.

7 Conclusion

This thesis has addressed the problem of reducing response times from different angles. In the following, Section 7.1 summarizes the main results, Section 7.2 discusses directions for further possible research before we conclude the thesis in Section 7.3.

7.1. Summary

In this section, we revisit the three approaches presented in the previous three chapters, which each focus on one of the protocol phases shown in Figure 7.1. We discuss how the approaches address our two central questions, namely which *improvements of the client-perceived response times* as well as *reduction of performance variations* they achieve.

Egalitarian Fault Tolerance

The first step to process a client request is the transmission of the request from the client to the responsible leader replica. If the client and the leader replica have a high communication delay, then the request transmission alone can result in a significant

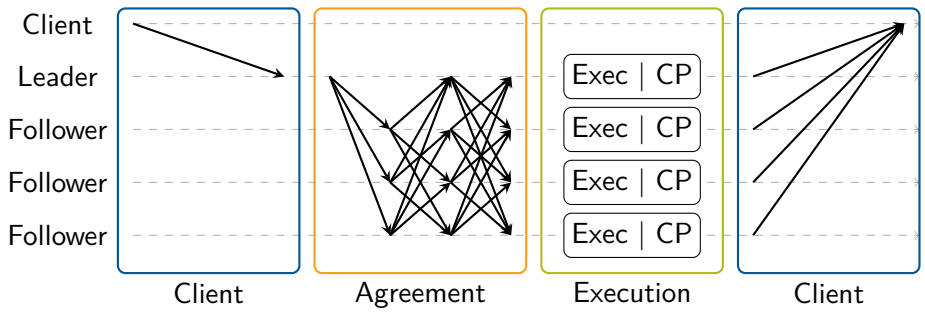


Figure 7.1: Protocol steps that are necessary to process a client request, grouped into client communication-, agreement- and execution-specific steps at the example of PBFT.

increase of the response time. In addition, response times become sensitive to changes of the leader location and have the potential for large performance variations.

ISOS requires $3f + 1$ replicas and allows clients to submit their request to the nearest replica, which immediately initiates the agreement for the request. For this, each replica has its own set of sequence numbers to which it can assign requests. The order across different replicas is then established using dependencies between requests. This relies on the fact, that for strong consistency it is sufficient to only order requests which conflict with each other [169]. The dependencies for a request are determined by a quorum of replicas, which were selected by the replica that received the request. If each dependency is reported by at least $f+1$ of those replicas, then the agreement can complete on the fast path. Otherwise, the protocol finishes processing the request via a reconciliation path for which the replicas exchange their dependencies in an additional protocol phase before completing the agreement. To ensure that faulty replicas cannot introduce non-existent dependencies to prevent a request from executing, a replica only accepts dependencies which are known to exist.

Before executing a request, the dependencies are used to order the requests accordingly and thereby the execution ensures that conflicting requests are executed in the same order on all replicas. In case of cyclic dependencies between requests, the whole cycle is sorted deterministically and afterwards executed in this order. To bound the required state, the execution only processes a limited number of requests at a time. If a request cycle exceeds this limit, then it is deterministically cut into smaller parts.

In addition, replicas periodically create new checkpoints to garbage collect old requests. As each replica can independently assign requests to its sequence numbers, replicas have to coordinate at which point to create a new checkpoint. For this each replica periodically proposes a checkpoint request, which divides all other requests into before and after, therefore allowing the request execution to create a consistent checkpoint covering the same requests on each replica.

Our evaluation on Amazon EC2 shows that ISOS for low conflict rates is able to provide response times similar to the best PBFT configuration for each region, but for all regions at the same time. That is, the client-perceived response time improves for many clients. In addition, the egalitarian design avoids performance variations due to changes of the leader replica. For large requests, ISOS is able to spread the work of distributing requests across all replicas and thereby significantly outperform PBFT.

Cloud-Based Hierarchical Replication

The second major contributor to the response time for a request are the communication steps necessary for the agreement between the replicas themselves. It is possible to minimize their cost in terms of latency by placing all replicas at the same location, such that they can communicate with very low latency. However, this increases the risk that all replicas fail at the same time.

SPIDER makes use of the properties of modern cloud environments to resolve this dilemma. We split the replicas into an agreement group and multiple execution groups, which are each located in a cloud region. The replicas within a region can communicate with each other with low latency but are placed in different availability zones, which are

engineered by the cloud providers to minimize the risk of correlated failures. The size of the agreement group depends on the used agreement protocol, whereas the execution groups always consist of only $2f + 1$ replicas each. This simplifies the deployment as most cloud regions consist of only three availability zones [19, 101, 159]. The communication between groups uses IRMCs which allow replicas to exchange messages, but prevent faulty replicas from sending manipulated messages by requiring that at least $f + 1$ replicas vouch for the correctness of each message.

A client submits its request to its local execution group, which forwards it to the agreement group using a client-specific subchannel. Requests that arrive at the agreement group are then totally ordered by the agreement protocol. Afterwards the ordered requests are distributed using the IRMCs to all execution groups, which execute the requests and return the results if the client is in the same region. If a weakly consistent reply to a read request is sufficient for the client, then the request can be processed locally by an execution group. This provides very low response times by removing the need for wide-area communication.

SPIDER periodically creates checkpoints to bound the state required at each replica. This process is coordinated using the flow-control mechanism provided by the IRMCs, which bounds the message queue in these channels, but also ensures that requests are only garbage collected after they are no longer required for another group.

We present two different implementations of the IRMC abstraction with different trade-offs regarding implementation complexity and message overhead.

Our evaluation shows that SPIDER offers lower response times to clients than PBFT and HFT. Using batching in the agreement protocol and for the IRMCs, SPIDER can provide a throughput of several thousand requests per second. As all replicas of the agreement group are located in the same region, SPIDER provides similar response time independent of the current leader replica and thereby reduces performance variations.

Concurrent Checkpointing

As third protocol step, the ordered requests have to be executed. In order to bound the required state of the system, replicas have to periodically create checkpoints to garbage collect old requests. To tolerate Byzantine faults, all replicas have to capture a consistent snapshot of the application after processing the request for the same sequence number. However, creating a consistent snapshot also requires pausing the request execution. For applications with multiple gigabytes of application state, this can result in notable spikes in the response time.

DFC minimizes these delays by starting the state capture earlier on and running it concurrently to the request execution. This results in the creation of a fuzzy snapshot. In addition, replicas collect a modification list with information about changes to the application state that were made in the meantime. Together with the fuzzy snapshot the list is combined into a regular checkpoint. The latter step runs concurrently to the request execution to avoid delays. For efficiency, the start of the state capture is adjusted dynamically based on the time required to capture the last checkpoint.

We present two variants of DFC. Firstly, a basic copy-after-write variant, which tracks changed state objects and captures an updated copy of them when reaching the

checkpoint sequence number. To make a fuzzy snapshot deterministic again, captured objects are replaced with their latest version. Secondly, an update-based variant, which uses application-specific methods to capture a fuzzy snapshot of the whole application state and collect updates for each state modification to further reduce delays. The fuzzy checkpoint is afterwards updated by applying the collected updates. Both variants can be used to create differential checkpoints, which only capture state objects that have changed since the last checkpoint, in order to minimize the amount of copied data.

The described variants reduce delays introduced by checkpointing, for example, a 3 GiB application state from multiple seconds to less than a second. Capturing differential checkpoints further decreases the delay. Overall, fewer execution pauses can slightly improve the throughput and virtually eliminate client-visible response-time spikes.

7.2. Outlook

In the following we sketch possible directions for future research. In particular, we suggest methods to scale SPIDER to large numbers of execution groups and how to combine DFC and ISOS.

7.2.1. Scaling Hierarchical Fault Tolerance

In SPIDER, to offer optimal response times for weakly consistent read requests to more clients, it is necessary to add further execution groups. As the agreement group is responsible for distributing the ordered requests to all execution groups, it can become a bottleneck with a growing number of execution groups.

The IRMC-SC implementation already reduces the transmission overhead by collecting a certificate for each message at the sender side and then forwarding each message only once to each receiver. However, to minimize wide-area transmission costs between groups, ideally only a single copy of the message is sent over a wide-area connection, followed by locally distributing the message within the receiver group, similar to the approach used by Amir et al. [20]. To apply this idea to SPIDER, the replicas have to agree on which pair of replicas is responsible for forwarding the message between groups to be able to replace a faulty pair if necessary. As the execution groups are too small to run a Byzantine agreement protocol, the agreement group hosts an additional agreement protocol instance that runs an application exclusively used to manage the IRMC configuration. That is, for each IRMC instance, this application manages the information which replica pair is currently responsible for forwarding messages. Once at least $f_r + 1$ receiver replicas report an issue with the message forwarding, then the application switches to the next replica pair and informs all involved replicas that the configuration has changed. These replicas then retrieve the updated configuration.

A central configuration application also provides opportunities for further optimization. Using the global view of which agreement replica forwards requests to how many execution groups, it can ensure that the work to distribute the ordered requests is equally spread across multiple sender replicas to improve the throughput.

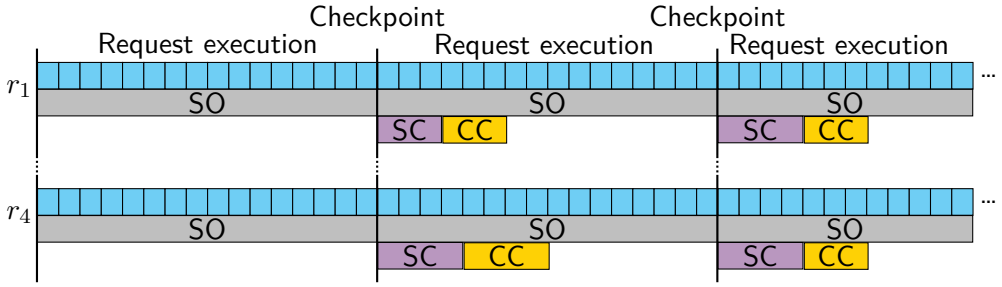


Figure 7.2: Schematic overview of creating a differential fuzzy checkpoint with a rollback-based approach. The checkpoint creation proceeds in the state observation (SO), state capture (SC) and checkpoint completion (CC) phases.

With an increasing number of execution groups, it becomes more and more likely that the transmission to one group is routed in proximity to another group. Thus, tasking the latter group with forwarding the ordered requests to the former one, adds little extra latency but reduces the work for the agreement group. In order to forward requests, the corresponding groups have to set up IRMCs with each other to distribute the ordered requests. The decision which execution groups forward messages to another one is made centrally by the configuration application at the agreement group. A forwarding execution group may only confirm that requests can be garbage collected, once all recipient groups have done so too. To detect that a forwarding group has fallen behind, the recipient groups additionally maintain an IRMC to the agreement group to learn about the current protocol progress.

7.2.2. Fuzzy Checkpoints for Egalitarian Fault Tolerance

The state capture phase used by DFC has to start sufficiently early before reaching the checkpoint sequence number. This works well for SPIDER, which uses a single global sequence number space and thereby allows selecting a suitable starting point. In contrast, in ISOS each replica has its own sequence number space and the creation of checkpoints is triggered via checkpoint requests. This complicates the selection of a suitable starting point for the concurrent state capture phase.

Instead of collecting a fuzzy snapshot earlier on and updating it, we suggest collecting the fuzzy snapshot afterwards and rolling it back as shown in Figure 7.2. The state capture runs concurrently to the request execution and starts after reaching the checkpoint sequence number. As a result, the modification list has to track information on how to roll back the fuzzy snapshot to the state at the checkpoint sequence number. Similar to DFC, the checkpoint completion phase which now rolls back the fuzzy snapshot, runs concurrently to the request execution to avoid delays.

We propose two variants of the application interface. Firstly, using the copy-on-write variant, the application has to inform the replication library about changed state objects before modifying them in order to allow the library to immediately create a copy of the state object. An object is only copied if this is the first modification since the checkpoint

capture has started. The fuzzy snapshot is then rolled back by replacing modified state objects with their unmodified copies.

Secondly, the undo variant collects application-specific undo objects, which can be used to roll back state objects. While the state capture is active, the application has to create these undo objects which afterwards are applied in reverse order to roll back the fuzzy snapshot to the expected checkpoint state. Similar to updates in DFC, an undo object must work at the granularity of individual objects as the fuzzy snapshot can contain state objects captured at different points in time.

Differential checkpointing works by tracking changed objects during the state observation phase, which is active for the full duration between two checkpoints. The state capture phase then only copies changed objects. As an optimization it is sufficient during the state capture phase to only collect changes for these changed objects.

The main benefit of this approach is that it is no longer necessary to estimate a suitable starting sequence number for the state capture phase. Instead, the state capture phase just runs until it is complete. Only if the state capture does not complete before reaching the next checkpoint sequence number, then the request execution still has to block.

The checkpoint requests in Isos can trigger the creation of checkpoints in quick succession. To prevent the just described execution delays, checkpoint requests should be ignored if less than k requests have been executed since creating the last checkpoint. As the agreement window for each request coordinator contains at least $2k$ sequence numbers, each request coordinator is still able to trigger the creation of a new checkpoint.

7.3. Concluding Remarks

The response time of a state-machine replication protocol is partially determined by the number of protocol phases, which for a certain number of replicas and faults have to adhere to certain theoretical lower bounds. But even then it is possible to reduce the cost of these phases, for example, by differentiating between local-area and wide-area communication. This allows reducing the latency for some protocol phases down to nearly zero, by turning them into local-area communication as suggested by this thesis, which in turn can also reduce the overall response time. Periodic operations at replicas such as creating checkpoints for garbage collection have the potential to introduce significant delays into these protocol phases. By running the checkpoint concurrently to the normal request processing, protocols can avoid these delays.

A

Safety and Liveness Proof for Egalitarian Fault Tolerance

This chapter is a partially revised version of the pseudocode and proofs presented in the appendix of the paper [88] of which I am the main author.

In Appendix A.1 we present the properties provided by ISOS as described in Chapter 4. Afterwards we prove these properties, first the agreement-specific parts in Appendix A.2 and then the execution in Appendix A.3. As last step, the checkpointing mechanism is integrated into the proof in Appendix A.4.

A.1. Properties

We show that ISOS provides the following properties. The Consistency, Execution Consistency and Agreement Liveness properties are based on those used by EPaxos [162].

- **Validity**: Only correctly signed client requests are executed.
- **Consistency**: Two correct replicas commit the same request and dependencies for a slot.
- **Execution Consistency**: Two conflicting requests are executed in the same order on all correct replicas.
- **Linearizability**: If two conflicting 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 these requests in that order.
- **Agreement Liveness**: During synchronous phases a client request will eventually commit at all correct replicas.
- **Execution Liveness**: During 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.

A message m signed by replica r_i is denoted as $\langle m \rangle_{\sigma_{r_i}}$. We make the following standard assumptions regarding cryptography (cf. Section 2.1.5). 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 or digest of a message m calculated using a collision-resistant hash function, that is, it must be virtually impossible to find two arbitrary messages m and m' with identical digest.

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 per slot in which the message type was sent has to be retransmitted.

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

A.2. Agreement

We start by presenting the pseudocode of the agreement, which includes the fast-path optimization from Section 4.7.1. Afterwards we prove the validity and consistency properties which only involve the agreement part of the protocol.

A.2.1. Pseudocode

Variables at each replica:
$p[s_j]$ // DepPropose *for agreement slot* s_j*, includes fast-path quorum* F
$pr[s_j]$ // Request *for* DepPropose *of agreement slot* s_j
$v[s_j][f_i]$ // DepVerify *for slot* s_j *from follower* f_i
$step[s_j] \in$ {init,proposed,fp-verified,fp-committed, rp-verified,rp-prepared,rp-committed,view-change}
$view[s_j]$ // *View number for slot* s_j*, initially* $view[s_j] := -1$
$views[s_j][r_i]$ // *Highest view number for slot* s_j *received from replica* r_i
$cert[s_j]$ // *Latest own certificate for slot* s_j
$exec[s_j]$ // *Tuple* $\langle r, D \rangle$ *of committed request* r *and its dependency set* D *for slot* s_j
$\Delta_{propose} := 2\Delta$; $\Delta_{commit} := 8\Delta$; $\Delta_{vc} := 3\Delta$; $\Delta_{vc-commit} := 3\Delta$; $\Delta_{query-exec} := 4\Delta$ // Δ_{vc} *is modified when adding checkpointing support*

Fast Path

11 **Request coordinator** co **receives new** r := $\langle \text{REQUEST}, w, t_c \rangle$:
12 assert r is a new request and is correctly signed
13 s_j := $\langle co, sc_j \rangle$ *// Smallest free slot*
14 D := `conflicts`(r)
15 F := Quorum of $2f$ followers
16 dp := $\langle \langle \text{DEPPROPOSE}, s_j, h(r), D, F \rangle_{\sigma_{co}}, r \rangle$
17 $\langle p[s_j], pr[s_j] \rangle$:= dp
18 $step[s_j]$:= `proposed`
19 Broadcast dp to all replicas
20 Start commit timeout Δ_{commit} for slot s_j

22 **Follower** f_i **receives** dp := $\langle \langle \text{DEPPROPOSE}, s_j, h(r), D, F \rangle, r \rangle$ **from** co:
23 `pre:` $step[s_j] =$ `init`
24 assert F is a valid fast-path quorum
25 assert $pr[s_j] = \emptyset \wedge s_j.co = co$ *// First* DEPPROPOSE *from coordinator*
26 `wait`$(D \cup s_{j-1})$ *// s_{j-1} is the previous slot from coordinator co*
27 if $p[s_j] = \emptyset$:
28 Start commit timeout Δ_{commit} for slot s_j
29 Start propose timeout $\Delta_{propose}$ for slot s_j
30 $p[s_j]$:= dp.DEPPROPOSE
31 if $r \neq \perp$: *// Check whether the full request is included*
32 assert r correctly signed
33 D_{f_i} := `conflicts`(r)
34 $pr[s_j]$:= r
35 $step[s_j]$:= `proposed`
36 if $f_i \in F$:
37 Broadcast $\langle \text{DEPVERIFY}, s_j, h(dp), D_{f_i} \rangle_{\sigma_{f_i}}$

39 **Replica** r_i **receives** m := $\langle \text{DEPVERIFY}, s_j, h(dp), D_{f_i} \rangle$ **from** f_i:
40 `pre:` $step[s_j] =$ `proposed` $\wedge\ h(p[s_j]) = h(dp)$
41 assert $v[s_j][f_i] = \emptyset$ *// First* DEPVERIFY *from follower*
42 assert $f_i \in p[s_j].F$ *// Follower is in fast-path quorum*
43 `wait`(D_{f_i})
44 $v[s_j][f_i]$:= m
45 $\vec{dv}$:= $\{v[s_j][f_i] \,|\, \forall f_i \in p[s_j].F\}$
46 if $|\vec{dv}| = 2f$:
47 Stop propose timeout $\Delta_{propose}$ for slot s_j
48 D := $\cup D_{f_i} \in \vec{dv}$
49 *// Every dependency is reported by at least $f+1$ followers*
50 if $\{d \in D \,|\, |\{f_i \,|\, \forall f_i : d \in v[s_j][f_i].D_{f_i}\}| \geq f+1\} = D$:
51 *// Slot s_j is now fp-verified at replica r_i*

52 $step[s_j]$:= `fp-verified`
53 Broadcast $\langle \text{DEPCOMMIT}, s_j, h(\vec{dv}) \rangle_{\sigma_{r_i}}$
54 else:
55 Enter reconciliation path, stop participating in fast path

57 **Replica r_i receives $\langle \text{DEPVERIFY}, s_j, *, * \rangle$ from $f+1$ replicas:**
58 Start commit timeout Δ_{commit} for slot s_j

60 **Replica r_i receives $\langle \text{DEPCOMMIT}, s_j, h(\vec{dv}) \rangle$ with identical $h(\vec{dv})$ from $2f+1$ replicas:**
61 **pre:** $step[s_j] = \texttt{fp-verified} \wedge h(\{v[s_j][f_i] \mid \forall f_i \in p[s_j].F\}) = h(\vec{dv})$
62 Stop propose/commit timeout $\Delta_{propose}$ and Δ_{commit} for slot s_j
63 D := $\cup D_{f_i} \in \vec{dv}$
64 $exec[s_j]$:= $\langle pr[s_j], D \rangle$
65 Forward $\langle pr[s_j], D, s_j \rangle$ to execution

67 VOID **wait**(DEPENDENCYSET D):
68 for $d \in D$:
69 sleep until either:
70 $p[d] \neq \emptyset$ *// Received a valid* DEPPROPOSE
71 received $f+1$ DEPVERIFYs
72 received $f+1$ VIEWCHANGEs

74 DEPENDENCYSET **conflicts**(REQUEST r):
75 *// The DependencySet must use the compact dependency encoding, see* Corollary A.3.9
76 return $\{s_i | \forall s_i, pr[s_i] \neq \emptyset : conflict(pr[s_i], r)\}$

Reconciliation Path

---

77 **Timeout $\Delta_{propose}$ for slot s_j expires:**
78 Broadcast $\langle p[s_j], \perp \rangle$ to all replicas *// Only distribute nil value*

80 **Timeout Δ_{commit} for slot s_j expires:**
81 Move to new view $v_{s_j} + 1$

83 **Enter reconciliation path for slot s_j at replica r_i:**
84 $step[s_j]$:= `rp-verified`
85 $\vec{dv}$:= $\{v[s_j][f_i] \mid \forall f_i \in p[s_j].F\}$
86 Broadcast $\langle \text{PREPARE}, s_j, view[s_j], h(\vec{dv}) \rangle_{\sigma_{r_i}}$

88 **Replica r_i receives $\langle \text{PREPARE}, s_j, v_{s_j}, h(\vec{dv}) \rangle$ with identical $h(\vec{dv})$ from $2f+1$ replicas:**

89 **pre:** $step[s_j] = \texttt{rp-verified} \wedge view[s_j] = v_{s_j}$
$\wedge\ h(\{v[s_j][f_i] \mid \forall f_i \in p[s_j].F\}) = h(\vec{dv})$
90 $step[s_j]$:= `rp-prepared`
91 Broadcast $\langle \textsc{Commit}, s_j, v_{s_j}, h(\vec{dv}) \rangle_{\sigma_{r_i}}$

93 **Replica r_i receives** $\langle \textsc{Commit}, s_j, v_{s_j}, h(\vec{dv}) \rangle$
with identical $h(\vec{dv})$ **from** $2f+1$ **replicas**:
94 **pre:** $step[s_j] = \texttt{rp-prepared} \wedge view[s_j] = v_{s_j}$
$\wedge\ h(\{v[s_j][f_i] \mid \forall f_i \in p[s_j].F\}) = h(\vec{dv})$
95 $step[s_j]$:= `rp-committed`
96 Stop commit timeout Δ_{commit} for slot s_j
97 D := $\cup D_{f_i} \in \vec{dv}$
98 $exec[s_j]$:= $\langle pr[s_j], D \rangle$
99 Forward $\langle pr[s_j], D, s_j \rangle$ to execution

View Change

100 **Move to new view v_{s_j} for slot s_j at replica r_i:**
101 if propose timeout $\Delta_{propose}$ for slot s_j is active: trigger its expiry
102 Stop commit/VC timeout Δ_{commit} and Δ_{vc} for slot s_j
103 dp := $\langle p[s_j], pr[s_j] \rangle$; $\vec{dv}$:= $\{v[s_j][f_i] \mid \forall f_i \in p[s_j].F$ if $p[s_j] \neq \perp\}$
104 *// Update certificate if current view fp-verified / rp-prepared*
105 if $step[s_j] \in$ {`fp-verified`, `fp-committed`}:
106 $cert[s_j]$:= $\langle \text{FPC}, dp, \vec{dv}, -1 \rangle$
107 else if $step[s_j] \in$ {`rp-prepared`, `rp-committed`}:
108 $\vec{prep}$:= set of $2f+1$ PREPAREs with $h(\vec{dv})$
109 $cert[s_j]$:= $\langle \text{RPC}, dp, \vec{dv}, \vec{prep}, view[s_j] \rangle$
110 $view[s_j]$:= v_{s_j}
111 $views[s_j][r_i]$:= v_{s_j}
112 $step[s_j]$:= `view-change`
113 Start query execute timeout $\Delta_{query-exec}$ for slot s_j
114 Broadcast $\langle \textsc{ViewChange}, s_j, v_{s_j}, cert[s_j] \rangle_{\sigma_{r_i}}$

116 **Replica r_i receives** $\langle \textsc{ViewChange}, s_j, v_{s_j}, * \rangle$ **from** r_k:
117 **pre:** $v_{s_j} > views[s_j][r_k]$ *// View number of a replica must only increase*
118 $views[s_j][r_k]$:= v_{s_j}
119 vn := $f+1$-highest in $\{views[s_j][r_l] \mid \forall r_l\}$ *// Move to f+1-highest known view*
120 if $vn > view[r_i]$:
121 Move to new view vn *// Sends new* VIEWCHANGE *message*

123 co := $(s_j.co + \max(0, v_{s_j}))$ mod N *// Determine View-change coordinator*

125 **View-change coordinator** co **for view** v_{s_j} **receives valid** $VCS := \{\langle \text{VIEWCHANGE}, s_j, v_{s_j}, * \rangle\}$ **from** $2f+1$ **replicas**:
126 **pre:** $step[s_j] = \texttt{view-change} \wedge view[s_j] = v_{s_j}$
127 assert $\forall VC \in VCS : VC$ is valid $\wedge$ $(VC.cert = \emptyset \vee VC.cert.view \leq VC.v_{s_j})$
128 select dp, $\vec{dv}$ from
129 reconciliation-path result for highest view — if RPC certificate exists
130 fast-path result — if FPC certificate exists
131 NO-OP — otherwise
132 Broadcast $\langle \text{NEWVIEW}, s_j, v_{s_j}, dp, \vec{dv}, VCS \rangle_{\sigma_{co}}$

134 **Replica** r_i **receives valid** $\langle \text{VIEWCHANGE}, s_j, v_{s_j}, * \rangle$ **from** $2f+1$ **replicas**:
135 **pre:** $step[s_j] = \texttt{view-change} \wedge view[s_j] = v_{s_j}$
136 Start VC timeout Δ_{vc} for slot s_j
137 Stop query execute timeout $\Delta_{query-exec}$ for slot s_j

139 **Timeout** Δ_{vc} **for slot** s_j **expires:**
140 Move to new view $v_{s_j} + 1$

142 **Replica** r_i **receives** $\langle \text{NEWVIEW}, s_j, v_{s_j}, dp, \vec{dv}, VCS \rangle$ **from** co:
143 **pre:** $step[s_j] = \texttt{view-change} \wedge view[s_j] = v_{s_j}$
144 assert co is view-change coordinator for view v_{s_j}
145 assert $\forall VC \in VCS : VC$ is valid
146 assert $dp, \vec{dv}$ correctly selected based on VCS
147 $\langle p[s_j], pr[s_j] \rangle := dp$
148 $v[s_j][*] := \emptyset$ // *Cleanup* DEPVERIFY*s*
149 $\forall dv \in \vec{dv} : v[s_j][dv.f_i] := dv$
150 if $s_j.i = r_i \wedge dp = \text{NO-OP}$:
151 Permute fast-path quorum
152 Re-propose request in a new slot
153 Start commit timeout Δ_{commit} with reduced timeout $\Delta_{vc-commit}$ for slot s_j
154 Enter reconciliation path

156 **Timeout** $\Delta_{query-exec}$ **for slot** s_j **expires:**
157 Broadcast $\langle \text{QUERYEXEC}, s_j \rangle_{\sigma_{r_i}}$ to all replicas

159 **Replica** r_i **receives** $\langle \text{QUERYEXEC}, s_j \rangle$ **from replica** r_j:
160 **pre:** $exec[s_j] \neq \emptyset$
161 $\langle dp, D \rangle := exec[s_j]$
162 Send $\langle \text{EXECUTE}, s_j, dp, D \rangle_{\sigma_{r_i}}$ to replica r_j

164 **Replica** r_i **receives** $\langle \text{EXECUTE}, s_j, dp, D \rangle$ **from** $f+1$ **replicas:**
165 **pre:** $exec[s_j] = \emptyset$

166 $exec[s_j]$:= $\langle dp, D \rangle$
167 Forward $\langle dp, D, s_j \rangle$ to execution with dependencies D

A.2.2. Validity

Theorem A.2.1 (Validity)**.** *Only correctly signed client requests are executed.*

Proof. Only committed requests are executed. A client request is passed to the execution either in Line 65 or 99 via the variable $pr[s_j]$ or received via EXECUTE messages in Line 167. $pr[s_j]$ is set in

- Lines 17 and 34: The validity of the client request was verified before setting the variable.
- Line 147: The value can be a NO-OP request 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 (Line 37), the request must be correct. Otherwise, a correct replica must have created two DEPVERIFYs which yields a contradiction.

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

The NO-OP request is skipped during execution and thus only correctly signed client requests are executed. □

A.2.3. Consistency

Theorem A.2.2 (Consistency)**.** *Two correct replicas commit the same request and dependencies for a slot.*

We first establish some additional terminology:

Definition A.2.3. A slot s_j is *verified* if a correct replica collects a valid DEPPROPOSE dp, $2f$ valid DEPVERIFYs from different replicas with matching $h(dp)$ and each DEPVERIFY is from a replica in the fast-path quorum $dp.F$.

Definition A.2.4. A slot s_j is *fp-verified* if a correct replica *verified* it and each dependency in the DEPVERIFYs occurs at least $f + 1$ times.

Definition A.2.5. A slot s_j is *fp-committed* if a correct replica collects $2f + 1$ DEPCOMMITs from different replicas with matching $h(\vec{dv})$.

Remark A.2.6. Note that *fp-committed* implies *fp-verified* as DEPCOMMITs are only sent by replicas which *fp-verified* the slot.

Definition A.2.7. A slot s_j is *rp-verified* if a correct replica *verified* it and it is not *fp-verified*.

Definition A.2.8. A slot s_j is *rp-prepared* if a correct replica collects $2f + 1$ PREPAREs from different replicas with matching $h(\vec{dv})$.

Definition A.2.9. A slot s_j is *rp-committed* if a correct replica collects $2f + 1$ COMMITs from different replicas with matching $h(\vec{dv})$.

Remark A.2.10. *rp-committed* implies *rp-prepared*. *rp-prepared* implies *rp-verified*.

Definition A.2.11. A slot s_j is *committed* if a correct replica *fp-committed* or *rp-committed* it.

We first show the following auxiliary lemmas.

Lemma A.2.12. *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 (Line 88). This requires $f + 1$ correct replicas to have entered the reconciliation path (via Lines 55 and 83). To be *fp-committed*, another replica must have received $2f + 1$ DEPCOMMITs. This requires a correct replica to send a DEPCOMMIT (Line 53) and to enter the reconciliation path (Line 55). However, those are mutually exclusive, which yields a contradiction. □

Lemma A.2.13. *The content of a reconciliation-path certificate (RPC) cannot be manipulated without detection.*

Proof. As each protocol phase includes hashes of the previous phase, faulty replicas can only manipulate the last round of messages that is included in a certificate without immediately invalidating the certificate. For an RPC only the PREPAREs can be manipulated, however, as the certificate must include PREPAREs from correct replicas, the certificate must still prove the correct DEPPROPOSE and DEPVERIFYs. □

Lemma A.2.14. *A faulty replica can only create a manipulated but valid fast-path certificate (FPC) if not* fp-committed.

Proof. A faulty replica could try to construct a faulty FPC using manipulated DEPVERIFYs, which allows 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 (Line 50). This requirement is also necessary for an FPC to be valid.

Only a single DEPPROPOSE can be *verified* as it requires the existence of $2f$ matching DEPVERIFYs and 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 prove 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 at least $f + 1$ DEPVERIFYs.

- 0 occurrences: A manipulated FPC can either not include the dependency in which case the FPC is effectively 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.2.15. *A manipulated FPC can only be used if neither* fp-committed *nor* rp-committed.

Proof. If *fp-committed*, then according to Lemma A.2.14 no manipulated but valid 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 (Lemma A.2.12) and *rp-committed* implies *rp-prepared*, no FPC from a correct replica can exist. □

Now we prove Theorem A.2.2 by contradiction:

Proof. **Case 1:** A replica r_i commits $\langle r, D, s_j \rangle$ via the fast-path (Line 65). r is the request committed with dependencies D for slot s_j.

- Case 1.1: Another replica r_k commits $\langle r', D', s_j \rangle$ with $r \neq r' \vee D \neq D'$ via the fast-path.
  - Case $r \neq r'$:

    Proof. Then $p^i[s_j] \neq p^k[s_j]$, as $h(r) \neq h(r')$ due to $r \neq r'$. $h(p[s_j])$ is part of the DepVerifys. Therefore, $h(\vec{dv})$ must differ. Then the replicas 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}, replicas r_i and r_k must use different $h(\vec{dv})$. Now, the proof of the previous case applies. □

- Case 1.2: Replica r_k commits in view -1 via the reconciliation path.

  Proof. Then *rp-committed* holds. This implies *rp-prepared* which according to Lemma A.2.12 conflicts with *fp-committed*, yielding a contradiction. □

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

- The cases are exhaustive.

Case 2: A replica r_i commits $\langle r, D, s_j \rangle$ via the reconciliation path in $view = -1$ (Line 99).

- Case 2.1: r_k commits $\langle r', D', s_j \rangle$ with $r \neq r' \vee D \neq D'$ via the fast-path.

  Proof. See Case 1.2. □

- Case 2.2: r_k commits $\langle r', D', s_j \rangle$ with $r \neq r' \vee D \neq D'$ 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 (Line 83). □

- Case 2.3: r_k commits $\langle r', D', s_j \rangle$ with $r \neq r' \vee D \neq D'$ in $view \geq 0$ via the reconciliation path.
  Deferred to Case 3.

- The cases are exhaustive.

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

Proof. We prove this by induction: Once a replica commits $m := \langle r, D, s_j \rangle$, with $r \neq r' \vee D \neq D'$, 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 or commits in $view'$. A correct replica only decides a result in view $view'$ after receiving a valid NEWVIEW (Line 154). No manipulated RPC and FPC can be used according to Lemma A.2.13 and A.2.15.

- 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, which must be selected by the view change. The FPC contains 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 RPC is selected, as 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.
- The cases are exhaustive.

Induction step: $view' > view + 1$:
To commit a slot, $2f + 1$ replicas have to send a DepCommit or Commit. One ViewChange message 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 (Line 105-109), 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 (Line 128-131).
- 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.
- The cases are exhaustive.

□

Case 4: A replica r_k commits $\langle dp, D, s_j \rangle$ after receiving $f + 1$ valid and matching $\langle \text{Execute}, s_j, *, dp, D \rangle$ messages (Line 156-167). This case allows lagging replicas to catch up and learn the agreement result as described in Section 4.3.4.

Proof. At least one of the Execute messages is from a correct replica, which either has committed the slot itself such that the other cases apply to that replica. Or the correct replica has learned from another correct replica that the slot was committed. □

The cases are exhaustive. □

A.3. Execution

This section first presents the execution pseudocode before providing proofs for the remaining protocol properties.

A.3.1. Pseudocode

We first introduce some additional notation.

The relation $conflict(a, b)$ states whether two requests a and b conflict with each other.

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. That is, for slot v, its sequence number is $s_i = \langle v.i, v.seq \rangle$. For two slots v_1 and v_2 with $v_1.i = v_2.i$, we use

$v_1 < v_2$ as shorthand for $v_1.seq < v_2.seq$. The dependencies that were committed for a slot s are given by $deps(s)$. If the context expects a request, we use slot v to refer to its committed request. Calling "Forward $\langle pr[s_j], D, s_j\rangle$ to execution" in the agreement pseudocode, informs the execution about a slot v with request $v = pr[s_j]$, dependencies $deps(v) = D$ and sequence number $\langle v.i, v.seq\rangle = s_j$.

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, $a \rightsquigarrow b \Leftrightarrow a \to b \vee a \to v_1 \to \ldots \to v_n \to b$, with $n \in \mathbb{N}$ and unique v_i.

A directed graph $G = (V, E)$ consists of a set V of vertices and a set of directed edges E. In a *dependency graph*, the graph contains slots v_i (vertices) and edges $v_j \to v_k$ between those slots. For brevity, we write $v \in G$ when referring to the slots/vertices of a graph, instead of the more verbose $v \in G.V$ and $(v_j \to v_k) \in G$ to refer to edges in the graph. Similarly, a comparison of a graph with a set of vertices, only compares against the vertices of the graph.

$rdeps(v)$ calculates a dependency graph starting from a slot v. By construction all slots and edges in the graph are reachable from v. $rdeps_{exp}(v)$ calculate a dependency graph that is limited to slots within the expansion limit.

```
168 Variables at each replica:
169 k                                                  // Size of execution window
170 committed, executed
        // Sets containing all slots that have been commmitted or executed so far
171 rhist[*] := ⊥   // History variable for dependency graph calculation. For each executed
        slot v, it stores the dependency graph for v as it existed when v was executed.
172 // Helper functions
173 exp(r_i) := min{v_min | v_min ∉ executed ∨ v_min.i = r_i}
        // Root node for replica r_i. This is the first not executed slot for replica r_i, that is,
        the lower bound of the execution window
174 exp_k := {v | v.seq < exp(v.i) + k}   // All slots that are currently below the expansion
        limit, that is, all executed slots and those inside the execution windows
```

Request Execution

```
175 // Calculate dependency graph for slot v_in. The resulting graph includes all slots
        reachable from v_in and the edges between them
176 DependencyGraph rdeps(Slot v_in):
177   G' := ({v_in}, {}); G := ({}, {})                                  // G = (V, E)
178   while G ≠ G':
179     G := G'
180     for v ∈ G:
181       if v ∉ executed:
182         G'.V := G'.V ∪ deps(v)
183         G'.E := G'.E ∪ {(v → d) | d ∈ deps(v)}
184       else:
185         G' := G' ∪ rhist[v]
```

```
186   return G

188 // Calculate dependency graph for slot v_in. Excludes slots outside the execution window
189 DependencyGraph rdeps_exp(Slot v_in):
190   G' := ({v_in} ∩ exp_k, {}); G := ({}, {})                    // G = (V, E)
191   while G ≠ G':
192     G := G'
193     for v ∈ G:
194       if v ∉ executed:
195         G'.V := G'.V ∪ {d | d ∈ deps(v) ∧ d ∈ exp_k}
196         G'.E := G'.E ∪ {(v → d) | d ∈ deps(v) ∧ d ∈ exp_k}
197       else:
198         G' := G' ∪ rhist[v]
199   return G

201 while true:
202   Update slots committed in the meantime
203   // Repeat loop until no further suitable v exists
204   // Process all slots in the execution window, whose dependency graph is committed and
          within the expansion limit
205   for all v ∈ (exp_k \ executed) ∧ rdeps(v) ⊆ (committed ∩ exp_k):
206     s⃗c := find not executed strongly connected components in rdeps(v) in inverse
            topological order
207     for sc ∈ s⃗c:
208       // Standard execution case
209       execute(sc, rdeps(sc))
210   // Process all slots in the execution window, whose dependency graph part that is inside
          the execution window is committed
211   for all v ∈ (exp_k \ executed) ∧ rdeps_exp(v) ⊆ committed:
212     s⃗c := find not executed strongly connected components in rdeps_exp(v) in inverse
            topological order
213     // Unblock execution case
214     execute(s⃗c[0], rdeps_exp(s⃗c[0]))

216 void execute(SCC v⃗, DependencyGraph G):
217   for v ∈ sort(v⃗):
218     Execute request v and reply to client
219     rhist[v] := G

221 Slots sort(SCC v⃗):
222   return v⃗ sorted by sequence numbers v.seq and use replica ID v.i as tie breaker
```

A.3.2. Execution Consistency

Similar to the Execution Consistency property in EPaxos [162], we show:

Theorem A.3.1 (Execution Consistency)**.** *Two conflicting requests are executed in the same order on all correct replicas.*

We first show that conflicting requests are connected by a dependency between each other, before showing that conflicting requests are executed in the same order on all replicas.

Lemma A.3.2. *If conflict(a, b) then, a has a dependency to b, that is, $b \in deps(a)$ or the other way around.*

Proof. For a request r, the dependencies are provided by one DepPropose and $2f$ DepVerify, that is, messages from $2f + 1$ replicas.

For requests a and b, due to the quorum intersection property, 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 $view = -1$. We now discuss what happens during a view change. When a replica *rp-prepares* the slot, then by construction its RPC must 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 no-op request is selected. As that request does not conflict with any other request (except CheckpointReq, which is added later on by the checkpointing support and is not relevant for now), 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.3.3 (SCC)**.** A strongly connected component (SCC) s [195] is a subset $s = (V_s, E_s)$ of a graph G, such that $V_s \subseteq G.V$ is a subset of the graph's slots, and that contains all edges connecting these slots $E_s = \{(v_i \to v_j) \in G.E \mid v_i, v_j \in V_s\}$. Like with a dependency graph, we use $v \in s$ to refer to some slot v that is part of the SCC s. The subset must additionally have the following properties.

For any two slots $\forall v_i, v_j \in s : v_i \rightsquigarrow v_j \land v_j \rightsquigarrow v_i$. That is, each slot in s must transitively depend on any other slot in s.

Additionally, no slot $v \in G, \notin s$ can be added to V_s such that the previous property still holds. It must not be possible to extend the SCC without losing its connectivity between all slots. That is, the SCC already has the maximum size.

Definition A.3.4 (Regular SCC)**.** A regular SCC is one that was executed via the standard case execution (Line 208).

Definition A.3.5 (SSCC)**.** A special-case strongly connected component (SSCC) is an SCC that was executed via the special case to unblock the execution (Line 213).

Definition A.3.6 (SCC trace)**.** A SCC trace t is a 0-based vector consisting of executed SCCs in the order of their execution. That is $t = [s_0, s_1, ..., s_n]$, with $t[0] = s_0$, and s_i are SCCs. We write t^i to refer to the trace belonging to a replica r_i.

For a trace t the function $flatten(t) = \{v \,|\, v \in s, s \in t\}$ returns a set of all slots contained in the trace t, where s is an SCC and v a slot.

Corollary A.3.7. *An SCC trace fully defines the order in which requests are executed. That is, all executed slots are part of the SCC trace. Every slot is only part of a single SCC.*

Proof. Slots can only be executed via $execute(\vec{v}, G)$, 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. Requests within an SCC are sorted before execution, which yields a stable order. □

Corollary A.3.8. *Note that the inverse topological sorting (Line 206 and 212) ensures that slots in an SCC can only depend on the SCC itself or earlier SCCs. Thus, when an SCC is executed, then all its dependencies have already been executed. We rely on this property in our following proofs.*

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

Corollary A.3.9. *The* compact dependency encoding *implicitly includes dependencies on all earlier slots of a replica. That is, a dependency from slot v_a to slot v_b ensures that $v_b \in deps(v_a) \Rightarrow cdeps(v_b) \subseteq deps(v_a)$ with $cdeps(v_b) = \{v | v.i = v_b.i \wedge v.seq \leq v_b.seq\}$.*

Corollary A.3.10. *Each slot v in an SCC s at replica i has the same $rdeps^i(v)$ or $rdeps^i_{exp}(v)$ when s gets executed. By definition $\forall v_1, v_2 \in s, v_1 \neq v_2 : v_1 \rightsquigarrow v_2 \wedge v_2 \rightsquigarrow v_1$. Thus, $rdeps^i(v_1) = rdeps^i(v_2)$. In the following we use $rdeps^i(v)$ and $rdeps^i(s)$ for slot $v \in SCC\ s$ interchangeably. This also applies to $rdeps^i_{exp}(s)$.*

In the following we show that two arbitrary SCC traces t^i and t^j from replicas r_i and r_j share certain properties. For this, we first prove a supporting Lemma that $rhist[v]$, which captures the dependency graph at the moment v is executed, is identical for slots v executed at both replicas ($v \in flatten(t^i) \cap flatten(t^j)$). That is, v was executed as part of the same SCC on both replicas.

Lemma A.3.11. *Assume replicas r_i and r_j have traces t^i and 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.*

We prove Lemma A.3.11 in multiple steps by induction. We first show that the Lemma holds for an empty trace, that slots are either executed via a regular SCC or an SSCC, that the Lemma holds when executing further regular SCCs, and also for further SSCCs.

Base case of Lemma A.3.11: $|t^i| = |t^j| = 0$:

Proof. $flatten(t^i) \cap flatten(t^j) = \emptyset$ □

We now show an auxiliary lemma that an SCC is consistently executed as regular SCC or SSCC on all replicas.

Lemma A.3.12. *If $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$ holds before executing a slot v_1 via the standard case, then $rdeps^i(v_1) = rdeps^j(v_1)$. Furthermore, if v_1 is executed as part of a regular SCC s at replica r_i and r_j (Line 208), then the SCC is identical at both replicas, that is, $s^i = s^j$ and $\forall v' \in s : rdeps^i(v') = rdeps^j(v')$.*

Proof. By construction, a slot v_1 in an SCC s is only executed after all slots of s were committed, therefore per (Agreement) Consistency for any committed slot v_x: $deps^i(v_x) = deps^j(v_x) = deps(v_x)$ are identical on all replicas. As by assumption no slot of the SCC s was executed before, $rdeps(v_1)$ uses the values from $deps(v_s)$ for slots v_s that are part of the SCC s (Line 182). As all dependencies of an SCC were executed before the SCC (cf. Corollary A.3.8), then these slots v_d must be $v_d \in flatten(t^i)$ and $\in flatten(t^j)$. Thus, the graph from $rhist[v_d]$ is used for executed slots (Line 185), which by assumption is identical on all replicas. Therefore, $rdeps^i(v_1) = rdeps^j(v_1) = rdeps(v_1)$.

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

Now assume that two regular SCCs at replica r_i and r_j have $s^i \neq s^j \wedge s^i \cap s^j \neq \emptyset$ (different SCCs, but with a common slot): 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 the dependency graph at r_i and r_j contains the same SCCs, that is, $v_1 \in s^j$, which yields a contradiction. Thus, $s^i = s^j$. □

Lemma A.3.13. *An SSCC $\hat{s}$ can only be executed iff $\forall v \in \hat{s} : exp(v.i) \in \hat{s}$.*

Proof. By Corollary A.3.9, $v_b \in deps(v_a) \Rightarrow cdeps(v_b) \subseteq deps(v_a)$. A dependency $v_b \in deps(v_a)$ can be omitted by $rdeps_{exp}(v)$ and consequently from $\hat{s}$ either if v_b is already executed (then $v_b < exp(v_b.i)$) or $\exists v_r = exp(v_b.i)$ with $v_b.seq - v_r.seq > k$. Due to Corollary A.3.9 $v_r \in deps(v_a)$. Thus, $v_r \in \hat{s}$ as otherwise it must already have been executed (all dependencies of an SSCC are executed first, Lines 205 and 212), which contradicts the definition of v_r. These considerations also apply to all other dependencies of slots in $\hat{s}$. Thus, an SSCC $\hat{s}$ can only be executed exactly at the moment where $\forall v \in \hat{s} : exp(v.i) \in \hat{s}$, that is, the relevant part of exp_k is defined by the SSCC. □

Lemma A.3.14. *If* $\forall v \in flatten(t^i) \cap flatten(t^j) : rhist^i[v] = rhist^j[v]$ *holds before executing a slot* v_1 *via the unblock execution case execution, then* $rdeps^i_{exp}(v_1) = rdeps^j_{exp}(v_1)$ *and* $rdeps^i(v_1) \not\subseteq exp^i_k, rdeps^j(v_1) \not\subseteq exp^j_k$*. Furthermore, if* v_1 *is executed as part of an SSCC* $\hat{s}$ *at replica* r_i *and* r_j *(Line 213), then the SSCC is identical at both replicas, that is,* $\hat{s}^i = \hat{s}^j$ *and* $\forall v' \in \hat{s} : rdeps^i_{exp}(v') = rdeps^j_{exp}(v')$*.*

Proof. Assume that $rdeps^i(v_1) \subseteq exp^i_k, rdeps^j(v_1) \subseteq exp^j_k$, then v_1 is always executed via the standard case (Line 208), such that Lemma A.3.12 applies. This prevents the unblock execution case from running, which yields a contradiction.

The dependency graph calculated by $rdeps_{exp}(v_1)$ depends on $deps(v)$, $rhist[v]$ and exp_k. $deps(v)$ is identical across replicas due to the (Agreement) Consistency property and $rhist[v]$ is identical across replicas as dependencies are executed first and thus this follows from the assumption.

This leaves showing that the parts of exp_k that influence the generated dependency graph are equivalent. According to Lemma A.3.13 the root nodes of an SSCC determine the relevant parts of exp_k. Thus, we show by induction that after a common starting point, that SSCCs executed by replicas r_i and r_j either use root nodes from disjunct sets of replicas, or that the SSCCs with overlapping replicas are identical.

Consider slots v from the latest SSCCs that were executed on both replica r_i and r_j. Per assumption these SSCCs must be equal as for them $rhist^i[v] = rhist^j[v]$, such that the same SCC must be calculated. Then let SSCC $\hat{s}^i$ be the next one to execute at replica r_i and SSCC $\hat{s}^j$ be the next one to execute at replica r_j.

Case 1: Assume that the root nodes of $\hat{s}^i$ and $\hat{s}^j$ belong to disjunct sets of replicas. Then we can consider each SSCC independently. Without loss of generality, we only consider $\hat{s}^i$. Then the previous observation applies that $\forall v \in \hat{s}^i : exp^i(v.i) \in \hat{s}^i$ and therefore $\hat{s}^i$ is determined by the already executed slots. Note that $\hat{s}^i$ and $\hat{s}^j$ cannot depend on each other, as that would contradict the assumption that the SSCCs so far have only executed at one replica.

Case 2: Assume that the root nodes of $\hat{s}^i$ and $\hat{s}^j$ overlap in at least one replica. We select one such replica r_o. Then $v_{ri} = exp^i(r_o) \in \hat{s}^i$ and $v_{rj} = exp^j(r_o) \in \hat{s}^j$. That is v_{ri} and v_{rj} are the root nodes for replica r_o from the perspective of replicas r_i and r_j. Without loss of generality, assume that $v_{ri} \leq v_{rj}$. We set $v_{mi} = \max\{v_x | v_x \in \hat{s}^i \wedge v_x.i = r_o\}$, which is the newest slot in $\hat{s}^i$ for replica r_o. Then either $v_{mi} < v_{rj}$ or $v_{ri} \leq v_{rj} \leq v_{mi}$.

- Case 2.1: $v_{mi} < v_{rj}$. Then by construction, $\hat{s}^j$ depends on v_{mi}, which either would have to be executed before, such that it would be covered by the common starting point, which contradicts that $v_{mi} \in \hat{s}^i$. Or if v_{mi} is not executed yet at replica r_j, then by definition of v_{rj} we know that $v_{rj} \leq v_{mi}$ which contradicts the assumption of the current case.

- Case 2.2: $v_{ri} \leq v_{rj} \leq v_{mi}$. Then a node $\exists v_x \in \hat{s}^j : v_x \rightarrow v_{ri}$, that is, a slot v_x in $\hat{s}^j$ depends on v_{ri}. And $v_{ri} \rightsquigarrow v_{rj}$. v_{ri} cannot execute before v_{rj}, and then by definition of v_{rj} it follows that $v_{ri} = v_{rj}$.

- The cases are exhaustive.

Then, v_{ri} and v_{rj} depend on the same slots, which per the previous cases must use the same root nodes. This applies to all root nodes of the SSCC. As a consequence, $\hat{s}$ is determined by any of its slots. That is, $\hat{s} = \hat{s}^i = \hat{s}^j$ and $\forall v \in \hat{s} : rdeps^i_{exp}(v) = rdeps^j_{exp}(v)$.

The cases are exhaustive.

□

Lemma A.3.15. *An SCC is either executed using the standard case execution on all replicas (regular SCC) or using the unblock execution case on all replicas (SSCC).*

Proof. According to Corollary A.3.7 a slot can only be executed once at a replica, thus it remains to be shown, that the slot is always executed via the same case.

Case 1: To arrive at a contradiction, assume that a regular SCC is executed via the unblock execution case.

For this, we must find a committed but not executed slot within the execution window, that is, a slot $v \in committed \cap exp_k \wedge v \notin executed$ that 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 211) and the second part ensures that the standard case execution does not apply (Line 205).
$\Leftrightarrow rdeps_{exp}(v) \subseteq committed \wedge (rdeps(v) \not\subseteq committed \vee rdeps(v) \not\subseteq exp_k)$
$\Leftrightarrow rdeps_{exp}(v) \subseteq committed \wedge (rdeps(v) \setminus rdeps_{exp}(v) \not\subseteq committed \vee rdeps(v) \not\subseteq exp_k)$.

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 against exp_k in $rdeps_{exp}$ never skips dependencies (Line 195) and therefore $rdeps(v) = rdeps_{exp}(v)$.

- Case 1.1: Assume that the unblock execution case would execute for $rdeps(v) \subseteq exp_k$. Then $rdeps(v) \setminus rdeps_{exp}(v) = \emptyset \subseteq committed$ which prevents the execution of the unblock execution case.

- Case 1.2: Thus, the unblock execution case can only execute 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 of $sc[0]$ must be executed and $sc[0] \subseteq rdeps_{exp}(sc[0])$. $sc[0] \subseteq exp_k$, that is, $sc[0]$ is a subset of the slots in the execution window. Thus, $rdeps_{exp}(sc[0]) \subset rdeps(sc[0])$ and therefore $\exists v_a \in rdeps(sc[0]) : (v_a \to v_e) \in rdeps(sc[0]) \wedge v_e \notin exp_k$. That is, $sc[0]$ depends on a slot v_e after the execution window. Due to the compact dependency encoding, this also results in a dependency on all earlier slots of the corresponding replica, including the root node v_r, which must be part of $sc[0]$ and was not executed yet. More formally, due to Corollary A.3.9, $exp(v_e.i) = v_r \in cdeps(v_e) \subseteq deps(v_a)$ and therefore $(v_a \to exp(v_e.i)) = (v_a \to v_r) \in rdeps_{exp}(sc[0]), v_r \in sc[0]$. By definition $v_e.seq - v_r.seq \geq k$ and therefore always $rdeps(sc[0]) \not\subset exp_k$. Thus, $sc[0]$ can never be executed via the standard case, which contradicts the assumption that $sc[0]$ is a regular SCC.

- The cases are exhaustive.

Case 2: For $v \in SSCC$, as $rdeps(v) \not\subset exp_k$ when executing v, it can never execute via the standard execution case.

The cases are exhaustive. □

As slots must be either executed via a regular SCC or an SSCC, either Lemma A.3.12 or Lemma A.3.14 applies to each slot.

Induction step 1 of Lemma A.3.11: Assume that the Lemma applies to $|t^i| = |t^j|$. We now show that is also applies to $|t'^i| = |t'^j| = |t^i| + 1$, where $t'^i[|t'^i| - 1]$ and $t'^j[|t'^j| - 1]$ are regular or empty SCCs.

The assumption $|t^i| = |t^j|$ can always be fulfilled by padding short traces with empty SCCs, which are skipped during execution.

Lemma A.3.16. *Lemma A.3.11 also applies to $|t'^i| = |t'^j| = |t^i|+1$, where $t'^i[|t'^i|-1]$ and $t'^j[|t'^j|-1]$ are regular SCCs. That is $\forall v \in flatten(t'^i) \cap flatten(t'^j) : rhist^i[v] = rhist^j[v]$.*

Proof. 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 to an s^j with swapped i and j.

Case 1: $s^i \in t'^j$: Both t'^i and t'^j contain $s = s^i$. Then according to Lemma A.3.15 SCC s must be executed as regular SCC at r_i and r_j. Thus, the proof follows from Lemma A.3.12.

Case 2: $s^i \notin t^j$: We show that in this case $s^i \cap flatten(t^j) = \emptyset$ such that the Lemma trivially holds. This is equivalent to $\forall s^j \in t'^j, s^i \neq s^j : s^i \cap s^j = \emptyset$. We prove this by contradiction. Assume that s_j is the SCC with the lowest index in t'^j with $s^i \cap s^j \neq \emptyset$. Then by Lemma A.3.15, both are regular SCCs, such that applying Lemma A.3.12 immediately results in a contradiction.

The cases are exhaustive.

□

Induction step 2 of Lemma A.3.11: Assume that the Lemma applies to $|t^i| = |t^j|$. We now show that is also applies to $|t'^i| = |t'^j| = |t^i| + 1$, where $t'^i[|t'^i| - 1]$ and $t'^j[|t'^j| - 1]$ are SSCCs or empty SCCs.

Lemma A.3.17. *Lemma A.3.11 also applies to $|t'^i| = |t'^j| = |t^i| + 1$, where $t'^i[|t'^i| - 1]$ and $t'^j[|t'^j| - 1]$ are SSCCs (or empty SCCs). That is $\forall v \in flatten(t'^i) \cap flatten(t'^j) : rhist^i[v] = rhist^j[v]$.*

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

Case 1: $\hat{s}^i \in t'^j$: Both t'^i and t'^j contain $\hat{s} = \hat{s}^i$. The proof immediately follows from Lemmas A.3.14 and A.3.15 as $rdeps^i_{exp}(\hat{s}) = rdeps^j_{exp}(\hat{s})$ and therefore $rhist^i[\hat{s}] = rhist^j[\hat{s}]$.

Case 2: $\hat{s}^i \notin t'^j$: We show that in this case $\hat{s}^i \cap flatten(t^j) = \emptyset$ such that the Lemma trivially holds. According to Lemma A.3.15 slots in an SSCC are executed via the unblock execution case on all replicas. As shown in the proof of Lemma A.3.14 an SSCC is fully determined by a single slot, such that $v_1 \in \hat{s}^i, v_1 \in \hat{s}^j \Rightarrow \hat{s}^i = \hat{s}^j$

The cases are exhaustive. □

This completes the proof of Lemma A.3.11 as according to Lemma A.3.15 slots are either executed via a regular SCC or an SSCC.

Theorem A.3.1 (repetition) (Execution Consistency)**.** *All replicas execute all pairs of committed, conflicting requests in the same order.*

Now we prove 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. Without loss of generality, 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 execution, which ensures a stable order.

Case 2: v_1 and v_2 are executed as part of different SCCs, $v_2 \in rhist[v_1]$: 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_1 and v_2 are executed as part of different SCCs, $v_2 \notin rhist[v_1]$: v_1 must be part of an SSCC, as only $rdeps_{exp}$ can exclude dependencies that are in $deps(v_1)$. When the SSCC was executed, this requires that $v_1 \in exp_k, v_2 \notin exp_k$. Then $v_r = exp(v_2.i) \in deps(v_1)$ due to Corollary A.3.9. In addition, $v_r \rightsquigarrow v_1$ as otherwise v_r would be executed before the SSCC. Therefore, $v_2.seq - v_r.seq \geq k$ such that v_2 cannot execute at any replica before v_1 due to the limited size of the execution windows.

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 s_B which depends on SCC s_A is tested first. If both SCCs are regular SCCs, then SCC s_A will be executed before s_B. As $rhist[v_a] := rdeps(s_A)$ for $v_a \in s_A$ it makes no difference whether $rdeps(s_B)$ is calculated before or after executing SCC s_A.

If only s_A is an SSCC, then s_A is executed first and afterwards the execution is restarted, which includes a recalculation of $rdeps(v_b)$. If only s_B is an SSCC, then we arrive at a contradiction, as the regular SCC s_A must already have been executed before the unblock execution case can apply. 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 that contain more than two SCCs. □

Remark A.3.18. It is sufficient for the unblock execution case to only check slots in $exp(*)$, that is, the root nodes. As shown in Lemma A.3.13, at least one slot in every SSCC is $\in exp(*)$.

Remark A.3.19. *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 partitioned 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 $\mathcal{A}$. This partitioning also applies to

the SCCs calculated for $rdeps(v)$ or $rdeps_{exp}(v)$. As already executed SCCs are always skipped, 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.3.20. 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 $\notin exp_k$ while traversing the graph.

A.3.3. Linearizability

Theorem A.3.21 (Linearizability)**.** *If two conflicting 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.*

Proof. This follows from Theorem A.2.2 and Theorem A.3.1. 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. □

A.3.4. Agreement Liveness

Similar to the liveness property in EPaxos [162], we show:

Theorem A.3.22 (Agreement Liveness)**.** *During synchronous phases a client request will eventually commit at all correct replicas.*

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.3.23. We say that `wait()` (Line 67) *accepts a slot as dependency*, if the function does not block permanently, that is, it returns eventually.

Lemma A.3.24. *If a correct replica r_i has accepted a* DepPropose *for slot s_j from replica r_j, then all other correct replicas will accept slot s_j as a dependency eventually.*

Proof. We show this by induction. For the base case assume that the DepPropose contains no dependencies. The propose timeout for slot s_j stays active at replica r_i until it has accepted $2f$ matching DepVerifys for the DepPropose (Line 47).

- **Case 1:** Coordinator r_j is correct.
  All replicas will receive the DepPropose and thus `wait()` accepts the slot s_j as dependency.

- **Case 2:** Coordinator r_j is faulty.
  r_j has created a valid DepPropose (otherwise it would not have been accepted by replica r_i) but does not distribute it correctly.
  - Case 2.1: Assume that replica r_i has accepted $2f$ matching DepVerifys. Only $f-1$ faulty DepVerifys are possible. Thus, at least $f+1$ of the $2f$ DepVerifys are from correct replicas. And therefore all replicas will receive $f+1$ DepVerifys causing `wait()` to accept the dependency.
  - Case 2.2: Alternatively, replica r_i will broadcast the DepPropose if it fails to collect $2f$ DepVerifys within the propose timeout. This allows all other replicas to learn about the slot corresponding to the DepPropose as the message was signed by the request coordinator and therefore `wait()` accepts the dependency.
  - The cases are exhaustive.
- **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 101).
- 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, the DepPropose for all these dependencies will, according to the induction assumption, be broadcasted if necessary such that the later DepPropose will be accepted eventually. □

Remark A.3.25. Note that the view-change special case to broadcast the DepPropose (Line 101) is not 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 DepVerifys in which case at least one of these was sent by a correct replica that has received the DepPropose and therefore also ensures its distribution. Together with the commit timeout 8Δ, which is much larger than the propose timeout of 2Δ, the special case would only trigger after another correct replica has already received and distributed the DepPropose.

Lemma A.3.26. *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.3.24 the corresponding messages will be accepted (by `wait()`). □

Lemma A.3.27. *`wait()` only accepts slots as dependencies once it is guaranteed that all correct replicas will eventually enforce the commit timeout for them.*

Proof. The `wait()` function waits for each dependency until one of the following cases holds (Line 69-72):

- **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.3.25.
- **Case 3:** DepPropose accepted.
  This enables a replica to broadcast the DepPropose itself if necessary.
- The cases are exhaustive.

Together with Lemma A.3.24 and A.3.26, the commit timeout is eventually active at all correct replicas. □

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

Lemma A.3.28. *A slot accepted by `wait()` will commit eventually at every correct replica during synchronous phases.*

Proof. Note that the Lemma only makes a statement about the slot but not which request will be committed.

Case 1: The request coordinator is correct and 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 (Line 26). While the network is in a synchronous phase, then the `wait()` calls in lines 26 and 43 do not block permanently according to Lemma A.3.26. 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 at least $f + 1$ 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 the slot 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 (Line 157). These up to f replicas then receive the result via Execute messages from the 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 (Line 119-121). 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 be in view $v+1$ at the same time if the network is synchronous, see also Lemma A.3.32. The correct replicas will start their view change timeout, as enough ViewChanges exist to ensure that a NewView can be created eventually.

  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 initiate a view change, will receive $2f+1$ ViewChanges and start their view-change timeouts. Then each replica either 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.3.26 we immediately arrive at a contradiction. □

- The cases are exhaustive.

Case 2: The request coordinator is correct and the 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 faulty 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 a no-op request 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.3.27, correct replicas that have received the DepPropose will time out while waiting that the dependencies are distributed to a quorum of replicas. This will trigger a view change that will fill the slot with a no-op request. Thus, non-existing dependencies for a slot cannot commit and consequently do not affect the request execution. □

- The cases are exhaustive.

Case 3: The request coordinator is faulty.
After the DepPropose has been accepted by `wait()`, the commit timeout for the slot will eventually be active at all replicas. Thus, either the slot commits as in Case 1.1, or causes a view change according to Case 1.2 or Case 2. In both cases the slot will commit eventually.
The cases are exhaustive. □

Lemma A.3.29. *The fast-path quorum F will eventually contain only correct replicas.*

Proof. After filling a slot with a no-op request during the view change, the fast-path quorum is rotated (Line 151). This will eventually result in a fast-path quorum F which only contains correct replicas. □

Remark A.3.30. Note that a faulty replica cannot prevent slots of correct replicas from committing by proposing manipulated DepProposes. Assume this were the case. Then a correct replica has to accept a DepPropose from the faulty coordinator. Therefore, Lemma A.3.24 applies, which yields a contradiction. Thus, a faulty coordinator can only cause the processing of its own DepPropose to block in `wait()`, which will also prevent all further slots of that faulty replica to block in `wait()` (Line 26) until the faulty DepPropose is finally accepted.

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

Lemma A.3.31. *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 timeout is 2Δ. Thus, after 2Δ replica r_i has either received $2f$ DepVerifys and therefore 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. The DepPropose will be accepted within 3Δ, as the argument also applies to all its dependencies, which were proposed before and thus must already be accepted earlier on. □

Lemma A.3.32. *The calculation of a* NewView *can complete within at most* 3Δ *during synchronous phases.*

Proof. Once a correct replica has received $2f + 1$ ViewChanges then within 2Δ every correct replica will receive $2f+1$ ViewChanges. 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. We will revisit this timeout after adding support for checkpointing. □

Lemma A.3.33. *A commit timeout of at least* 8Δ *allows correct coordinators to commit during 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. □

Now, we show Theorem A.3.22.

Theorem A.3.22 (repetition) (Agreement Liveness)**.** *During synchronous phases a client request will eventually commit at all correct replicas.*

Proof. For slots in which the request was replaced by a no-op request, the request coordinator will propose the request again (Line 152). Together with Lemmas A.3.26, A.3.28 and A.3.29 this ensures that a slot / slots and also eventually the request will commit at all correct replicas. The client also broadcasts its request to all replicas after a timeout. This guarantees that a correct coordinator will receive the request and commit it. □

Lemma A.3.34. *The compact dependency encoding (cf. Corollary A.3.9) 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, $s_d < max_{s_j} = \max_{r_j}\{d \in D_i \mid d.i = r_j\}$. 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 (Line 26). Thus, the guarantees provided by `wait()` also include the earlier additional sequence numbers s_d.

The compact dependency encoding does not affect execution consistency, as it can only add but not remove dependencies. □

A.3.5. Execution Liveness

Theorem A.3.35 (Execution Liveness)**.** *During synchronous phases a client will eventually receive a result.*

Lemma A.3.36. *Any slot included as dependency of a committed slot will commit eventually.*

Proof. The `wait()` calls in Lines 26 and 43 together with Lemmas A.3.27 and A.3.28 ensure that all dependencies of any committed slot will commit eventually. □

Lemma A.3.37. *A committed request will be executed eventually.*

Proof. Lemma A.3.36 shows that all slots on which a committed slot depends will commit eventually. In order to avoid the execution live lock problem discussed in EPaxos [162], we now show that there is a finite upper bound for the number of slots that have to commit before a slot can be executed.

A slot s can be executed via the standard case (Line 208) 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 (Line 213) executes slots in $rdeps_{exp}(s)$ which by construction always is $\subseteq exp_k$. Thus, it remains to be shown that exp_k can only contain a bounded number of slots that can block the execution.

A slot s can only depend on a bounded number of slots (as all dependencies must have been proposed using a DepPropose before). Thus, if any dependency v_d among these dependencies is not yet executed and therefore can prevent execution of s, then it serves as a finite upper bound for $exp(*)$ such that $exp(v_d.i).seq \leq v_d.seq$. Other dependee slots can further restrict $exp(*)$, which limits 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.3.22, Lemma A.3.37 and Line 218 which guarantee that a client receives at least $f + 1$ matching replies from correct replicas.

A.4. Checkpointing

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.

223 **Variables at each replica**:
224 Δ_{vc} := 5Δ

Fast Path

225 Propose checkpoint request CheckpointReq if $s_j.seq \mod k = 0$

227 **Follower** f_i **receives** $dp := \langle\langle \text{DepPropose}, s_j, h(r), D, F\rangle, r\rangle$ **from** co:
228 **pre:** $step[s_j] = \texttt{init}$

```
    // Each replica must propose a checkpoint request exactly every k slots
    assert (s_j.seq mod k = 0) ⊕ (r = CheckpointReq)
    [...]

conflicts(Request r):
    // A checkpoint request CheckpointReq conflicts with all other requests
    return {s_i | ∀s_i, pr[s_i] ≠ ∅ : conflict(pr[s_i], r)} ∪ barrier of latest stable checkpoint
```

View Change

```
Move to new view v_{s_j} for slot s_j at replica r_i:
    [...]
    else if step[s_j] ∈ {rp-prepared, rp-committed}:
        [...]
    else:                                              // Fallback to default request
        if s_j.seq mod k = 0:
            msg := CheckpointReq
            D_{r_i} := D_{r_i} used by r_i for own DepPropose / DepVerify
                       or as fallback conflicts(msg) \ s_j
        else:
            msg := NO-OP
            D_{r_i} := conflicts(msg) \ s_j
        dv := ⟨DepVerify, s_j, r_i, h(msg), D_{r_i}⟩_{σ_{r_i}}
        cert[s_j] := ⟨DRC-part, msg, dv, −1⟩            // For view -1
    view[s_j] := v_{s_j}
    [...]

View-change coordinator co for view v_{s_j} receives valid
        VCS := {⟨ViewChange, s_j, v_{s_j}, ∗⟩} from 2f + 1 replicas:
    pre: for each VC ∈ VCS containing a DRC-part: block until wait(VC.dv.D_{f_i})
         has returned
    assert VC.dv.h(msg) equals h(CheckpointReq) if VC.dv.s_j.seq mod k = 0
           else h(NO-OP)
    [...]
    select dp, d⃗v from [...]
    if dp = NO-OP:
        if s_j.seq mod k = 0:
            dp := CheckpointReq
        d⃗v := {VC.dv | VC ∈ VCS}                       // Each VC must contain a DepVerify
    Broadcast ⟨NewView, s_j, v_{s_j}, dp, d⃗v, VCS⟩_{σ_co}
```

Request Execution

```
execute(SCC v⃗, DependencyGraph G):
    barrier := ∅
    if CheckpointReq ∈ v⃗:
```

```
265    barrier := ({x | ∀r_i : x < exp(r_i)} ⋃_{v∈v⃗, v.req=CheckpointReq} deps(v) ∪ v) ∩ exp_k
266  for c ∈ sort(v⃗):
267    if c ≠ CheckpointReq ∧ (barrier = ∅ ∨ c ∈ barrier):
268      Execute request c and reply to client
269      rhist[v] := G
270  if barrier ≠ ∅:
271    Create execution checkpoint with barrier
272    Restart request execution
```

As described in Section 4.5, a replica broadcasts 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 retrieves the checkpoint certificate 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.

To show that the Consistency property also holds for checkpoint requests, the following Lemma adapts the proof of Theorem A.2.2 accordingly.

Lemma A.4.1. *For a slot, if a DRC is selected during a view change then the slot did not commit previously.*

Proof. As shown in the proof of Theorem A.2.2, the NEWVIEW calculation always includes an FPC or RPC if the slot committed. Thus, the DRC cannot be selected. □

Lemma A.4.2. *All correct replicas create identical checkpoints when executing the same 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 265).

We now show that the checkpoint barrier is tight.

Case 1: 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.3.9 the checkpoint request depends on x which therefore must be executed first.

Case 2: 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.

- Case 2.1: 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 on 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 checkpoint barrier covers $\{x' \mid x' < exp(*)\}$.
- Case 2.2: 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.
- The cases are exhaustive.

The cases are exhaustive.
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 becomes stable, all later requests will include dependencies on all slots included in the checkpoint, that is, they will depend on everything covered by the checkpoint barrier. Compared to an execution without checkpointing this can only introduce additional dependencies. However, as all slots covered by the checkpoint barrier are already executed, these have no influence on the request execution. □

The following Lemma adapts the proof of Theorem A.3.22 to also consider checkpoint requests.

Lemma A.4.3. *If no RPC or FPC is included in the view change for a slot, that is, the slot did not commit, then a DRC is selected during a view change.*

Proof. The NewView calculation requires $2f + 1$ ViewChanges, which are sufficient to generate a DRC (Line 257-260). As shown in the proof of Lemma A.3.28, 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 253). Once a DRC has committed via the reconciliation path, then it is handled like any other request. This ensures that no-op requests or CheckpointReqs collect dependencies reported by a quorum of replicas such that Lemma A.3.2 also holds for these requests. □

We modify Lemma A.3.32 as follows:

Lemma A.4.4. *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 from correct replicas are sent after Δ and are accepted at most 3Δ later, similar to Lemma A.3.31. 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. □

B

Safety and Liveness Proof for Cloud-Based Hierarchical Replication

This chapter is a partially revised version of the pseudocode and proofs presented in the appendix of the paper [86] of which I am the main author. In Appendix B.1 we first provide a detailed description of the individual components of SPIDER, along with the assumptions and definitions used for proving the correctness and liveness properties of SPIDER. Afterwards we present the pseudocode for SPIDER in Appendix B.2 and the proof in Appendix B.3. We conclude with pseudocode for both IRMC implementation variants (IRMC-RC and IRMC-SC) in Appendix B.4.

B.1. Properties

We first describe the properties provided by SPIDER before stating the required properties of the agreement protocol black box, the checkpoint transfer component, the application and the IRMCs. These components are identical to the building blocks described in Section 5.3.

We assume that each execution group consists of $2f_e + 1$ replicas and that there are at most f_e faulty execution replicas per execution group. The agreement group has $3f_a + 1$ replicas of which at most f_a agreement replicas may be faulty. All faults are assumed to be Byzantine. We assume a partially synchronous network with periods of synchrony that are long enough to allow the protocol to make progress [80].

B.1.1. Properties of Spider

The definitions of E-Safety and E-Validity follow the lines of those used for Steward [24]. E-Safety II and E-Liveness are adapted from PBFT [57]. E-Validity II captures the usual at-most-once guarantee.

Theorem B.1.1 (E-Safety)**.** *If two correct replicas execute the i^{th} write, then these writes are identical.*

Theorem B.1.2 (E-Safety II)**.** *The system provides linearizability regarding requests from correct clients.*

Theorem B.1.3 (E-Validity)**.** *Only a correctly authenticated write request from a client may be executed.*

Theorem B.1.4 (E-Validity II)**.** *A correct replica executes a write request at most once.*

Theorem B.1.5 (E-Liveness)**.** *A correct client will eventually receive a reply to its request.*

Consistency Guarantees

SPIDER provides strong consistency (linearizability) for write requests. Read requests with strong consistency are treated similarly, but only the designated execution group gets the full request, whereas all other groups just receive the client identifier c and counter value t_c. Weakly consistent read requests provide prefix consistency.

B.1.2. Cryptographic Primitives and Assumptions

The pseudocode uses the following cryptographic primitives:

- $sign(m)$: Digitally signs message m (e.g., using RSA or ed25519).
- $valid\_sig_{\mathcal{E}}(m)$: Verifies that the signature for message m is valid and that the signer is part of group $\mathcal{E}$.
- $mac_{r_a,r_e}(m)$: Adds a single message authentication code (MAC) to authenticate message m from replica r_a towards replica r_e [199]. This primitive, for example, may be implemented using HMAC-SHA256 [133, 164].
- $mac_{r_a,\mathcal{E}}(m)$: Adds a MAC authenticator such that replica r_a authenticates message m to a replica group $\mathcal{E}$ [57]. It consists of a MAC for each replica in group $\mathcal{E}$.
- $valid\_mac_{r_a,e}(m)$ and $valid\_mac_{r_a,\mathcal{E}}(m)$ are used to verify these MACs.
- $unwrap\_mac(m)$: Strips the added MAC from message m and returns the original message without the authentication.
- $h(m)$: Calculate a cryptographically secure hash digest of message m, for example, using SHA256.

We make the standard assumptions regarding cryptographic functions. We assume them to be secure, that is, a malicious replica cannot forge signatures or MACs of other replicas nor can it create a message $m' \neq m$ with hash $h(m') = h(m)$. That is, the hash function is collision resistant.

```
1  interface Agreement {
2      // Blocks until the request is returned by the ordered callback on this replica
3      VOID order_request(CLIENTID c, CLIENTCTR t_c, REQUEST r)
4      // Deliver ordered requests one after another
5      // Blocking callback, that is, the agreement can only deliver the next message after
           the previous call has completed
6      // Delays in the callback may cause timeouts in the agreement protocol black box to
           expire
7      callback ordered(SEQNR s, REQUEST r)
8      // After this call no sequence number < s must be delivered
9      VOID collect_garbage_before(SEQNR s, CLIENTCTR[] ts)
10 }
```

Figure B.1: Interface of the agreement protocol black box

B.1.3. Agreement Protocol Black Box

We assume the agreement component to be a black box with the interface shown in Figure B.1 and the following properties. The comments at the interface methods detail their expected behavior. We assume that the first delivered sequence number is 1.

Definition B.1.6 (A-Safety)**.** If two correct agreement replicas deliver an ordered message for sequence number s, then these messages are identical.

Definition B.1.7 (A-Liveness)**.** Once $2f + 1$ correct replicas receive a message m for ordering, then eventually $f + 1$ correct replicas will deliver message m and all preceding messages.

Definition B.1.8 (A-Validity)**.** A correct agreement replica will only deliver correctly authenticated client requests.

Definition B.1.9 (A-Order)**.** A correct agreement replica will deliver a message for sequence number s only after all preceding sequence numbers were delivered or garbage collected.

These requirements are, for example, fulfilled by PBFT [57].

B.1.4. Checkpoint Transfer Component

We assume that each replica has a checkpoint component with the interface from Figure B.2 and the following properties. The comments at the interface methods detail their expected behavior.

Definition B.1.10 (Stable checkpoint)**.** A checkpoint is called *stable* once a correct replica collects a certificate consisting of $f + 1$ valid and matching checkpoint messages.

```
1  interface Checkpoint {
2      // Create and distribute checkpoint
3      // By default only checkpoint transfer components within a single group communicate
           with each other (i.e., checkpoints are group specific)
4      VOID generate(SEQNR s, STATE st)
5      // Sequence numbers for returned checkpoints must increase
6      // Checkpoints may be skipped
7      callback stable(SEQNR s, STATE st)
8      // Explicitly request the retrieval of a checkpoint, possibly from another group
           (execution groups only)
9      VOID fetch(SEQNR s)
10 }
```

Figure B.2: Interface of the checkpoint transfer component

Once a replica possesses a stable checkpoint, it will pass the checkpoint to the `stable()` callback, unless it has already delivered a checkpoint with a higher sequence number.

Definition B.1.11 (CP-Safety)**.** A stable checkpoint was created by at least one correct replica.

As shown later on, all correct replicas in a group will create identical checkpoints for the same sequence number.

Definition B.1.12 (CP-Liveness)**.** If one correct replica of a group delivers a checkpoint, then eventually all correct replicas of that group will deliver that checkpoint, unless a newer checkpoint was already delivered.

Definition B.1.13 (CP-Liveness II)**.** Once $f + 1$ correct replicas create and distribute identical checkpoint messages, the checkpoint will eventually become stable, unless it is superseded by a newer one before.

An implementation should consider the following aspects:

- With an execution group size of $2f_e + 1$, CP-Safety requires that each checkpoint message is authenticated using a signature. Section 2.3.3.5 describes a possible implementation.
- In order to provide CP-Liveness, correct replicas within a group must continuously inform or query each other about their latest stable checkpoint.
- Replica should only exchange checkpoint messages containing a hash $h(st)$ of the checkpoint state st to keep the network overhead low.
- The full checkpoint state should only be transferred when necessary.

```
1  interface Application {
2      // Execute request and return a result
3      RESULT execute(REQUEST m)
4      // Verify that a request is readonly
5      BOOLEAN is_read_only(REQUEST m)
6      // Create application snapshot
7      APPSTATE snapshot()
8      // Apply application snapshot
9      VOID apply(APPSTATE st)
10 }
```

Figure B.3: Interface of the application component

B.1.5. Application

We assume that the application provides the interface shown in Figure B.3 and is implemented as a deterministic state machine, which can `execute()` client requests and provide a reply to them. In addition, the application must be able to serialize the application state using `snapshot()` and `apply()` it.

Definition B.1.14 (Replicated state machine (RSM))**.** Different application instances have an identical state for sequence number i when processing writes according to the same total order [179].

B.1.6. IRMC Properties

The sender and receiver endpoint interfaces of the IRMC are shown in Figure B.4. As before, the comments specify the expected behavior of the methods. All sender replicas are contained in the set R_s and all receiver replicas in R_r. The capacity of an IRMC (subchannel) is denoted as $|IRMC|$ and is assumed to be ≥ 1. It is identical for all subchannels of an IRMC. $IRMC_{sc}.win$ refers to the window of subchannel sc, which is initialized to start at 1. $min(IRMC_{sc}.win)$ and $max(IRMC_{sc}.win)$ return the lower and upper limit (inclusive) of the window of subchannel sc, respectively. $receive(sc, p) = m$ denotes that the receive call returned the message m.

Definition B.1.15 (IRMC-Correctness I)**.** Receive only returns a message sent by a correct sender:
$(receive(sc, p) = m) \Rightarrow$ a correct sender called $send(sc, p, m)$
$\wedge$ the receiver called $move\_window(sc, p')$ such that $p' \leq p < p' + |IRMC_{sc}|$.

Definition B.1.16 (IRMC-Correctness II)**.** Moving a window requires a move request by at least one correct replica:
($receive(sc, p)$ returns a $\langle \text{TOOOLD}, p' \rangle$ message with $p' > p$) $\Rightarrow$ a correct sender endpoint called $move\_window(sc, \hat{p})$ with $\hat{p} \geq p' \vee$ a correct receiver called $move\_window(sc, \hat{p})$ with $\hat{p} \geq p'$.

```
1  // Sender endpoint
2  interface IRMC_Sender {
3      // If p is too old: discard m and return immediately
4      // If p is in the current window: send m and return immediately
5      // If p is after the current window (p > max(IRMC_sc.win)): block/wait
6      VOID send(SUBCHANNEL sc, POSITION p, MESSAGE m)
7      // Ask receiver endpoint to move the window forward
8      // The receiver endpoint will internally call move_window with the f_s + 1-highest
           received position
9      VOID move_window(SUBCHANNEL sc, POSITION p)
10 }

12 // Receiver endpoint
13 interface IRMC_Receiver {
14     // Blocks until either
15     // (1) a message m is delivered, then returns m,
16     // (2) or until the window is ahead of p, that is, p < min(IRMC_sc.win), then
           returns ⟨TOOOLD, s⟩, with s = new window lower bound
17     MESSAGE receive(SUBCHANNEL sc, POSITION p)
18     // Position p must increase monotonically, calls with lower values are silently
           ignored
19     VOID move_window(SUBCHANNEL sc, POSITION p)
20 }
```

Figure B.4: IRMC interfaces (pseudocode)

Remark B.1.17. Calls to the `send` method block if the requested position is after the upper limit of the current subchannel window. Calls to the `receive` method block if the position is in or after the subchannel window and the corresponding message was not yet received by the IRMC.

Definition B.1.18 (IRMC-Liveness I)**.** An identical message sent (`send` method call has returned) by at least $f_s + 1$ correct replicas will eventually cause some message to be received by all correct receivers unless it is skipped (see also IRMC-Correctness II):
If $f_s + 1$ correct senders call $send(sc, p, m)$, then eventually $\forall$ correct $r \in R_s$ that call(ed) $receive(sc, p)$: $receive(sc, p) = * \vee receive(sc, p) = \langle \text{TOOOLD}, p' \rangle$ with $p' > p$.

Remark B.1.19. Due to IRMC-Correctness I, the received message can only be one that was sent by at least one correct sender.

Definition B.1.20 (IRMC-Liveness II)**.** Send calls return once the position is below the subchannel window's upper bound:
If $f_r + 1$ correct receivers $r \in R_r$ call $move\_window(sc, p_r)$, where p_r is a receiver-specific position, then eventually all $send(sc, p', m)$ calls will have returned on all correct sender replicas where $p' < \tilde{p} + |IRMC_{sc}|$ and $\tilde{p} = f + 1$-largest p_r.

```
1  t_c := 1                                                    // Client request counter
2  rep := ∅                                                    // Reply for last request
3  g := {}                                                     // Collected replies
4  E := nearest execution group with |E| = 2f_e + 1
5  write(WRITE w):
6      // Authenticate request
7      m := mac_{c,E}(sign_c(⟨WRITE, w, c, t_c⟩))
8      rep := ∅
9      g := {}
10     // Repeat sending until reply was received
11     while rep = ∅:
12         broadcast m to E
13         sleep for t_retry ∨ until rep ≠ ∅
14     t_c := t_c + 1
15     return rep

17 on receive(m = ⟨REPLY, u, t'_c⟩ from e ∈ E):
18     // Only process correctly authenticated replies
19     // Only accept first reply from each replica
20     if valid_mac_{e,c}(m) ∧ t'_c = t_c ∧ (⟨REPLY, *, *⟩ from e) ∉ g:
21         g := g ∪ {m}
22         // Return reply after receiving f_e + 1 replies with matching t_c and u
23         if ∃u : |{v|v = ⟨REPLY, u, t_c⟩ ∈ g}| ≥ f_e + 1:
24             rep := u
```

Figure B.5: Client c (pseudocode)

Definition B.1.21 (IRMC-Liveness III)**.** Receiver endpoints will move the window at least as far as the $f_s + 1$-highest `move_window()` call by a sender replica:
If $f_s + 1$ correct senders call $move\_window(sc, p_s)$, then eventually all correct receiver endpoints will have (internally) called $move\_window(sc, p)$ with p such that $p \in [f + 1$-largest p_s, largest $p_s]$.

Remark B.1.22. Note that if a receiver endpoint has already moved a subchannel window to a higher position than p, then the call to `move_window()` has no effect.

B.2. Spider Pseudocode

The pseudocode for the client is shown in Figure B.5, for the execution replica in Figure B.6 and for the agreement replica in Figure B.7. Line numbers in the following refer to one of these figures. The presented pseudocode covers the write request processing as described in Section 5.4.2. The pseudocode for the agreement and execution replicas has already been presented in Section 5.4.2.

```
25 s_n := 0                                          // Sequence number for last executed request
26 t[c] := 0                                         // Counter of latest forwarded client request
27 u[c] := ∅                                         // Reply cache ⟨Reply, u_c, t_c⟩
28 app = application, cp = checkpoint transfer component
29 E := execution group with |E| = 2f_e + 1
30 r_E = request IRMC sender                         // Each subchannel has a capacity of 2
31 // Subchannel 0 is used as commit channel, any other subchannel could also be used
32 c_E = commit IRMC receiver                        // Commit subchannel capacity must be ≥ k_e
33 on receive(m = ⟨Write, w, c, t_c⟩ from c):
34     if !valid_mac_{c,E}(m): return                // Ignore invalid requests
35     if t_c ≤ t[c]:
36         if u[c] = ⟨Reply, *, t'_c⟩ ∧ t'_c = t_c: // Check if a reply is available for the request
37             send mac_{r_e,c}(u[c]) to c
38         return                                    // Silently return on retry with no result yet
39     if !valid_sig_c(unwrap_mac(m)): return
40     // Each execution replica must forward a request once, even already executed ones
41     t[c] := t_c
42     r_E.move_window(c, t_c)                       // Notify agreement of new request
43     r_E.send(c, t_c, ⟨Request, unwrap_mac(m), E⟩)

45 main loop:
46     while true:
47         m := c_E.receive(0, s_n + 1)
48         if m = ⟨TooOld, s'⟩:
49             // Executor missed committed requests → fetch checkpoint
50             cp.fetch(s')                          // Ask other groups if necessary
51         else: // m = ⟨Execute, ⟨Request, ⟨Write, w, c, t_c⟩, E'⟩, s_n + 1⟩
52             s_n := s_n + 1
53             // Only execute new requests
54             if (u[c] = ⟨Reply, *, t'_c⟩ ∧ t_c > t'_c) ∨ u[c] = ∅:
55                 u_c := app.execute(m)
56                 u[c] := ⟨Reply, u_c, t_c⟩          // Store reply
57                 if E = E': // Only the local execution group sends the reply to the client
58                     send mac_{r_e,c}(u[c]) to c
59             if s_n ≡ 0 mod k_e:                   // Periodically create a checkpoint
60                 cp.generate(s_n, (u, app.snapshot()))

62 on cp.stable(s, st = (u', app')):
63     c_E.move_window(0, s + 1)                     // Allow garbage collection of commit channel
64     if s ≥ s_n:
65         s_n := s; app.apply(app'); u := u'
```

Figure B.6: Execution replica r_e (pseudocode)

```
66 s_n := 0                                                    // Last ordered sequence number
67 win := [1,AG-WIN]                                  // Range with [lower, upper] bound, both inclusive
68 AG-WIN ≥ k_a                                                // Size of agreement window
69 t[c] := 0                                      // Counter values of latest ordered request per client
70 t⁺[c] := 0                                                 // Counter values for next expected request
71 n_e := number of execution groups; z := limit on slow execution groups
72 hist := last |c_{E,0}| EXECUTEs
73 ag = agreement protocol black box, cp = checkpoint transfer component
74 A := agreement group with |A| = 3f_a + 1
75 for each execution group E:
76     r_E = request IRMC receiver                          // Each subchannel has a capacity of 2
77     c_E = commit IRMC sender                    // Commit subchannel capacity must be ≥ k_e
78 parallel for each client c and execution group E:
79     while true:
80         m := r_E.receive(c, t⁺[c])
81         if m = ⟨TOOOLD, t_c⟩: t⁺[c] := t_c
82         else: // m = ⟨REQUEST, ⟨WRITE, w, c, t_c⟩, E⟩
83             ag.order_request(c, t_c, m)                  // Returns once request is ordered
84             t⁺[c] := t_c + 1

86 // Delivered in-order, agreement must timeout if blocked for too long
87 on ag.ordered(s, r = ⟨REQUEST, ⟨WRITE, w, c, t_c⟩, E⟩):
88     sleep until s ≤ max(win)          // Force agreement to periodically create a checkpoint
89     // Update state with new request
90     t[c] := t_c; t⁺[c] := max(t_c + 1, t⁺[c]); hist.add(⟨EXECUTE, r, s⟩)
91     s_n := s
92     parallel for each execution group E:
93         c_E.send(0, s, ⟨EXECUTE, r, s⟩)
94     sleep until completed for n_e − z groups     // Send calls continue in the background
95     if s_n ≡ 0 mod k_a: cp.generate(s_n, (t, hist))   // Create checkpoint periodically

97 on cp.stable(s, st = (t', hist')):
98     parallel for each execution group E:
99         c_E.move_window(0, s − |hist'| + 1)              // Move commit window forward
100    ag.collect_garbage_before(s + 1, t')
101    if s > s_n:
102        tmp := s_n; s_n := s; t := t'; hist := hist'
103        parallel for each execution group E:
104            for x = ⟨EXECUTE, r, s'⟩ ∈ hist, s' ∈ [tmp + 1, s]:
105                c_E.send(0, s', x)     // Add missing requests from hist to commit channel
106        sleep until completed for n_e − z groups
107    win := [s+1, s+AG-WIN]
```

Figure B.7: Agreement replica r_a (pseudocode)

We assume that each method is executed atomically, unless it calls a blocking method, at which point the execution may switch to other methods. Variable definitions are written as `var := value`, whereas `=` is used for comparisons and destructuring of values, for example, $x = \langle \text{EXECUTE}, r, s' \rangle$ uses the value in x to define r and s' using pattern matching.

B.3. Proof

The proof primarily considers write requests. We assume for now that there is only one execution group, that is, $n_e = 1$ and $z = 0$. Later on, we will relax this assumption. Strongly and weakly consistent read requests are considered afterwards.

B.3.1. Agreement Checkpoint Equivalence

We begin the proof with an auxiliary lemma to simplify the handling of checkpoints.

Lemma B.3.1 (CP-A-Equivalence)**.** *The state of an agreement replica (s_n, t, $hist$ and queued commit IRMCs messages) that has reached sequence number s via processing `ag.ordered`(s, r) (Line 87) is equivalent to that of a replica that reaches sequence number s by applying a checkpoint for sequence number s.*

Proof. We prove this by induction.

Base case: All correct agreement replicas initialize s_n, t, $hist$ and the commit IRMCs with identical values. There is no checkpoint for that sequence number, as no checkpoint was generated yet.

Induction step: All correct agreement replicas pass through the same states by processing ordered requests or jump forward to one of those states via a checkpoint.

As the considered state parts are only updated in either `ag.ordered` (Line 87) or `cp.stable` (Line 97), it suffices to show that when either of them updates s_n to a certain sequence number, then the resulting replica states are equivalent. Note that the sequence number s_n increases monotonically as `ag.ordered` is per A-Order only called for increasing sequence numbers and `cp.stable` only increases the value of s_n (Line 101).

Assume that from a common starting point, replicas reach sequence number s by processing `ag.ordered`(s, r) (Line 87): Per A-Safety and A-Order all correct agreement replicas receive the same sequence of requests via their `ag.ordered` callback, that is, s_n, t and $hist$ (Line 90) evolve identically on those replicas. Therefore, a possible later call to `cp.generate`$(s, (t, hist))$ (Line 95) for a sequence number s has identical parameters on all correct agreement replicas.

As per CP-Safety only checkpoints which were created by at least one correct replica can become stable, any call of `cp.stable`$(s, (t', hist'))$ (Line 97) can only deliver that checkpoint for sequence number s. Applying a checkpoint for the current or an older sequence number $s \leq s_n$ does not change s_n, t and $hist$ (Line 101). Applying a checkpoint for a newer sequence number $s > s_n$ atomically updates s_n, t and $hist$ to the state they had when the checkpoint was created (Line 102) and adds missing requests (i.e., those skipped

by updating s_n) to the commit IRMCs. The call to `ag.collect_garbage_before`$(s+1)$ (Line 100), which happens atomically with the state update, ensures that `ag.ordered` will only be called for sequence numbers $\geq s+1$. Per A-Order the next `ag.ordered` call must be for $s_n + 1 = s + 1$.

When called for an old checkpoint ($s \leq s_n$), then $c_{\mathcal{E}}$`.move_window` (Line 99) has no effect, as a $c_{\mathcal{E}}$`.send` call for s_n must already have been issued, such that the IRMC has queued messages at least up to sequence number s. Therefore, $max(c_{\mathcal{E},0}.win) \geq s \Leftrightarrow min(c_{\mathcal{E},0}.win) \geq s - |c_{\mathcal{E},0}| + 1$, that is, the window start is already at least at the position requested by the $c_{\mathcal{E}}$`.move_window` call, see also the remark below.

For a newer checkpoint, as $|hist'| = |c_{\mathcal{E},0}|$, this together with moving the window forward from the sender side (per IRMC-Liveness II and IRMC-Liveness III) is enough to fully update the state of the commit channel if necessary. Requests that were already contained in the IRMC must be identical as the message sent for a specific sequence number s in `ag.ordered` or `cp.stable` (Line 93 and 105) must be identical per induction assumption. □

Remark B.3.2. $c_{\mathcal{E}}$`.move_window` (Line 99) is actually called with $s - |hist'| + 1$ which has the same effect as $s - |c_{\mathcal{E},0}| + 1$ such that we assume $|hist'| = |c_{\mathcal{E},0}|$ in the following to simplify the presentation of the proof. As the first delivered agreement sequence number is 1 and for every delivered request a new message is added to *hist* (Line 90), the size of $|hist| = min(s_n, |c_{\mathcal{E},0}|)$. Thus, when applying a checkpoint $s - |hist'| + 1 = s - min(s, |c_{\mathcal{E},0}|) + 1 = max(1, s - |c_{\mathcal{E},0}| + 1)$. As the lower bound of the subchannel window $min(c_{\mathcal{E},0}.win)$ is initialized to 1 and $c_{\mathcal{E}}$`.move_window` ignores calls which would move the window backwards, $s - |c_{\mathcal{E},0}| + 1$ is equivalent to $s - |hist'| + 1$.

B.3.2. Execution Safety

Theorem B.1.1 (repetition) (E-Safety)**.** If two correct replicas execute the i$^{\text{th}}$ write, then these writes are identical.

To prove theorem E-Safety we start with the following lemma:

Lemma B.3.3. *When two execution replicas e_1 and e_2 receive message m and m' at position p in the commit channel, then $m = m'$.*

Proof. We prove this by contradiction. Assume that $m \neq m'$. Per IRMC-Correctness I $c_{\mathcal{E}}$`.receive`$(0, p)$ (Line 47) only delivers a message m that was sent by a correct agreement replica, the same holds for m'. Therefore, a correct agreement replica must have called $c_{\mathcal{E}}$`.send`$(0, p, m)$ and another correct agreement replica $c_{\mathcal{E}}$`.send`$(0, p, m')$ (either at Line 93 or 105). For this to happen via the $c_{\mathcal{E}}$`.send` call in `ag.ordered`, the agreement protocol black box must have delivered message m and m' on two correct replicas, which contradicts A-Safety. And according to CP-A-Equivalence the $c_{\mathcal{E}}$`.send` when applying a checkpoint in `cp.stable` is equivalent to the previous send call in `ag.ordered`, which contradicts the assumption. □

With this we can prove E-Safety:

Corollary B.3.4. *An execution replica only executes requests received from the commit channel (cf. Lines 47 and 55) which according to Lemma B.3.3 cannot receive different requests on different correct execution replicas.*

B.3.3. Execution Checkpoint Equivalence

Lemma B.3.5 (CP-E-Equivalence)**.** *The state of an execution replica (s_n, app and u) that has reached sequence number s_n via processing the corresponding* Execute *message (Line 51) for s_n is equivalent to that of a replica that arrives there via a checkpoint for sequence number s_n.*

The proof follows along the lines of CP-A-Equivalence.

Proof. We prove this by induction.

Base case: All correct execution replicas initialize s_n, *app* and u with identical values. There is no checkpoint for that sequence number, as no checkpoint was generated yet.

Induction step: All correct execution replicas pass through the same states or jump forward to one of those states via a checkpoint.

As the considered state parts are only updated in either the main loop (Line 45) or `cp.stable` (Line 62), it suffices to show that when either of them updates s_n to a certain sequence number, then the resulting replica states are equivalent. Note that the sequence number s_n increases monotonically as the main loop only increments it (Line 52) and `cp.stable` can only increase the value of s_n (Line 64).

Assume that from a common starting point, execution replicas reach sequence number s_n by processing the corresponding Execute message (Line 51): As $c_{\mathcal{E}}$`.receive`$(0, s_n + 1)$ (Line 47) is called sequentially (without skipping) for each sequence number and per E-Safety all correct execution replicas process the same requests for each sequence number, the (atomic) modifications of s_n, $u[c]$ and *app* in the main loop (Line 52 and following) are identical across execution replicas. Either all correct execution replicas skip the execution of request r (Line 54) based on $u[c]$, which must be identical across replicas as per induction assumption the replica states were identical which includes $u[c]$, or according to the RSM the execution replicas arrive at identical $u[c]$ and `app` for s_n after processing r.

Therefore, a call to `cp.generate`(s, (u, `app`)) (Line 60) for sequence number s has identical parameters on all correct execution replicas and thus per CP-Safety `cp.stable`(s, (u', `app'`)) (Line 62) can only deliver that checkpoint.

Applying a checkpoint for the current or an older sequence number $s \leq s_n$ does not change s_n, *app* and u (Line 64). Applying a checkpoint for a newer sequence number $s > s_n$ atomically updates s_n, *app* and u to the state they had when the checkpoint was created (Line 65). Later calls to $c_{\mathcal{E}}$`.receive` (Line 47) will request the next sequence number after the checkpoint.

$c_{\mathcal{E}}$`.move_window` (Line 63) will cause any $c_{\mathcal{E}}$`.receive` calls for an old sequence number to finish with a TooOld message and request a sequence number after the checkpoint on the next iteration. □

B.3.4. Execution Safety II

Theorem B.1.2 (repetition) (E-Safety II)**.** The system provides linearizability regarding requests from correct clients.

We begin by proving the following auxiliary lemma.

Lemma B.3.6. *When a client accepts a reply for its request, then that reply is correct and replies from correct execution replicas are identical.*

Proof. A client waits for replies (Line 11) from $f_e + 1$ different replicas of its execution group with the same content (Line 20 and 23), such that per failure assumption at least one of the replies is from a correct execution replica. As shown in CP-E-Equivalence, all correct execution replicas that process a request arrive at the same state and result. That result is either sent directly to the client (Line 58) or retrieved from $u[c]$ on a request retry (Line 37). □

We can now prove E-Safety II:

Proof. In order to prove that SPIDER provides linearizability, we have to show that requests issued at any point in time are always executed after all requests for which a client has accepted the reply, and that the execution follows the application's specification [118].

The latter part of the requirement was already shown in CP-E-Equivalence, which uses the fact that requests are executed (Line 55) in a total order. This also guarantees that at least one correct replica has processed the EXECUTE message for each sequence number. An executed request must have been delivered by the agreement protocol black box (see the proof in Appendix B.3.2 for E-Safety).

Assume that the execution replicas have executed request r which was ordered at sequence number s. Now let the execution replicas execute a request r' afterwards that was ordered at a sequence number s' with $s' < s$. However, as execution replicas only process requests in order, this contradicts the assumption that r was already executed. Thus, new requests are always ordered and executed at a sequence number higher than that of previously executed requests. Per Lemma B.3.6, a client cannot receive different replies from correct execution replicas.

That is, as soon as a single correct execution replica sends a reply to the client, which by construction happens before that client has accepted the reply, later requests are always ordered at a higher sequence number. □

Remark B.3.7. The request IRMCs do not matter for E-Safety and E-Safety II, as the agreement protocol black box is safe independent of the input.

Remark B.3.8. It is not necessary to store (unordered) client messages in an execution checkpoint as a correct client keeps repeating incomplete requests. Already executed requests are either part of a checkpoint or still available from the commit channel.

Remark B.3.9. A correct execution replica might not receive a request from a correct client when the other execution replicas have already processed it. This is the reason why `cp.stable` at execution replicas (Line 63) must push the window of a client's subchannel forward.

B.3.5. Execution Validity

Theorem B.1.3 (repetition) (E-Validity)**.** Only a correctly authenticated write request from a client may be executed.

E-Validity follows as a corollary:

Corollary B.3.10. *Per Lemma B.3.3, an executed request must have been delivered by the agreement protocol black box, and per A-Validity only valid client requests are delivered, thus together with the cryptographic assumptions the request must originate from that client.*

B.3.6. Execution Validity II

Theorem B.1.4 (repetition) (E-Validity II)**.** A correct replica executes a write request at most once.

Next, we prove E-Validity II:

Proof. This follows by construction of the main loop (Line 45): Requests that are not either the first request of a client or that do not have a higher counter value t_c than the last one are skipped (Line 54). After executing a request the latest counter for client c is stored (Line 56). As a request cannot have a counter value higher than its own counter value, it can be executed at most once. Per CP-E-Equivalence u and *app* are always restored together, such that if the application state contains the effects of executing the write request, this fact is also reflected in u. And therefore the request will not be executed more than once. □

B.3.7. Execution Liveness

Theorem B.1.5 (repetition) (E-Liveness)**.** A correct client will eventually receive a reply to its request.

We now prove that a correct client will eventually receive a reply to its request(s). Without loss of generality, we consider all requests to originate from the same client. For this we show that each of the processing steps a request passes through will eventually make progress. The lemmas assume implicitly that the client has either collected a stable reply (in which case the request processing is finished) or that it still waits for replies to its request and thus keeps resending its request.

Lemma B.3.11. *When a correct client sends a new request r, then an execution replica will pass it on to its request IRMC (unless it has already seen a newer request from that client).*

Proof. Assume that an execution replica receives a, from its perspective, new request (Line 33). By construction a request $r = \langle \textsc{Write}, w, c, t_c \rangle$ sent by a correct client is correctly authenticated and signed (Line 7) and therefore passes the MAC and signature checks (Line 34 and 39). The counter value t_c is $t_c > t'_c$, with t'_c being the counter value of any older request, as a correct client always increments its counter value after

accepting a reply (Line 14). As $t[c]$ is only modified when the execution replica receives a valid request from the client (Line 41), it must contain either some older value t'_c or the default of 0. (The client starts with $t_c = 1$, whereas an execution replica defaults to $t[c] = 0$.) Therefore, $t_c > t[c]$ and the execution replica calls $r_{\mathcal{E}}$.`send`(c, t_c, $\langle \text{REQUEST}, unwrap\_mac(m), \mathcal{E} \rangle$) (Line 43).

In case the request is not new to the execution replica, then this Lemma provides no assurances. □

Lemma B.3.12. *The send call by the execution replicas for the client's request channel will not block indefinitely.*

Proof. The $r_{\mathcal{E}}$.`send` (Line 43) call only blocks if the request counter t_c is larger than $\max(r_{\mathcal{E},c}.win)$, that is, the upper bound of the client's request subchannel. To arrive at a contradiction, assume that the $r_{\mathcal{E}}$.`send` call (Line 43) blocks indefinitely. As a correct client sends its (new) request to all execution replicas, eventually $f_e + 1$ correct execution replicas will per Lemma B.3.11 have called $r_{\mathcal{E}}$.`send` and therefore also $r_{\mathcal{E}}$.`move_window`(c, t_c) (Line 42) in the line before. Per IRMC-Liveness III eventually all agreement replicas will call $r_{\mathcal{E}}$.`move_window`(c, t_c). With IRMC-Liveness II it follows that $r_{\mathcal{E}}$.`send` returns, which contradicts the assumption. □

Lemma B.3.13. *An agreement replica will eventually try to receive a new correct request r from a correct client (unless it has already seen a newer one or skipped it with a checkpoint).*

Proof. Lemma B.3.12 has already shown that all ($\geq f_e + 1$) correct execution replicas will $r_{\mathcal{E}}$.`send` the new client request r which per IRMC-Liveness I can be received by a corresponding call on the agreement replicas, unless it is no longer part of the window of the subchannel. According to IRMC-Correctness I only request r can be received, as all correct execution replicas send this request. We therefore have to show that an agreement replica will call $r_{\mathcal{E}}$.`receive`(c, $t^+[c]$) (Line 80) for the right request counter value t_c.

Assume that $t^+[c] < t_c$: As shown above in the proof of Lemma B.3.12 all correct agreement replicas will eventually call $r_{\mathcal{E}}$.`move_window`(c, t_c), which according to the semantics of the `send` method will cause it to return $\langle \text{TOOOLD}, t_c \rangle$, which is used to update $t^+[c]$ (Line 81) and request t_c next.

Assume that $t^+[c] > t_c$: We show that this case never applies. An agreement replica cannot have received a too new TOOOLD message and stored its counter value (Line 81): According to IRMC-Correctness II, at least one correct execution replica must have called $r_{\mathcal{E}}$.`move_window` accordingly, which requires that a correct execution replica has received a valid request with counter $t^+[c] > t_c$ from a correct client. This contradicts the assumption that the request is new.

Incrementing $t^+[c]$ after having received a previous request (Line 84) or processing it in `ag.ordered` (Line 90) would require a previous request with counter value $t'_c \geq t_c$, which contradicts the assumption. (A faulty client could cause some chaos here, but this is no problem as the effects are strictly limited to the client's subchannel.) □

Remark B.3.14. These properties effectively allow the $r_{\mathcal{E}}$`.receive` call to synchronize itself. That is, each agreement replica will eventually try to receive the latest request for each client.

Lemma B.3.15. *The agreement protocol black box will call* `ag.ordered` *(Line 87) for a new request r at sequence number s within bounded time or apply a checkpoint for a later or equal sequence number.*

Proof. After $f_e + 1$ execution replicas complete their call to $r_{\mathcal{E}}$`.send`(c, t_c, r) (Line 43), an agreement replica can receive request r and start the agreement process.

Assume that the request r is not delivered within bounded time and is also not skipped via a checkpoint. The request of a correct client will eventually arrive at all correct ($\geq f_e + 1$) execution replicas. With Lemmas B.3.11 and B.3.12 it follows that $f_e + 1$ correct execution replicas call $r_{\mathcal{E}}$`.send`. With IRMC-Liveness I, IRMC-Correctness I and Lemma B.3.13 it follows that all correct agreement replicas will eventually receive the request r or a $\langle \textsc{TooOld}, t'_c \rangle$ message if $r_{\mathcal{E}}$`.move_window` (Line 42) is called by $f_e + 1$ execution replicas with $t'_c > t_c$. As a correct client does not issue a request with counter $t'_c > t_c$ before r was executed, all correct execution replicas will eventually call $r_{\mathcal{E}}$`.move_window` with exactly t_c, but no higher value, such that receiving TooOld would violate IRMC-Correctness II. (Executing r would require that it was delivered before by at least one correct agreement replica, as shown in the proof of Lemma B.3.3.)

Thus, per IRMC-Liveness III all correct agreement replicas will eventually internally call `move_window`(c, t_c) on the request IRMC and $2f_a + 1$ correct agreement replicas eventually $r_{\mathcal{E}}$`.receive` request r as long as r is not delivered via `ag.ordered`. Thus, the replicas call `ag.order_request()` (Line 83) to start the agreement for that request. With A-Liveness it follows that $f_a + 1$ correct agreement replicas eventually deliver r, contradicting the assumption.

Skipping the `ag.ordered` call via `cp.stable` (Line 97) requires per CP-Safety that at least one correct agreement replica created the checkpoint (Line 95) and thus the agreement protocol black box would already have delivered r, which contradicts the assumption. □

Lemma B.3.16. *A request r delivered at sequence number s that is $c_{\mathcal{E}}$`.send` by $f_a + 1$ correct agreement replicas will eventually either execute on $f_e + 1$ correct execution replicas or on one correct execution replica once a stable checkpoint with sequence number $s_{CP} \geq s$ was created.*

Proof. Assume that no stable checkpoint with sequence number $s_{CP} \geq s$ is applied at the execution replica (Line 62) before processing r: IRMC-Liveness I states that $f_e + 1$ correct execution replicas receive some request or a $\langle \textsc{TooOld}, s' \rangle$ message (Line 47) with $s' > s$, as $f_a + 1$ agreement replicas sent the request (Line 93). According to IRMC-Correctness I the request can only be request r, as per Lemma B.3.3 all correct agreement replicas send request r. The execution replicas cannot receive the TooOld message, as this would violate IRMC-Correctness II.

- Based on the assumption, execution replicas can only call $c_{\mathcal{E}}$.`move_window`$(0, s_{CP}+1)$ (Line 63) with $s_{CP} < s$, and thus $s_{CP}+1 \leq s$, which does not allow TooOld to be returned.

  As the agreement protocol black box delivers requests in sequence number order according to A-Order, an execution replica will also be able to receive any other previous request between s_{CP} and s and therefore will eventually try to receive s.

- Agreement replicas call $c_{\mathcal{E}}$.`move_window`$(0, \hat{s} - |c_{\mathcal{E},0}| + 1)$ (Line 99). To create an agreement checkpoint at $\hat{s}$ (Line 95), with $\hat{s} > s$, the window of the commit channel must have included $\hat{s}$ (as $c_{\mathcal{E}}$.`send` (Line 93) would have blocked otherwise), that is $max(c_{\mathcal{E},0}.win) \geq \hat{s} \Leftrightarrow min(c_{\mathcal{E},0}.win)+|c_{\mathcal{E},0}|-1 \geq \hat{s} \Leftrightarrow min(c_{\mathcal{E},0}.win) \geq \hat{s}-|c_{\mathcal{E},0}|+1$. That is, the lower bound of the commit channel window must have been larger or equal to $\hat{s} - |c_{\mathcal{E},0}| + 1$. Therefore, an agreement replica cannot advance the window of the commit channel by applying a checkpoint unless an execution group triggered the window move before. However, as shown in the previous paragraph the latter would contradict the assumption. Therefore, $f_e + 1$ correct execution replicas will eventually execute the request and possibly create a checkpoint.

Assume that a stable checkpoint with sequence number $s_{CP} \geq s$ gets applied: Per CP-Safety at least one correct execution replica must have created the checkpoint and thus have executed the request as per the previous part of the proof. Per CP-Liveness all other correct execution replicas will eventually receive and apply the checkpoint or have executed the request. □

Lemma B.3.17. *A correct execution checkpoint at sequence number s_{CP} for which $f_a + 1$ agreement replicas delivered and called $c_{\mathcal{E}}$.`send`$(0, s_{CP})$ (Line 93) will eventually become stable (Line 62) unless it is superseded by a newer one.*

Proof. Assume that no such stable checkpoint exists and that it is not superseded by a newer one. Then per Lemma B.3.16 $f_e + 1$ correct execution replicas will execute the request and thereby create their checkpoint messages (Line 60), which per CP-E-Equivalence are identical and according to CP-Liveness II will become stable. □

Lemma B.3.18. *If no progress occurs, then eventually the start of the subchannel window of the commit channel is $min(c_{\mathcal{E},0}.win) = s_{CP} + 1$ with s_{CP} being the latest stable execution checkpoint.*

Proof. Per CP-Liveness eventually all execution replicas will receive the latest stable execution checkpoint (Line 62) and call $c_{\mathcal{E}}$.`move_window`$(0, s_{CP} + 1)$ (Line 63). No correct execution replica calls $c_{\mathcal{E}}$.`move_window` for a higher sequence number as s_{CP} is the number of the latest checkpoint.

Agreement replicas call $c_{\mathcal{E}}$.`move_window`$(0, \hat{s} - |c_{\mathcal{E},0}| + 1)$ (Line 99). To create an agreement checkpoint at $\hat{s}$, the window of the commit channel must have included $\hat{s}$ (as $c_{\mathcal{E}}$.`send` (Line 93) would have blocked otherwise, preventing the checkpoint generation), that is $max(c_{\mathcal{E},0}.win) \geq \hat{s} \Leftrightarrow min(c_{\mathcal{E},0}.win)+|c_{\mathcal{E},0}|-1 \geq \hat{s} \Leftrightarrow min(c_{\mathcal{E},0}.win) \geq \hat{s}-|c_{\mathcal{E},0}|+$

1. Therefore, an agreement replica cannot advance the window of the commit channel to a sequence number that is larger than that of the execution replicas' $c_{\mathcal{E}}$`.move_window` calls. Thus, all correct agreement replicas eventually arrive at $min(c_{\mathcal{E},0}.win) = s_{CP} + 1$ with s_{CP} being the latest stable execution checkpoint. □

Lemma B.3.19. *Agreement replicas will eventually complete* $c_{\mathcal{E}}$`.send`(s, r) *(Line 93).*

Proof. `ag.ordered` blocks when *win* is full (Line 88). `AG-WIN` $\geq k_a$ and *win* is always anchored directly after the sequence number of the last stable agreement checkpoint. Thus, *win* contains at least one sequence number for which a new agreement checkpoint will be created.

Assume that `ag.ordered` blocks permanently on the window check. In that case, per assumption, there can be no stable agreement checkpoint with sequence number $s_{CP} \geq s$ and $s_{CP} \in win$, which would lead to progress. Therefore, as the client waits for r to be executed, per Lemma B.3.15 eventually $f_a + 1$ agreement replicas also deliver all requests in *win*. That is, $f_a + 1$ correct agreement replicas create a new agreement checkpoint, which will become stable and moves *win* forward. This contradicts the assumption.

Assume that the $c_{\mathcal{E}}$`.send` (Line 93) call blocks permanently, which requires that $s > max(c_{\mathcal{E},0}.win)$. Per A-Order and CP-A-Equivalence it follows that all previous slots in the subchannel window are filled with requests. With Lemma B.3.15 this applies to at least $f_a + 1$ agreement replicas. As $|c_{\mathcal{E},0}| \geq k_e$ at least one position in the commit channel window is an execution checkpoint sequence number. Per Lemma B.3.17 this causes a new checkpoint to become stable, which according to Lemma B.3.18 eventually moves the commit channel window forward and thus contradicts the assumption. □

Now we can prove that a correct client will eventually receive a reply to its request:

Proof. Assume that the client does not get a reply. Then per Lemmas B.3.16 and B.3.19 $f_e + 1$ correct execution replicas will eventually have the reply in $u[c]$. As a correct client does not send a new request before having obtained a reply to the last one, $u[c]$ must eventually contain the reply. Per CP-E-Equivalence the reply is identical on all correct execution replicas. At latest after the next request retry, the client will receive the (identical) reply from $f_e + 1$ correct execution replicas and therefore accept the reply (Line 23), which contradicts the assumption. □

Remark B.3.20. An agreement replica will receive a request r either via the request IRMC, the agreement protocol black box or skip the request via a checkpoint.

B.3.8. Multiple Execution Groups

We now generalize to $n_e \geq 1$ execution groups of which $z < n_e$ might be skipped if these are slow.

Lemma B.3.21. *E-Liveness also holds for multiple execution groups.*

Proof. Even though an agreement replica only waits for $n_e - z$ groups (Line 94) to complete $c_{\mathcal{E}}$`.send`, an execution group will only miss requests if the agreement replicas call $c_{\mathcal{E}}$`.move_window` (Line 99) with a sequence number not yet received by a slow execution group. As shown in the proof of Lemma B.3.18 an agreement replica can only create a checkpoint that would push the window of the commit channel forward if the execution group already has created a newer or matching checkpoint. Generalized to n_e execution groups, the $c_{\mathcal{E}}$`.send` (Line 93) calls for $n_e - z$ execution groups have to complete, before an agreement checkpoint can be created (Line 95). Therefore, an execution group that has fallen behind can always retrieve an up-to-date checkpoint from one of the $n_e - z$ up-to-date execution groups.

As agreement replicas unconditionally move the commit channel window forward (Line 99), this will lead to at least $f_a + 1$ agreement replicas calling $c_{\mathcal{E}}$`.move_window` (per Lemma B.3.19 a corresponding checkpoint will eventually exist and according to CP-Liveness all correct agreement replicas will eventually receive it), which based on IRMC-Liveness I and IRMC-Liveness III will eventually allow execution groups that fell behind to receive a TooOld message. This triggers fetching an up-to-date checkpoint (Line 50). □

B.3.9. Consistency Guarantees

We now revisit the consistency guarantees provided by Spider.

Write Requests As previously shown in Appendix B.3.4, Spider provides linearizability for write requests.

Read Requests with Strong Consistency Read requests with strong consistency work like write requests with one exception: Only the designated execution group receives the full request, whereas the other groups only get the client id c and counter t_c. This leads to the following observation:

Lemma B.3.22. *With read requests, the content of checkpoints can vary between groups in regard to the reply stored in $u[c]$. That is, CP-E-Equivalence only applies to individual groups at a time.*

Proof. Only the client's execution group will receive the read request and modify $u[c]$ accordingly after executing the request (Line 56). All other execution groups store a placeholder in $u[c]$ which includes the request counter. Therefore, the reply parts of $u[c]$ can differ between groups. Note that this divergence is self-correcting in the sense that it will disappear after executing the next write request for that client. □

Remark B.3.23. This does not prevent the checkpoint from being transferred between groups, as each group can still generate a valid proof for its checkpoint. However, the global flow control could force a group to skip some requests, which might include group-specific read requests. In that case an execution replica has to tell the client to resubmit its request if only a placeholder is stored in $u[c]$. This does not affect consistency as read requests do not modify the application state.

```
1  rwin[r][sc] := [1, |IRMC_sc|]                    // Received windows, r ∈ R_R ∪ r_s
2  cwin[sc] := [1, |IRMC_sc|]                       // Combined window, f+1-highest received window
3  VOID send(SUBCHANNEL sc, POSITION p, MESSAGE m):
4      sleep until p ≤ max(cwin[sc])
5      if p ≥ min(cwin[sc]): // p ∈ cwin[sc]
6          send sign_{r_s}(⟨SEND, m, sc, p⟩) to R_R

8  VOID move_window(SUBCHANNEL sc, POSITION p):
9      // The subchannel window start may only increase
10     if p > min(rwin[r_s][sc]):
11         // Send and store window move
12         send mac_{r_s,R_R}(⟨MOVE, sc, p⟩) to R_R
13         rwin[r_s][sc] := [p, p + |IRMC_sc| − 1]

15 on receive(m = ⟨MOVE, sc, p⟩ from r_r ∈ R_R):
16     if !valid_mac_{r_r,R_S}(m): return
17     // Only accept new move messages
18     if p > min(rwin[r_r][sc]):
19         rwin[r_r][sc] := [p, p + |IRMC_sc| − 1]
20         // Calculate actual window start
21         w := f_r + 1 highest {min(rwin[r'_r][sc]) | r'_r ∈ R_R}
22         cwin[sc] := [w, w + |IRMC_sc| − 1]
23         garbage collect messages with SEQNR s < cwin[sc]
```

Figure B.8: IRMC-RC sender endpoint at replica r_s (pseudocode)

Read Requests with Weak Consistency

Lemma B.3.24. *Weakly consistent read requests provide prefix consistency.*

Proof. All write requests are totally ordered. As a correct client only accepts a result that is sent by at least one correct execution replica, the result will correspond to some point in this total order.

□

B.4. IRMC Pseudocode

In this section we provide pseudocode for the IRMC-RC and IRMC-SC, which have been described in Sections 5.6.3 and 5.6.4.

B.4.1. IRMC-RC

The IRMC-RC variant shown in Figures B.8 and B.9 is a simple implementation of the sender and receiver endpoint. In case a sender replica has multiple IRMCs and sends

```
24 rwin[r][sc] = [1, |IRMC_sc|]                    // Received windows, r ∈ R_S ∪ r_r
25 cwin[sc] = [1, |IRMC_sc|]                       // Combined window
26 d[sc][p][r_s] = ∅                               // Messages received via SEND messages
---
27 MESSAGE receive(SUBCHANNEL sc, POSITION p):
28     sleep until p ≤ max(cwin[sc])
29     sleep until either:
30         case p < min(cwin[sc]):
31             return ⟨TOOOLD, min(cwin[sc])⟩
32         case ∃m : |{r_s | m = d[sc][p][r_s], r_s ∈ R_S}| ≥ f_s + 1:
33             return m // Received m from at least f_s + 1 senders

35 VOID move_window(SUBCHANNEL sc, POSITION p):
36     // The subchannel window start may only increase
37     if p > min(cwin[sc]):
38         send mac_{r_r,R_S}(⟨MOVE, sc, p⟩) to R_S
39         cwin[sc] := [p, p + |IRMC_sc| − 1]
40         garbage collect messages in endpoint state with SEQNR s < cwin[sc]

42 on receive(r = ⟨SEND, m, sc, p⟩ from r_s ∈ R_S):
43     if !valid_sig_{R_S}(r): return
44     if p ≥ min(cwin[sc]):
45         d[sc][p][r_s] := m

47 on receive(m = ⟨MOVE, sc, p⟩ from r_s ∈ R_S):
48     if !valid_mac_{r_s,R_R}(m): return
49     // Only accept new move messages
50     if p > min(rwin[r_s][sc]):
51         rwin[r_s][sc] := [p, p + |IRMC_sc| − 1]
52         nw := f_s + 1 highest {min(rwin[r'_s][sc]) | r'_s ∈ R_S}
53         if nw > min(cwin[sc]):
54             move_window(s, nw)
```

Figure B.9: IRMC-RC receiver endpoint at replica r_r (pseudocode)

identical messages on the same subchannel and position, then these send the same SEND message, which allows the IRMCs to share the message's signature. For a more detailed explanation of signature sharing refer to Section 5.7.1.

Without loss of generality, we assume the set of senders R_S and receivers R_R to be disjoint, that is, $R_S \cap R_R = \varnothing$. We assume reliable point-to-point channels between replicas, that is, messages sent between individual replicas will be delivered eventually, unless messages are garbage collected at which point a replica discards old messages, even when they were not successfully delivered yet. This can be achieved using the outbox abstraction described in Section 5.6.2 and letting the `send` calls enqueue the message at the corresponding message or acknowledgement outbox. Thereby the outboxes can limit the transmission of messages to each receiver such that only messages are sent which fit into the subchannel window at a receiver. The receiver replicas then drop received messages outside the current subchannel window and thus can bound their state.

To simplify the presentation, the messages do not include an IRMC identifier. Such an identifier has to be added in case it becomes necessary to differentiate between multiple IRMC instances.

B.4.2. IRMC-SC

IRMC-SC shown in Figures B.10 and B.11 is a more complex but also more efficient implementation than IRMC-RC.

For liveness, we assume that the MOVE message is protected against replay attacks, for example, by including a counter to filter out already processed instances of the message. In case a sender replica has multiple IRMCs and sends identical messages on the same subchannel and position, then it can share a single signed CERTIFICATE message between IRMCs.

```
1  + Variables from IRMC-RC sender endpoint
2  sig[sc][p][r_s] = ∅          // SigShare from sender r_s for subchannel sc at position p
3  bundle[sc][p] = ∅                    // Certificate for subchannel sc at position p
4  sender[sc][r_r] = ⊥                  // Selected sender for subchannel sc to receiver r_r
5  d[sc][p] = ∅                                  // Message sent in subchannel sc at position p
6  void send(Subchannel sc, Position p, Message m):
7      sleep until p ≤ max(cwin[sc])
8      if p ≥ min(cwin[sc]):                                                  // p ∈ cwin[sc]
9          d[sc][p] := m
10         // SigShare is also processed locally
11         send sign_{r_s}(⟨SigShare, h(m), sc, p⟩) to R_S

13 // Collect SigShares to assemble a Certificate
14 on receive(sg = ⟨SigShare, h(m), sc, p⟩ from r_s ∈ R_S):
15     if !valid_sig_{R_S}(sg): return
16     if p ≥ min(cwin[sc]) ∧ sig[sc][p][r_s] = ∅:      // Only accept first share per sender
17         sig[sc][p][r_s] := sg
18         // Shares with matching hash
19         v := {sig[sc][p][r] | r ∈ R_S, sig[sc][p][r].h = h(m)}
20         limit v to f_s + 1 values
21         // Check if replica has f_s + 1 matching shares and the actual request
22         if |v| = f_s + 1 ∧ d[sc][p] ≠ ∅ ∧ bundle[sc][p] = ∅:
23             bundle[sc][p] := mac_{r_s,R_R}(⟨Certificate, d[sc][p], sc, p, v⟩)
24             send bundle[sc][p] to all receivers r_r with sender[sc][r_r] = r_s

26 periodic:
27     // Send position of latest certificate per subchannel up to which there are no gaps at
           previous positions in the subchannel window
28     for each Subchannel sc:
29         prog[sc] := highest p ∈ cwin[sc]
               with ∀p' ∈ cwin[sc], p' ≤ p : bundle[sc][p'] ≠ ∅
30     send mac_{r_s,R_R}(⟨Progress, prog⟩) to R_R

32 // move_window and receive(Move) are identical to IRMC-RC

34 // Select sender for subchannel
35 on receive(m = ⟨Select, sc, s⟩ from r_r ∈ R_R):
36     if !valid_mac_{r_r,R_S}(m): return
37     sender[sc][r_r] := s
38     // Send queued messages for subchannel sc to r_r
39     ∀p : send bundle[sc][p] to receiver r_r if s = r_s
```

Figure B.10: IRMC-SC sender endpoint at replica r_s (pseudocode)

```
40 + Variables from IRMC-RC receiver endpoint
41 d[sc][p] = ∅                          // Message received for subchannel sc at position p
42 pe[r][sc] := 0                        // Expected progress for sc reported by r ∈ R_S
43 pm[sc] := 0                           // Merged progress values for sc (f_s + 1 highest)
44 MESSAGE receive(SUBCHANNEL sc, POSITION p):
45     sleep until p ≤ max(cwin[sc])
46     sleep until either:
47         case p < min(cwin[sc]):
48             return ⟨TOOOLD, min(cwin[sc])⟩
49         case d[sc][p] ≠ ∅:
50             return d[sc][p]

52 on receive(r = ⟨CERTIFICATE, m, sc, p, v⟩ from r_s ∈ R_S):
53     if !valid_mac_{r_s,R_R}(r): return
54     // Certificate must contain f_s + 1 matching signatures from different senders
55     if p ≥ min(cwin[sc]) ∧ |v| = f_s + 1 ∧ ∀sg ∈ v : valid_sig_{R_S}(sg) for m
           ∧ sg from different senders:
56         d[sc][p] := m

58 on receive(m = ⟨PROGRESS, np⟩ from r_s ∈ R_S):
59     if !valid_mac_{r_s,R_R}(m): return
60     // Merge progress vectors
61     for each SUBCHANNEL sc ∈ np:
62         pe[r_s][sc] := max(pe[r_s][sc], np[sc])
63         pm[sc] := f_s + 1 highest {pe[r'][sc] | r' ∈ R_S}
64         // Start timeout if some messages are still missing
65         if ∃s' ∈ [min(cwin[sc]), pm[sc]] : d[sc][s'] = ∅:
66             p := pm[sc]
67             // sc@p is a timer for subchannel sc at the position p
68             start timer for sc@p if not started yet

70 on timeout for sc@p:
71     // Timeout expired and there are still missing certificates
72     if ∃s' ∈ [min(cwin[sc]), p] : d[sc][s'] = ∅:
73         select new sender r_s for sc
74         send mac_{r_r,R_S}(⟨SELECT, sc, r_s⟩) to R_S
75         restart timer for sc@p

77 // move_window and receive(MOVE) are identical to IRMC-RC
```

Figure B.11: IRMC-SC receiver endpoint at replica r_r (pseudocode)

List of Acronyms

COTS	commercial of-the-shelf
COW	copy-on-write
CRDT	conflict-free replicated data type
DDFC	Differential Deterministic Fuzzy Checkpointing
DFC	Deterministic Fuzzy Checkpointing
DNS	domain name system
DRC	default-request certificate
FIFO	first-in-first-out
FPC	fast-path certificate
GPS	global positioning system
HMAC	hash-based message authentication code
IRMC	Inter-Regional Message Channel
IRMC-RC	Inter-Regional Message Channel with Receiver-side Collection
IRMC-SC	Inter-Regional Message Channel with Sender-side Collection
MAC	message authentication code
RPC	reconciliation-path certificate
RSM	replicated state machine
SCC	strongly connected component
SSCC	special-case strongly connected component
TCP	transmission control protocol
TLS	transport layer security
YCSB	Yahoo! Cloud Serving Benchmark

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. "Fault-Scalable Byzantine Fault-Tolerant Services." In: *Proceedings of the 20th Symposium on Operating Systems Principles*. SOSP '05. 2005, pages 59–74. DOI: 10.1145/1095810.1095817.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. *Revisiting Fast Practical Byzantine Fault Tolerance*. arXiv. 2017. DOI: 10.48550/ARXIV.1712.01367.

[3] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. *Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma*. arXiv. 2018. DOI: 10.48550/ARXIV.1801.10022.

[4] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. *Malicious Security Comes for Free in Consensus with Leaders*. Cryptology ePrint Archive, Paper 2020/1480. 2020. URL: https://eprint.iacr.org/2020/1480.

[5] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. "RACS: A Case for Cloud Storage Diversity." In: *Proceedings of the 1st Symposium on Cloud Computing*. SoCC '10. 2010, pages 229–240. DOI: 10.1145/1807128.1807165.

[6] Matt Adorjan. *AWS Latency Monitoring*. 2023. URL: https://www.cloudping.co/grid (visited on 06/03/2023).

[7] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. "WPaxos: Wide Area Network Flexible Consensus." In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (Jan. 2020), pages 211–223. DOI: 10.1109/TPDS.2019.2929793.

[8] Aiswarya Lakshmi. *AWS Buys Hawaiki Submarine Cable*. 2016. URL: https://www.marinetechnologynews.com/news/hawaiki-submarine-cable-532845 (visited on 06/03/2023).

[9] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. "Matrix Signatures: From MACs to Digital Signatures in Distributed Systems." In: *Proceedings of the 22nd International Symposium on Distributed Computing*. DISC '08. 2008, pages 16–31. DOI: 10.1007/978-3-540-87779-0_2.

[10] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Efficient State-Based CRDTs by Delta-Mutation." In: *Proceedings of the 3rd International Conference on Networked Systems*. NETYS '15. 2015, pages 62–76. DOI: 10.1007/978-3-319-26850-7_5.

[11] Amazon EC2. *AWS Service Health Dashboard - Amazon S3 Availability Event: July 20, 2008*. 2008. URL: https://web.archive.org/web/20220403060108/https://status.aws.amazon.com/s3-20080720.html (visited on 06/03/2023).

[12] Amazon EC2. *Summary of the AWS Service Event in the Sydney Region*. 2016. URL: https://aws.amazon.com/en/message/4372T8/ (visited on 06/03/2023).

[13] Amazon Elastic Compute Cloud. *Set the time for your Linux instance*. 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html (visited on 06/03/2023).

[14] Amazon Elastic Compute Cloud. *Regions and Availability Zones*. 2023. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html (visited on 06/03/2023).

[15] Amazon Web Services. *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. 2011. URL: https://aws.amazon.com/message/65648/ (visited on 06/03/2023).

[16] Amazon Web Services. *Amazon Compute Service Level Agreement*. 2022. URL: https://aws.amazon.com/compute/sla/ (visited on 06/03/2023).

[17] Amazon Web Services. *AWS Transit Gateway Pricing*. 2023. URL: https://aws.amazon.com/transit-gateway/pricing/ (visited on 06/03/2023).

[18] Amazon Web Services. *EC2 On-Demand Instance Pricing*. 2023. URL: https://aws.amazon.com/ec2/pricing/on-demand/ (visited on 06/03/2023).

[19] Amazon Web Services. *Global Infrastructure Regions & AZs*. 2023. URL: https://aws.amazon.com/about-aws/global-infrastructure/regions_az/ (visited on 06/03/2023).

[20] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. "Customizable Fault Tolerance for Wide-Area Replication." In: *Proceedings of the 26th International Symposium on Reliable Distributed Systems*. SRDS '07. 2007, pages 65–82. DOI: 10.1109/SRDS.2007.40.

[21] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. *Customizable Fault Tolerance for Wide-Area Replication*. Technical report CNDS-2007-1. Distributed Systems and Networks Lab, Computer Science Department, Johns Hopkins University, 2007. URL: http://www.cnds.jhu.edu/pub/papers/cnds-2007-1.pdf.

[22] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. "Prime: Byzantine Replication Under Attack." In: *IEEE Transactions on Dependable and Secure Computing* 8.4 (2010), pages 564–577. DOI: 10.1109/TDSC.2010.70.

[23] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. *ENISA THREAT LANDSCAPE 2021*. Technical report. European Union Agency for Cybersecurity (ENISA), 2021. DOI: 10.2824/324797.

[24] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. "Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks." In: *IEEE Transactions on Dependable and Secure Computing* 7.1 (2010), pages 80–93. DOI: 10.1109/TDSC.2008.53.

[25] Andrew Ayer. *Yeti 2022 not furnishing entries for STH 65569149*. 2021. URL: https://groups.google.com/a/chromium.org/g/ct-policy/c/PCkKU357M2Q/ (visited on 06/03/2023).

[26] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains." In: *Proceedings of the 13th European Conference on Computer Systems*. EuroSys '18. 2018. DOI: 10.1145/3190508.3190538.

[27] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. "Leaderless Consensus." In: *Proceedings of the 41th International Conference on Distributed Computing Systems*. ICDCS '21. 2021, pages 392–402. DOI: 10.1109/ICDCS51616.2021.00045.

[28] Todd Arnold, Jia He, Weifan Jiang, Matt Calder, Italo Cunha, Vasileios Giotsas, and Ethan Katz-Bassett. "Cloud Provider Connectivity in the Flat Internet." In: *Proceedings of the 20th Internet Measurement Conference*. IMC '20. 2020, pages 230–246. DOI: 10.1145/3419394.3423613.

[29] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. "Speeding up Consensus by Chasing Fast Decisions." In: *Proceedings of the 47th International Conference on Dependable Systems and Networks*. DSN '17. 2017, pages 49–60. DOI: 10.1109/DSN.2017.35.

[30] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. "ezBFT: Decentralizing Byzantine fault-tolerant state machine replication." In: *Proceedings of the 39th International Conference on Distributed Computing Systems*. ICDCS '19. 2019, pages 565–577. DOI: 10.1109/ICDCS.2019.00063.

[31] Hagit Attiya and Jennifer L. Welch. "Sequential Consistency versus Linearizability." In: *ACM Transactions on Computer Systems* 12.2 (May 1994), pages 91–122. DOI: 10.1145/176575.176576.

[32] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. "The Next 700 BFT Protocols." In: *ACM Transactions on Computer Systems* 32.4 (2015), 12:1–12:45. DOI: 10.1145/2658994.

[33] Algirdas Avižienis. "The N-version Approach to Fault-tolerant Software." In: *IEEE Transactions on Software Engineering* SE-11.12 (1985), pages 1491–1501. DOI: 10.1109/TSE.1985.231893.

[34] Baruch Awerbuch and Christian Scheideler. "Group Spreading: A Protocol for Provably Secure Distributed Name Service." In: *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*. ICALP '04. 2004, pages 183–195. DOI: 10.1007/978-3-540-27836-8_18.

[35] Azure DevOps. *Postmortem – VSTS Outage – 4 September 2018*. 2018. URL: https://devblogs.microsoft.com/devopsservice/?p=17485 (visited on 06/03/2023).

[36] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. "Making BFT Protocols Really Adaptive." In: *Proceedings of the 29th International Parallel and Distributed Processing Symposium*. IPDPS '15. 2015, pages 904–913. DOI: 10.1109/IPDPS.2015.21.

[37] Elaine Barker. *Recommendation for Key Management: Part 1 – General*. Technical report NIST Special Publication (SP) 800-57 Part 1 Rev. 5. National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-57pt1r5.

[38] Rida Bazzi and Maurice Herlihy. "Clairvoyant State Machine Replication." In: *Information and Computation* 285 (2021). DOI: 10.1016/j.ic.2021.104701.

[39] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Consensus-Oriented Parallelization: How to Earn Your First Million." In: *Proceedings of the 16th International Middleware Conference*. Middleware '15. 2015, pages 173–184. DOI: 10.1145/2814576.2814800.

[40] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Hybrids on Steroids: SGX-Based High Performance BFT." In: *Proceedings of the 12th European Conference on Computer Systems*. EuroSys '17. 2017, pages 222–237. DOI: 10.1145/3064176.3064213.

[41] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services." In: *Proceedings of the 19th Symposium on Operating Systems Principles*. SOSP '03. 2003, pages 268–281. DOI: 10.1145/945445.945471.

[42] Christian Berger, Hans P. Reiser, and Alysson Bessani. "Making Reads in BFT State Machine Replication Fast, Linearizable, and Live." In: *Proceedings of the 40th International Symposium on Reliable Distributed Systems*. SRDS '21. 2021, pages 1–12. DOI: 10.1109/SRDS53918.2021.00010.

[43] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani. "Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication." In: *Proceedings of the 38th International Symposium on Reliable Distributed Systems*. SRDS '19. 2019, pages 183–192. DOI: 10.1109/SRDS47363.2019.00029.

[44] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. "From Byzantine Replication to Blockchain: Consensus is Only the Beginning." In: *Proceedings of the 50th International Conference on Dependable Systems and Networks*. DSN '20. 2020, pages 424–436. DOI: 10.1109/DSN48063.2020.00057.

[45] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds." In: *ACM Transactions on Storage (TOS)* 9.4 (2013), 12:1–12:33. DOI: 10.1145/2535929.

[46] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. "On the Efficiency of Durable State Machine Replication." In: *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX ATC '13. 2013, pages 169–180.

[47] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. "State Machine Replication for the Masses with BFT-SMaRt." In: *Proceedings of the 44th International Conference on Dependable Systems and Networks*. DSN '14. 2014, pages 355–362. DOI: 10.1109/DSN.2014.43.

[48] Loïck Bonniot, Christoph Neumann, and François Taïani. "PnyxDB: a Lightweight Leaderless Democratic Byzantine Fault Tolerant Replicated Datastore." In: *Proceedings of the 39th International Symposium on Reliable Distributed Systems*. SRDS '20. 2020, pages 155–164. DOI: 10.1109/SRDS51746.2020.00023.

[49] Gabriel Bracha and Sam Toueg. "Asynchronous Consensus and Broadcast Protocols." In: *Journal of the ACM* 32.4 (Oct. 1985), pages 824–840. DOI: 10.1145/4221.214134.

[50] Doug Brake. *Submarine Cables: Critical Infrastructure for Global Communications*. Technical report. Information Technology & Innovation Foundation, 2019. URL: https://www2.itif.org/2019-submarine-cables.pdf.

[51] Marc Brooker, Tao Chen, and Fan Ping. "Millions of Tiny Databases." In: *Proceedings of the 17th Symposium on Networked Systems Design and Implementation*. NSDI '20. 2020, pages 463–478.

[52] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. 2006, pages 335–350.

[53] C. Cachin and A. Samar. "Secure distributed DNS." In: *Proceedings of the 34th International Conference on Dependable Systems and Networks*. DSN '04. 2004, pages 423–432. DOI: 10.1109/DSN.2004.1311912.

[54] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency." In: *Proceedings of the 23rd Symposium on Operating Systems Principles*. SOSP '11. 2011, pages 143–157. DOI: 10.1145/2043556.2043571.

[55] Carlos Carvalho, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani. "Dynamic Adaptation of Byzantine Consensus Protocols." In: *Proceedings of the 33rd Symposium on Applied Computing*. SAC '18. 2018, pages 411–418. DOI: 10.1145/3167132.3167179.

[56] Miguel Castro. "Practical Byzantine Fault Tolerance." PhD thesis. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA, 2001.

[57] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance." In: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. OSDI '99. 1999, pages 173–186.

[58] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery." In: *ACM Transactions on Computer Systems* 20.4 (2002), pages 398–461. DOI: 10.1145/571637.571640.

[59] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. "BASE: Using Abstraction to Improve Fault Tolerance." In: *ACM Transactions on Computer Systems* 21.3 (2003), pages 236–269. DOI: 10.1145/859716.859718.

[60] Tarcisio Ceolin, Fernando Dotti, and Fernando Pedone. "Parallel State Machine Replication from Generalized Consensus." In: *Proceedings of the 39th International Symposium on Reliable Distributed Systems*. SRDS '20. 2020, pages 133–142. DOI: 10.1109/SRDS51746.2020.00021.

[61] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live migration of virtual machines." In: *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. NSDI '05. 2005, pages 273–286.

[62] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. "UpRight Cluster Services." In: *Proceedings of the 22nd Symposium on Operating Systems Principles*. SOSP '09. 2009, pages 277–290. DOI: 10.1145/1629575.1629602.

[63] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults." In: *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*. NSDI '09. 2009, pages 153–168.

[64] Vinicius V. Cogo, André Nogueira, João Sousa, Marcelo Pasin, Hans P. Reiser, and Alysson Bessani. "FITCH: Supporting Adaptive Replicated Services in the Cloud." In: *Proceedings of the 13th International Conference on Distributed Applications and Interoperable Systems*. DAIS '13. 2013, pages 15–28. DOI: 10.1007/978-3-642-38541-4_2.

[65] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform." In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pages 1277–1288. DOI: 10.14778/1454159.1454167.

[66] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st Symposium on Cloud Computing*. SoCC '10. 2010, pages 143–154. DOI: 10.1145/1807128.1807152.

[67] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally Distributed Database." In: *ACM Transactions on Computer Systems* 31.3 (Aug. 2013). DOI: 10.1145/2491245.

[68] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. 2006, pages 177–190.

[69] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. "DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains." In: *Proceedings of the 17th International Symposium on Network Computing and Applications*. NCA '18. 2018. DOI: 10.1109/NCA.2018.8548057.

[70] Data Center Knowledge. *Massive Flooding Damages Several NYC Data Centers*. 2012. URL: https://www.datacenterknowledge.com/archives/2012/10/30/major-flooding-nyc-data-centers (visited on 06/03/2023).

[71] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale." In: *Communications of the ACM* 56.2 (2013), pages 74–80. DOI: 10.1145/2408776.2408794.

[72] Tobias Distler. "Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective." In: *ACM Computing Surveys* 54.1 (2021). DOI: 10.1145/3436728.

[73] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. "Resource-efficient Byzantine Fault Tolerance." In: *IEEE Transactions on Computers* 65.9 (2016), pages 2807–2819. DOI: 10.1109/TC.2015.2495213.

[74] Tobias Distler and Rüdiger Kapitza. "Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency." In: *Proceedings of the 6th European Conference on Computer Systems*. EuroSys '11. 2011, pages 91–105. DOI: 10.1145/1966445.1966455.

[75] Tobias Distler, Rüdiger Kapitza, Ivan Popov, Hans P. Reiser, and Wolfgang Schröder-Preikschat. "SPARE: Replicas on Hold." In: *Proceedings of the 18th Network and Distributed System Security Symposium*. NDSS '11. 2011, pages 407–420.

[76] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. "State Transfer for Hypervisor-Based Proactive Recovery of Heterogeneous Replicated Services." In: *Proceedings of the 5th "Sicherheit, Schutz und Zuverlässigkeit" Conference*. SICHERHEIT '10. 2010, pages 61–72.

[77] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. *Silent Data Corruptions at Scale*. arXiv. 2021. DOI: 10.48550/ARXIV.2102.11245.

[78] M. Duke, R. Braden, W. Eddy, E. Blanton, and A. Zimmermann. *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*. RFC 7414. RFC Editor, Feb. 2015. DOI: 10.17487/RFC7414.

[79] Partha Dutta, Rachid Guerraoui, and Marko Vukolić. *Best-case complexity of asynchronous Byzantine consensus*. Technical report 200499. 2005. URL: http://lpdwww.epfl.ch/upload/documents/publications/567931850DGV-feb-05.pdf.

[80] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony." In: *Journal of the ACM* 35.2 (1988), pages 288–323. DOI: 10.1145/42282.42283.

[81] Michael Eischer, Markus Büttner, and Tobias Distler. "Deterministic Fuzzy Checkpoints." In: *Proceedings of the 38th International Symposium on Reliable Distributed Systems*. SRDS '19. 2019, pages 153–162. DOI: 10.1109/SRDS47363.2019.00026.

[82] Michael Eischer and Tobias Distler. "Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems." In: *Proceedings of the 1st Workshop on Byzantine Consensus and Resilient Blockchains*. BCRB '18. 2018, pages 140–145. DOI: 10.1109/DSN-W.2018.00053.

[83] Michael Eischer and Tobias Distler. "Efficient Checkpointing in Byzantine Fault-Tolerant Systems." In: *Tagungsband des FB-SYS Herbsttreffens 2019*. 2019. DOI: 10.18420/fbsys2019-01.

[84] Michael Eischer and Tobias Distler. "Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers." In: *Computing* 101.2 (2019), pages 97–118. DOI: 10.1007/s00607-018-0652-3.

[85] Michael Eischer and Tobias Distler. "Resilient Cloud-Based Replication with Low Latency." In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. 2020, pages 14–28. DOI: 10.1145/3423211.3425689. (Best student paper).

[86] Michael Eischer and Tobias Distler. *Resilient Cloud-based Replication with Low Latency (Extended Version)*. arXiv. 2020. DOI: 10.48550/ARXIV.2009.10043.

[87] Michael Eischer and Tobias Distler. "Egalitarian Byzantine Fault Tolerance." In: *Proceedings of the 26th Pacific Rim International Symposium on Dependable Computing*. PRDC '21. 2021, pages 1–10. DOI: 10.1109/PRDC53464.2021.00019.

[88] Michael Eischer and Tobias Distler. *Egalitarian Byzantine Fault Tolerance (Extended Version)*. arXiv. 2021. DOI: 10.48550/arXiv.2109.06811.

[89] Michael Eischer, Benedikt Straßner, and Tobias Distler. "Low-Latency Geo-Replicated State Machines with Guaranteed Writes." In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '20. 2020. DOI: 10.1145/3380787.3393686.

[90] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. "State-Machine Replication for Planet-Scale Systems." In: *Proceedings of the 15th European Conference on Computer Systems*. EuroSys '20. 2020. DOI: 10.1145/3342195.3387543.

[91] Bo Fang, Panruo Wu, Qiang Guan, Nathan DeBardeleben, Laura Monroe, Sean Blanchard, Zhizong Chen, Karthik Pattabiraman, and Matei Ripeanu. "SDC is in the Eye of the Beholder: A Survey and Preliminary Study." In: *Proceedings of the 46th International Conference on Dependable Systems and Networks Workshop*. DSN-W '16. 2016, pages 72–76. DOI: 10.1109/DSN-W.2016.46.

[92] Mark Filer, Jamie Gaudette, Yawei Yin, Denizcan Billor, Zahra Bakhtiari, and Jeffrey L. Cox. "Low-margin optical networking at cloud scale [Invited]." In: *J. Opt. Commun. Netw.* 11.10 (Oct. 2019), pages C94–C108. DOI: 10.1364/JOCN.11.000C94.

[93] M. J. Fischer, Nancy A. Lynch, and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." In: *Journal of the ACM* 32.2 (1985), pages 374–382. DOI: 10.1145/3149.214121.

[94] R. Friedman and R. van Renesse. "Packing messages as a tool for boosting the performance of total ordering protocols." In: *Proceedings of the 6th International Symposium on High Performance Distributed Computing*. HPDC '97. 1997, pages 233–242. DOI: 10.1109/HPDC.1997.626423.

[95] Miguel Garcia, Alysson Bessani, and Nuno Neves. "Lazarus: Automatic Management of Diversity in BFT Systems." In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. 2019, pages 241–254. DOI: 10.1145/3361525.3361550.

[96] Miguel Garcia, Alysson Neves Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. "OS Diversity for Intrusion Tolerance: Myth or Reality?" In: *Proceedings of the 41st International Conference on Dependable Systems and Networks*. DSN '11. 2011, pages 383–394. DOI: 10.1109/DSN.2011.5958251.

[97] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." In: *SIGACT News* 33.2 (June 2002), pages 51–59. DOI: 10.1145/564585.564601.

[98] Google Cloud. *Compute Engine Service Level Agreement (SLA)*. 2021. URL: https://cloud.google.com/compute/sla (visited on 06/03/2023).

[99] Google Cloud. *Encryption in transit*. 2023. URL: https://cloud.google.com/docs/security/encryption-in-transit#physical_boundaries (visited on 06/03/2023).

[100] Google Cloud. *Global Locations - Regions & Zones*. 2023. URL: https://cloud.google.com/about/locations/#network (visited on 06/03/2023).

[101] Google Cloud. *Regions and Zones - Compute Engine Documentation*. 2023. URL: https://cloud.google.com/compute/docs/regions-zones/ (visited on 06/03/2023).

[102] Google Cloud. *VPC networks*. 2023. URL: https://cloud.google.com/vpc/docs/vpc#specifications (visited on 06/03/2023).

[103] Google Developers. *Public NTP*. 2023. URL: https://developers.google.com/time/ (visited on 06/03/2023).

[104] Greg Linden. *Make Data Useful*. 2006. URL: http://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html (visited on 06/03/2023).

[105] Greg Linden. *Marissa Mayer at Web 2.0*. 2006. URL: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html (visited on 06/03/2023).

[106] Ilya Grigorik. *High Performance Browser Networking*. 1st. O'Reilly Media, Inc., 2013. Chapter 1. ISBN: 9781449344764.

[107] Qiang Guan, Xunchao Hu, Terence Grove, Bo Fang, Hailong Jiang, Heng Yin, and Nathan DeBadeleben. "Chaser: An Enhanced Fault Injection Tool for Tracing Soft Errors in MPI Applications." In: *Proceedings of the 50th International Conference on Dependable Systems and Networks*. DSN '20. 2020, pages 355–363. DOI: 10.1109/DSN48063.2020.00051.

[108] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. "SBFT: A Scalable and Decentralized Trust Infrastructure." In: *Proceedings of the 49th International Conference on Dependable Systems and Networks*. DSN '19. 2019, pages 568–580. DOI: 10.1109/DSN.2019.00063.

[109] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. "RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing." In: *Proceedings of the 37th International Conference on Data Engineering*. ICDE '21. 2021, pages 1392–1403. DOI: 10.1109/ICDE51399.2021.00124.

[110] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. "ResilientDB: Global Scale Resilient Blockchain Fabric." In: *Proceedings of the VLDB Endowment* 13.6 (2020), pages 868–883. DOI: 10.14778/3380750.3380757.

[111] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. "PeerReview: Practical Accountability for Distributed Systems." In: *Proceedings of the 21st Symposium on Operating Systems Principles*. SOSP '07. 2007, pages 175–188. DOI: 10.1145/1294261.1294279.

[112] Andreas Haeberlen and Petr Kuznetsov. "The Fault Detection Problem." In: *Proceedings of the 13th International Conference on Principles of Distributed Systems*. OPODIS '09. 2009, pages 99–114. DOI: 10.1007/978-3-642-10877-8_10.

[113] Robert B. Hagmann. "A crash recovery scheme for a memory-resident database system." In: *IEEE Transactions on Computers* C-35.9 (1986), pages 839–843. DOI: 10.1109/TC.1986.1676845.

[114] Osama Haq, Mamoon Raja, and Fahad R. Dogar. "Measuring and Improving the Reliability of Wide-Area Cloud Paths." In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. 2017, pages 253–262. DOI: 10.1145/3038912.3052560.

[115] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. "IronFleet: Proving Practical Distributed Systems Correct." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. 2015, pages 1–17. DOI: 10.1145/2815400.2815428.

[116] Jelle Hellings and Mohammad Sadoghi. "Coordination-Free Byzantine Replication with Minimal Communication Costs." In: *Proceedings of the 23rd International Conference on Database Theory*. ICDT '20. 2020, 17:1–17:20. DOI: 10.4230/LIPIcs.ICDT.2020.17.

[117] Stephen Hemminger. *Network Emulation with NetEm*. linux.conf.au. 2005.

[118] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pages 463–492. DOI: 10.1145/78969.78972.

[119] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. "Cores That Don't Count." In: *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*. HotOS '21. 2021, pages 9–16. DOI: 10.1145/3458336.3465297.

[120] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. "B4 and after: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN." In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. 2018, pages 74–87. DOI: 10.1145/3230543.3230545.

[121] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. "Flexible Paxos: Quorum Intersection Revisited." In: *Proceedings of the 20th International Conference on Principles of Distributed Systems*. OPODIS '16. 2017, 25:1–25:14. DOI: 10.4230/LIPIcs.OPODIS.2016.25.

[122] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *Proceedings of the 2010 USENIX Annual Technical Conference*. USENIX ATC '10. 2010, pages 145–158.

[123] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7." In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pages 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[124] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. "B4: Experience with a Globally-Deployed Software Defined WAN." In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), pages 3–14. DOI: 10.1145/2534169.2486019.

[125] Leander Jehl. "Quorum Selection for Byzantine Fault Tolerance." In: *Proceedings of the 39th International Conference on Distributed Computing Systems*. ICDCS '19. 2019, pages 2168–2177. DOI: 10.1109/ICDCS.2019.00213.

[126] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. "CheapBFT: Resource-efficient Byzantine Fault Tolerance." In: *Proceedings of the 7th European Conference on Computer Systems*. EuroSys '12. 2012, pages 295–308. DOI: 10.1145/2168836.2168866.

[127] Rüdiger Kapitza, Thomas Zeman, Franz J. Hauck, and Hans P. Reiser. "Parallel State Transfer in Object Replication Systems." In: *Proceedings of the 7th International Conference on Distributed Applications and Interoperable Systems*. DAIS '07. 2007, pages 167–180. DOI: 10.1007/978-3-540-72883-2_13.

[128] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. "All about Eve: Execute-Verify Replication for Multi-Core Servers." In: *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*. OSDI '12. 2012, pages 237–250. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kapritsos.

[129] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. "Byzantine Fault Detectors for Solving Consensus." In: *The Computer Journal* 46.1 (Jan. 2003), pages 16–35. DOI: 10.1093/comjnl/46.1.16.

[130] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. "Zyzzyva: Speculative Byzantine Fault Tolerance." In: *ACM Transactions on Computer Systems* 27.4 (2009). DOI: 10.1145/1658357.1658358.

[131] Ramakrishna Kotla and Mike Dahlin. "High Throughput Byzantine Fault Tolerance." In: *Proceedings of the 34th International Conference on Dependable Systems and Networks*. DSN '04. 2004, pages 575–584. DOI: 10.1109/DSN.2004.1311928.

[132] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. "MDCC: Multi-Data Center Consistency." In: *Proceedings of the 8th European Conference on Computer Systems*. EuroSys '13. 2013, pages 113–126. DOI: 10.1145/2465351.2465363.

[133] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. RFC Editor, Feb. 1997. DOI: 10.17487/RFC2104.

[134] Nico Kruber, Maik Lange, and Florian Schintke. "Approximate Hash-Based Set Reconciliation for Distributed Replica Repair." In: *Proceedings of the 34th International Symposium on Reliable Distributed Systems*. SRDS '15. 2015, pages 166–175. DOI: 10.1109/SRDS.2015.30.

[135] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. "Revisiting Optimal Resilience of Fast Byzantine Consensus." In: *Proceedings of the 40th Symposium on Principles of Distributed Computing*. PODC '21. 2021, pages 343–353. DOI: 10.1145/3465084.3467924.

[136] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. "To Relay or Not to Relay for Inter-Cloud Transfers?" In: *Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing*. HotCloud '18. 2018. URL: https://www.usenix.org/conference/hotcloud18/presentation/lai.

[137] Leslie Lamport. *Generalized Consensus and Paxos*. Technical report MSR-TR-2005-33. Microsoft Research, 2005. URL: https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/.

[138] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem." In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pages 382–401. DOI: 10.1145/357172.357176.

[139] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. RFC 7748. RFC Editor, Jan. 2016. DOI: 10.17487/RFC7748.

[140] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. "SoK: Automated Software Diversity." In: *Proceedings of the 35th Symposium on Security and Privacy*. SP '14. 2014, pages 276–291. DOI: 10.1109/SP.2014.25.

[141] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. "Troxy: Transparent Access to Byzantine Fault-Tolerant Systems." In: *Proceedings of the 48th International Conference on Dependable Systems and Networks*. DSN '18. 2018, pages 59–70. DOI: 10.1109/DSN.2018.00019.

[142] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. "SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance." In: *Proceedings of the 12th European Dependable Computing Conference*. EDCC '16. 2016, pages 77–88. DOI: 10.1109/EDCC.2016.36.

[143] Jun-Lin Lin and Margaret H. Dunham. "Segmented fuzzy checkpointing for main memory databases." In: *Proceedings of the 11th Symposium on Applied Computing*. SAC '96. 1996, pages 158–165. DOI: 10.1145/331119.331168.

[144] Linux Network Developers. *netem*. 2021. URL: https://wiki.linuxfoundation.org/networking/netem (visited on 06/03/2023).

[145] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quema, and Marko Vukolic. "XFT: Practical Fault Tolerance beyond Crashes." In: *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*. OSDI '16. 2016, pages 485–500. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/liu.

[146] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. "The SMART Way to Migrate Replicated Stateful Services." In: *Proceedings of the 1st European Conference on Computer Systems*. EuroSys '06. 2006, pages 103–115. DOI: 10.1145/1217935.1217946.

[147] Dahlia Malkhi, Kartik Nayak, and Ling Ren. "Flexible Byzantine Fault Tolerance." In: *Proceedings of the 25th Conference on Computer and Communications Security*. CCS '19. 2019, pages 1041–1053. DOI: 10.1145/3319535.3354225.

[148] Dahlia Malkhi and Michael Reiter. "Byzantine quorum systems." In: *Distributed Computing* 11.4 (1998), pages 203–213. DOI: 10.1007/s004460050050.

[149] Jean-Philippe Martin and Lorenzo Alvisi. "A Framework for Dynamic Byzantine Storage." In: *Proceedings of the 34th International Conference on Dependable Systems and Networks*. DSN '04. 2004, pages 325–334. DOI: 10.1109/DSN.2004.1311902.

[150] Jean-Philippe Martin and Lorenzo Alvisi. "Fast Byzantine Consensus." In: *IEEE Transactions on Dependable and Secure Computing* 3.3 (2006), pages 202–215. DOI: 10.1109/DSN.2005.48.

[151] Odorico Machado Mendizabal, Fernando Luís Dotti, and Fernando Pedone. "High Performance Recovery for Parallel State Machine Replication." In: *Proceedings of the 37th International Conference on Distributed Computing Systems*. ICDCS '17. 2017, pages 34–44. DOI: 10.1109/ICDCS.2017.193.

[152] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. USA: CRC Press, Inc., 1996. Chapter 11, pages 425–488. ISBN: 0849385237.

[153] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. USA: CRC Press, Inc., 1996. Chapter 9, pages 323–324. ISBN: 0849385237.

[154] Ralph Charles Merkle. "Secrecy, Authentication, and Public Key Systems." AAI8001972. PhD thesis. Stanford University, Stanford, California, USA, 1979. Chapter V.4, pages 40–45.

[155] Microsoft Azure. *Availability sets overview - Azure Virtual Machines*. 2022. URL: https://learn.microsoft.com/en-us/azure/virtual-machines/availability-set-overview (visited on 06/03/2023).

[156] Microsoft Azure. *Time sync for Windows VMs in Azure - Azure Virtual Machines*. 2022. URL: https://learn.microsoft.com/en-us/azure/virtual-machines/windows/time-sync (visited on 06/03/2023).

[157] Microsoft Azure. *What are Azure regions and availability zones?* 2022. URL: https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview#availability-zones (visited on 06/03/2023).

[158] Microsoft Azure. *Azure Virtual Network FAQ*. 2023. URL: https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-faq#do-vnets-support-multicast-or-broadcast (visited on 06/03/2023).

[159] Microsoft Azure. *Choose the Right Azure Region for You*. 2023. URL: https://azure.microsoft.com/en-us/explore/global-infrastructure/geographies/#overview (visited on 06/03/2023).

[160] Microsoft Azure. *Service Level Agreements (SLA) for Online Services*. 2023. URL: https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services (visited on 06/03/2023).

[161] Zarko Milosevic, Martin Biely, and André Schiper. "Bounded Delay in Byzantine-Tolerant State Machine Replication." In: *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*. SRDS '13. 2013, pages 61–70. DOI: 10.1109/SRDS.2013.15.

[162] Iulian Moraru, David G. Andersen, and Michael Kaminsky. "There Is More Consensus in Egalitarian Parliaments." In: *Proceedings of the 24th Symposium on Operating Systems Principles*. SOSP '13. 2013, pages 358–372. DOI: 10.1145/2517349.2517350.

[163] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. RFC Editor, Nov. 2016. DOI: 10.17487/RFC8017.

[164] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Technical report Federal Information Processing Standards Publication 180-4. Washington, D.C.: U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[165] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. "DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications." In: *Proceedings of the 44th International Conference on Management of Data*. SIGMOD '18. 2018, pages 1221–1236. DOI: 10.1145/3183713.3196928.

[166] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. "Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs." In: *Proceedings of the 6th European Conference on Computer Systems*. EuroSys '11. 2011, pages 343–356. DOI: 10.1145/1966445.1966477.

[167] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX ATC '14. 2014, pages 305–320.

[168] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. "Facebook's Tectonic Filesystem: Efficiency from Exascale." In: *Proceedings of the 19th Conference on File and Storage Technologies*. FAST '21. 2021, pages 217–231. URL: https://www.usenix.org/conference/fast21/presentation/pan.

[169] Fernando Pedone and André Schiper. "Generic Broadcast." In: *Proceedings of the 13th International Symposium on Distributed Computing*. DISC '99. 1999, pages 94–106. DOI: 10.1007/3-540-48169-9_7.

[170] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. "Making Fast Consensus Generally Faster." In: *Proceedings of the 46th International Conference on Dependable Systems and Networks*. DSN '16. 2016, pages 156–167. DOI: 10.1109/DSN.2016.23.

[171] Miguel Pires, Srivatsan Ravi, and Rodrigo Rodrigues. "Generalized Paxos Made Byzantine (and less complex)." In: *Algorithms* 11.9 (2018). DOI: 10.3390/a11090141.

[172] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. "Characterizing the Impact of Intermittent Hardware Faults on Programs." In: *IEEE Transactions on Reliability* 64.1 (2015), pages 297–310. DOI: 10.1109/TR.2014.2363152.

[173] Robbert van Renesse, Chi Ho, and Nicolas Schiper. "Byzantine Chain Replication." In: *Proceedings of the 16th International Conference on Principles of Distributed Systems*. OPODIS '12. 2012, pages 345–359. DOI: 10.1007/978-3-642-35476-2_24.

[174] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018. DOI: 10.17487/RFC8446.

[175] Tuanir França Rezende and Pierre Sutra. "Leaderless State-Machine Replication: Specification, Properties, Limits." In: *Proceedings of the 34th International Symposium on Distributed Computing*. DISC '20. 2020, 24:1–24:17. DOI: 10.4230/LIPIcs.DISC.2020.24.

[176] Mathieu Rosemain and Raphael Satter. *Millions of websites offline after fire at French cloud services firm*. 2021. URL: https://www.reuters.com/article/us-france-ovh-fire-idUSKBN2B20NU (visited on 06/03/2023).

[177] Alírio Santos de Sá, Allan Edgard Silva Freitas, and Raimundo José de Araújo Macêdo. "Adaptive Request Batching for Byzantine Replication." In: *SIGOPS Operating System Review* 47.1 (2013), pages 35–42. DOI: 10.1145/2433140.2433149.

[178] Kenneth Salem and Hector Garcia-Molina. "Checkpointing memory-resident databases." In: *Proceedings of the 5th International Conference on Data Engineering*. ICDE '89. 1989, pages 452–462. DOI: 10.1109/ICDE.1989.47249.

[179] Fred B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial." In: *ACM Computing Surveys* 22.4 (1990), pages 299–319. DOI: 10.1145/98163.98167.

[180] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. "Design and Analysis of a Logless Dynamic Reconfiguration Protocol." In: *Proceedings of the 25th International Conference on Principles of Distributed Systems*. OPODIS '21. 2021, 26:1–26:16. DOI: 10.4230/LIPIcs.OPODIS.2021.26.

[181] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-Free Replicated Data Types." In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. SSS '11. 2011, pages 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[182] Victor Shoup. "Practical Threshold Signatures." In: *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT '00. 2000, pages 207–220. DOI: 10.1007/3-540-45539-6_15.

[183] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. "Dynamic Reconfiguration of Primary/Backup Clusters." In: *Proceedings of the 2012 USENIX Annual Technical Conference*. USENIX ATC '12. 2012, pages 1–13.

[184] Nibesh Shrestha and Mohan Kumar. *Revisiting ezBFT: A Decentralized Byzantine Fault Tolerant Protocol with Speculation*. arXiv. 2019. DOI: 10.48550/ARXIV.1909.03990.

[185] Douglas Simões Silva, Rafal Graczyk, Jérémie Decouchant, Marcus Völp, and Paulo Esteves-Verissimo. "Threat Adaptive Byzantine Fault Tolerant State-Machine Replication." In: *Proceedings of the 40th International Symposium on Reliable Distributed Systems*. SRDS '21. 2021, pages 78–87. DOI: 10.1109/SRDS53918.2021.00017.

[186] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. "The Internet at the Speed of Light." In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. HotNets '14. 2014, pages 1–7. DOI: 10.1145/2670518.2673876.

[187] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls." In: *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*. OSDI '10. 2010, pages 33–46.

[188] João Sousa and Alysson Bessani. "From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation." In: *Proceedings of the 9th European Dependable Computing Conference*. EDCC '12. 2012, pages 37–48. DOI: 10.1109/EDCC.2012.32.

[189] João Sousa and Alysson Bessani. "Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines." In: *Proceedings of the 34th International Symposium on Reliable Distributed Systems*. SRDS '15. 2015, pages 146–155. DOI: 10.1109/SRDS.2015.40.

[190] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. "Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery." In: *IEEE Transactions on Parallel and Distributed Systems* 21.4 (2010), pages 452–465. DOI: 10.1109/TPDS.2009.83.

[191] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. "Bullshark: DAG BFT Protocols Made Practical." In: *Proceedings of the 28th Conference on Computer and Communications Security*. CCS '22. 2022, pages 2705–2718. DOI: 10.1145/3548606.3559361.

[192] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. *Mir-BFT: High-Throughput Robust BFT for Decentralized Networks*. arXiv. 2019. DOI: 10.48550/ARXIV.1906.05552.

[193] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. "State Machine Replication Scalability Made Simple." In: *Proceedings of the 17th European Conference on Computer Systems*. EuroSys '22. 2022, pages 17–33. DOI: 10.1145/3492321.3519579.

[194] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. "Basil: Breaking up BFT with ACID (Transactions)." In: *Proceedings of the 28th Symposium on Operating Systems Principles*. SOSP '21. 2021, pages 1–17. DOI: 10.1145/3477132.3483552.

[195] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms." In: *SIAM Journal on Computing* 1.2 (1972), pages 146–160. DOI: 10.1137/0201010.

[196] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. "Consistency-based Service Level Agreements for Cloud Storage." In: *Proceedings of the 24th Symposium on Operating Systems Principles*. SOSP '13. 2013, pages 309–324. DOI: 10.1145/2517349.2522731.

[197] *The State of Online Retail Performance*. Technical report. Akamai, 2017. URL: https://s3.amazonaws.com/sofist-marketing/State+of+Online+Retail+Performance+Spring+2017+-+Akamai+and+SOASTA+2017.pdf.

[198] Sarah Tollman, Seo Jin Park, and John Ousterhout. "EPaxos Revisited." In: *Proceedings of the 18th Symposium on Networked Systems Design and Implementation*. NSDI '21. 2021, pages 613–632. URL: https://www.usenix.org/conference/nsdi21/presentation/tollman.

[199] Gene Tsudik. "Message Authentication with One-Way Hash Functions." In: *ACM SIGCOMM Computer Communication Review* 22.5 (1992), pages 29–38. DOI: 10.1145/141809.141812.

[200] International Telecommunication Union. *Global Connectivity Report*. Technical report. International Telecommunication Union, Place des Nations, CH-1211 Geneva, Switzerland, 2022. URL: https://www.itu.int/dms_pub/itu-d/opb/ind/d-ind-global.01-2022-pdf-e.pdf.

[201] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary." In: *Proceedings of the 28th International Symposium on Reliable Distributed Systems*. SRDS '09. 2009, pages 135–144. DOI: 10.1109/SRDS.2009.36.

[202] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. "EBAWA: Efficient Byzantine Agreement for Wide-Area Networks." In: *Proceedings of the 12th Symposium on High-Assurance Systems Engineering*. HASE '10. 2010, pages 10–19. DOI: 10.1109/HASE.2010.19.

[203] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. "Efficient Byzantine Fault-Tolerance." In: *IEEE Transactions on Computers* 62.1 (2013), pages 16–30. DOI: 10.1109/TC.2011.221.

[204] Paolo Viotti and Marko Vukolić. "Consistency in Non-Transactional Distributed Storage Systems." In: *ACM Computing Surveys* 49.1 (June 2016). DOI: 10.1145/2926965.

[205] Rory Ward and Betsy Beyer. "BeyondCorp: A New Approach to Enterprise Security." In: *;login:* 39.6 (2014), pages 6–11.

[206] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-conditioned, Scalable Internet Services." In: *Proceedings of the 18th Symposium on Operating Systems Principles*. SOSP '01. 2001, pages 230–243. DOI: 10.1145/502034.502057.

[207] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. "Tolerating Latency in Replicated State Machines through Client Speculation." In: *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*. NSDI '09. 2009, pages 245–260.

[208] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. 2015, pages 357–368. DOI: 10.1145/2737924.2737958.

[209] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. "ZZ and the art of practical BFT execution." In: *Proceedings of the 6th European Conference on Computer Systems*. EuroSys '11. 2011, pages 123–138. DOI: 10.1145/1966445.1966457.

[210] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. "Separating Agreement from Execution for Byzantine Fault Tolerant Services." In: *Proceedings of the 19th Symposium on Operating Systems Principles*. SOSP '03. 2003, pages 253–267. DOI: 10.1145/945445.945470.

[211] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness." In: *Proceedings of the 38th Symposium on Principles of Distributed Computing*. PODC '19. 2019, pages 347–356. DOI: 10.1145/3293611.3331591.

[212] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. "SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines." In: *Proceedings of the 9th Symposium on Cloud Computing*. SoCC '18. 2018, pages 68–81. DOI: 10.1145/3267809.3267837.

[213] Siyuan Zhou and Shuai Mu. "Fault-Tolerant Replication with Pull-Based Consensus in MongoDB." In: *Proceedings of the 18th Symposium on Networked Systems Design and Implementation*. NSDI '21. 2021, pages 687–703. URL: https://www.usenix.org/conference/nsdi21/presentation/zhou.