Resource-efficient Fault and Intrusion Tolerance

—

Ressourceneffiziente Fehler- und Einbruchstoleranz

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

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Tobias Distler

aus Nürnberg

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

Tag der mündlichen Prüfung:	24. Juni 2014
Vorsitzende des Promotionsorgans:	Prof. Dr.-Ing. habil. Marion Merklein
Gutachter:	Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
	Prof. Dr.-Ing. Felix Freiling
	Prof. Dr.-Ing. Rüdiger Kapitza

Abstract

More and more network-based services are considered essential by their operators: either because their unavailability might directly lead to economic losses, as with e-commerce applications or online auction services, for example, or because their well-functioning is crucial for the well-functioning of other services, which is, for example, the case for distributed file systems or coordination services. Byzantine fault-tolerant replication allows systems to be built that are able to ensure the availability and reliability of network-based services, even if a subset of replicas fail arbitrarily. As a consequence, such systems not only tolerate fault scenarios in which replicas crash, but also cases in which replicas have been taken over by an adversary as the result of a successful intrusion.

Despite the fact that several major outages of network-based services in the past have been caused by non-crash failures, industry is still reluctant to broadly exploit the available research results on Byzantine fault tolerance. One of the main reasons for the decision to retain crash-tolerant systems is the high resource demand associated with Byzantine fault-tolerant systems: Besides the need to execute more costly protocols, the more complex fault model also requires Byzantine fault-tolerant systems to comprise more replicas than their crash-tolerant counterparts.

In this thesis, we propose and evaluate different protocols and techniques to increase the resource efficiency of Byzantine fault-tolerant systems. The key insights that serve as a basis for all of these approaches are that during normal-case operation it is sufficient for a system to detect (or at least suspect) faults, while during fault handling a system must be able to actually tolerate faults, and that the former usually requires less resources than the latter. Utilizing these insights, we investigate different ways to improve resource efficiency by implementing a clear separation between normal-case operation and fault handling based on two modes of operation: During normal-case operation, a system reduces its resource usage to a level at which it is only able to ensure progress as long as all replicas behave according to specification. In contrast, in case of suspected or detected faults, the system switches to an operation mode in which it may use additional resources in order to tolerate faults.

An important outcome of this thesis is that passive replication can be an effective building block for the implementation of a resource-efficient operation mode for normal-case operation in Byzantine fault-tolerant systems. Furthermore, experimental results show that improving the resource efficiency of a system can also lead to an increase in performance.

Kurzzusammenfassung

Netzwerkbasierte Dienste werden von ihren Betreibern zunehmend als unentbehrlich angesehen: entweder weil ihr Ausfall direkt zu ökonomischen Verlusten führen könnte, wie etwa bei elektronischen Handelssystemen oder internetgestützten Auktionsdiensten, oder weil die Verfügbarkeit anderer Dienste von ihnen abhängt, wie es beispielsweise bei Netzwerkdateisystemen oder Koordinierungsdiensten der Fall sein kann. Das Prinzip der byzantinisch fehlertoleranten Replikation erlaubt es Systemen die Verfügbarkeit und Zuverlässigkeit von netzwerkbasierten Diensten sogar dann zu gewährleisten, wenn ein Teil der Replikate willkürliches Fehlverhalten zeigt. Solche Systeme können daher nicht nur Replikatausfälle tolerieren, sondern auch Szenarien, in denen Replikate als Folge von Einbrüchen von einem Angreifer übernommen wurden.

Trotz der Tatsache, dass willkürliches Fehlverhalten von Systemkomponenten in der Vergangenheit zu mehreren schwerwiegenden Ausfällen von netzwerkbasierten Diensten geführt hat, werden existierende Forschungsergebnisse aus dem Bereich der byzantinischen Fehlertoleranz weiterhin kaum für den Produktiveinsatz genutzt. Einer der Hauptgründe dafür sich wie bisher auf ausfalltolerante Systeme zu beschränken ist der mit byzantinisch fehlertoleranten Systemen verbundene hohe Ressourcenbedarf: Neben der Notwendigkeit des Einsatzes aufwendigerer Protokolle macht es das komplexere Fehlermodell außerdem erforderlich, dass byzantinisch fehlertolerante Systeme mehr Replikate als vergleichbare ausfalltolerante Systeme bereitstellen.

Diese Dissertation schlägt mehrere Protokolle und Techniken zur Steigerung der Ressourceneffizienz von byzantinisch fehlertoleranten Systemen vor und evaluiert sie. Allen hier präsentierten Ansätzen liegen dabei die zentralen Erkenntnisse zugrunde, dass es für den Normalbetrieb ausreicht Fehler erkennen (oder sie zumindest vermuten) zu können, wogegen sie im Zuge einer Fehlerbehandlung tatsächlich toleriert werden müssen, und dass ersteres weniger Ressourcen benötigt als letzteres. Aufbauend darauf, werden verschiedene Vorgehensweisen untersucht, wie sich die Ressourceneffizienz eines Systems durch eine klare Trennung des Normalbetriebs von der Fehlerbehandlung und die damit verbundene Einführung zweier Betriebsmodi steigern lässt: Im Normalfall befindet sich das System in einem Modus, in dem es seinen Ressourcenverbrauch so weit senkt, dass Fortschritt nur noch gewährleistet ist, solange sich alle Replikate korrekt verhalten. Im Unterschied dazu stehen im Fehlerbehandlungsmodus zusätzliche Ressourcen zur Verfügung, um Fehler tolerieren zu können; in ihn wird umgeschaltet, sobald das Auftreten von Fehlern entweder vermutet oder erkannt wird.

Ein zentrales Resultat dieser Arbeit ist die Erkenntnis, dass passive Replikation ein effektives Mittel zur Implementierung eines ressourceneffizienten Normalbetriebsmodus darstellt. Darüber hinaus belegen Evaluationsergebnisse, dass eine verbesserte Ressourceneffizienz auch zu einer gesteigerten Leistungsfähigkeit eines Systems führen kann.

Contents

List of Figures

1
Introduction

1.1 Motivation

In an ongoing process, conventional computing infrastructure is increasingly replaced by services that are accessed over a network, typically the Internet. With their importance growing, ensuring the availability and reliability of network-based services becomes more and more crucial. The key task in this regard is to make the systems that provide such services resilient against faults of system components, including both hardware (e.g., disk, memory, processor) and software (e.g., operating system, middleware, service application). A common approach to address this problem is to tolerate crashes of system components by applying replication [138]: Instead of comprising a single physical server, a replicated system is distributed across multiple machines, each running an instance of the service application (i.e., a replica). This way, the overall system is able to still provide its service even if some of the replicas crash.

Unfortunately, crashing is not the only way system components fail in practice [60]: As the result of a firmware error, for example, disks may report writes as successfully completed without actually having stored the corresponding data [128]. As another example, software bugs in database systems may lead to clients receiving incorrect replies and/or the state of the database being corrupted [74, 148]. Note that, in the past, non-crash faults have been identified as the culprit responsible for several major outages of network-based services [64, 121, 123].

Byzantine fault tolerance [105] is an approach that allows systems to be built that are not only resilient against crashes of replicas but can also deal with non-crash failures of system components: In contrast to a crash-tolerant system, a Byzantine fault-tolerant system assumes that a subset of its components may be subject to arbitrary, so-called *Byzantine faults*; that is, in addition to crashes, the Byzantine fault model considers components to fail in an arbitrary way, which for example includes scenarios where a system component sends confirmations for operations it has not performed, as well as cases in which a replica provides clients with incorrect replies.

Making no limiting assumptions on how and why system components become faulty, the Byzantine fault model also covers situations where components fail as the result of an adversary deliberately trying to disrupt the service. In particular, being resilient against Byzantine faults allows systems to tolerate intrusions [31, 130, 136, 149, 150, 151]; that is, a Byzantine fault-tolerant system is able to continue to provide its service although some replicas have been compromised by an adversary and possibly try to prevent other replicas from making progress. For this reason, we use the terms *Byzantine fault tolerance* and *fault and intrusion tolerance* interchangeably in this thesis.

Despite offering many benefits, Byzantine fault-tolerant systems are currently not broadly used to improve the availability and reliability of network-based services [43], even though numerous research efforts in recent years have contributed to bringing their performance [34, 42, 51, 79, 96, 97, 139, 153], scalability [1, 9, 90, 161], implementation costs [35, 77], and resilience [8, 44, 141, 144] to practical levels. As one of the main reasons for why industry is reluctant to exploit the available research, the *high resource demand* of Byzantine fault tolerance has been identified [80, 101]: In the general case, a Byzantine fault-tolerant system must comprise a minimum of $3f + 1$ replicas to tol-

erate up to f faulty replicas [125]; that is, making a non-fault-tolerant system resilient against one Byzantine fault, for example, requires three additional servers. Furthermore, in order to ensure availability and reliability in the presence of faulty replicas, prior to processing a client request, fault and intrusion-tolerant systems have to execute more complex agreement protocols than systems that are only resilient to crashes.

1.2 Purpose of this Thesis

In this thesis, we join the efforts of others [41, 49, 80, 130, 152, 159, 161] by investigating techniques to increase the resource efficiency of fault and intrusion-tolerant systems. Note that there are two general ways to make a system more resource efficient, which are both addressed in this thesis: First, to provide the same performance with less resources and, second, to improve performance while keeping resource usage constant. Research so far has mainly focused on the former.

Saving Resources In order to be able to increase resource efficiency, it is essential to know which types of resources are used by a Byzantine fault-tolerant system and for what purposes: Due to the distributed nature of the system, clients and replicas communicate via messages exchanged over a *network*. Furthermore, the authentication, transmission, and processing of messages consumes *CPU*. Finally, each replica must manage a separate copy of the service state on *disk*. As a consequence, there are different basic approaches which, individually or combined, result in resources being saved: For example, minimizing the total number of messages that are exchanged to agree on a client request [41, 49, 80, 152] reduces network and CPU usage. Moreover, decreasing the total number of replicas in the system [41, 49, 80, 130, 152] and/or the number of replicas on which a request is executed on [159, 161] saves CPU and disk space. In addition, due to the power consumption of today's servers not being proportional to their workloads [17], operating less replicas in general also leads to a reduced energy consumption.

Occurrences of Faults and Intrusions We do not advocate the Byzantine fault-tolerant protocols presented in this thesis to be the only tools applied to improve the availability and reliability of network-based services. Instead, we argue that different approaches should be combined for this purpose: For example, checksums can be effectively used to detect and repair corrupted messages [110]. Apart from that, relying on formal methods [92] allows to decrease the number of software bugs and vulnerabilities that could be exploited for intrusions; techniques to minimize the attack surface of a system [99] can further reduce the range of possibilities that are available to an adversary.
Applying these and other approaches may not completely prevent Byzantine faults from surfacing, as the examples in Section 1.1 have shown; however, thanks to them, in our work, we can utilize the justified assumption that in the domain of network-based services Byzantine faults at the replica level are rare [3, 50, 77, 78, 79, 159]. Note that we expect our protocols and systems to be relied on in use-case scenarios for which the same applies to intrusions, based on the rationale that, if a system is subject to frequent attacks, resource efficiency is likely to be the least concern of the system's operator.

Suggested Approach In this thesis, we propose to improve the resource efficiency of fault and intrusion-tolerant systems by implementing a clear separation between normal-case operation and fault-handling procedures [106]. The fundamental insights behind this approach are that during normal-case operation it is sufficient for a system to be able to detect (or at least suspect) faults, while during fault handling a system must be able to tolerate faults, and that the former usually requires less resources than the latter. In particular, we present different ways to realize a normal-case operation mode, in which a system only uses enough resources to ensure progress in the absence of faults; while in this mode, a system saves resources (see above) by running a protocol that sends less messages and executes a client request on less replicas than state-of-the-art Byzantine fault-tolerant protocols. Being in normal-case operation mode, a system is not able to actually tolerate faults. Therefore, in case of suspected or detected faults, a system switches to fault-handling mode in which additional resources are used to provide resilience against faults and intrusions. Assuming such fault scenarios to be rare, we expect a system to spend only a small amount of time on fault-handling procedures, allowing it to save resources during the rest of the time.

1.3 Scope of this Thesis

The approaches presented in this thesis are targeted at improving the resource efficiency of network-based services that are considered essential by their operators and therefore have strong availability and reliability requirements: On the one hand, this includes services that are (business) critical themselves as their unavailability leads to economic losses (e.g., e-commerce applications or online auction services). On the other hand, this pertains to services whose well-functioning is crucial for the well-functioning of other services (e.g., distributed file systems [75, 145] or coordination services [30, 84]).

General Focus Note that, in this thesis, we address neither real-time systems nor embedded systems. Although Byzantine faults pose a serious problem in these domains [60], fault-tolerant protocols for such systems can usually exploit strong synchrony assumptions (e.g., upper bounds on network and processing delays) to reduce the minimum number of replicas required. Furthermore, if all replicas of a service are executed on the same node, there is no need for a full-fledged Byzantine fault-tolerant agreement protocol in order to provide resilience against Byzantine faults [147]. In contrast, we consider systems whose replicas are distributed across multiple physical servers that are (if not stated otherwise) connected via an unreliable network for which no upper bounds on communication delays are known. These weaker assumptions come at the cost of an increased overhead for Byzantine fault-tolerant replication [34]; furthermore, they prevent the systems presented in this thesis from guaranteeing worst-case execution times, for example, for fault-handling procedures.

Intrusion Tolerance Laying the focus on the resource efficiency of Byzantine fault tolerance, we do not address a number of security-related aspects that are essential for fault and intrusion-tolerant systems in practice: For example, we do not discuss mechanisms to perform access control or techniques to improve protection against denial-of-service

attacks [34, 44]; in addition, we also do not consider the problem of ensuring confidentiality [22, 161]. In general, our focus does not lie on the question how to prevent intrusions from happening, but on how to tolerate the presence of compromised replicas after an attack has been successful. For this purpose, we not only consider non-malicious but also malicious Byzantine faults.
Resilience against Byzantine faults is based on the assumption that faults in different replicas are not correlated [23, 40, 148]. When taking malicious faults into account, special attention has to be given to this issue: For example, if all replicas run the same operating system, an adversary may exploit the same vulnerability to take over all of them [73], thereby violating the assumption of replicas failing independently. Note that, in this thesis, we do not directly contribute to the effort of reducing the probability for such a scenario. However, taking up the results of a recent study [72], which has concluded that introducing diversity at the operating-system level significantly reduces the probability of correlated faults, we discuss the use of heterogeneous replica implementations in the context of our SPARE system presented in Chapter 4.

1.4 Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2 introduces the basic architecture and system model of agreement-based fault and intrusion-tolerant systems, which serve as a basis for our work. In addition, the chapter discusses related work in order to identify techniques that can be used, modified, or extended to build resource-efficient Byzantine fault-tolerant systems.

Chapter 3 analyzes resource consumption in existing fault and intrusion-tolerant systems and identifies the challenges in making such systems resource efficient. In addition, the chapter gives an overview of the general approach proposed in this thesis to address theses challenges: Enabling a system to increase resource efficiency in the absence of faults by making use of different modes of operation.

Chapter 4 presents SPARE, a fault and intrusion-tolerant system that partially relies on passive replication to save resources during normal-case operation. In addition, SPARE utilizes virtualization to efficiently provide resilience against Byzantine faults for long-running services.

Chapter 5 details REBFT, an approach to introduce a resource-saving operation mode for the normal case into existing fault and intrusion-tolerant systems. In this chapter, the approach is applied to two state-of-the-art Byzantine fault-tolerant systems.

Chapter 6 presents ODRC, a Byzantine fault-tolerant system that minimizes the overhead for request execution in the context of state-machine replication. In contrast to SPARE and REBFT, which aim at reducing the resource footprint of a fault and intrusion-tolerant system, ODRC is geared towards improving resource efficiency by increasing performance based on the resources available.

Chapter 7 summarizes the findings and contributions of this theses. Furthermore, the chapter outlines further ideas on how the techniques and mechanisms presented in previous chapters could be used to address problems beyond resource efficiency.

1.5 Related Publications

The ideas and results presented in this dissertation have partly also been published as:

[23] Alysson Neves Bessani, Hans P. Reiser, Paulo Sousa, Ilir Gashi, Vladimir Stankovic, Tobias Distler, Rüdiger Kapitza, Alessandro Daidone, and Rafael Obelheiro. “FOREVER: Fault/intrusiOn REmoVal through Evolution & Recovery.” In: *Proceedings of the Middleware 2008 Conference Companion (Middleware’08, Poster Session)*. 2008, pages 99–101.

[56] 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.

[57] 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.

[58] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. “Efficient State Transfer for Hypervisor-Based Proactive Recovery.” In: *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS’08)*. 2008, pages 7–12.

[59] 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.

[86] 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.

I was a contributor to the project presented in [23]. In [56], I was the leading author and major contributor to the design and implementation of the system. In [57], [58], and [59], I was one of two main authors and the major contributor to the design and implementation of the systems. In [86], I was one of three main authors and the major contributor to the design of the protocol. The REBFT approach presented in Chapter 5 is a revised and generalized version of the protocol published in [86]; I am the major contributor to the design and implementation of REBFT.

2

Background, System Model, and State of the Art

In this chapter, we present the standard system model and basic architecture of the fault and intrusion-tolerant systems relevant to this thesis and explain how system components interact in order to provide resilience against Byzantine faults. Furthermore, we introduce common assumptions made in the context of this particular field of research that have an influence on our own system and protocol designs proposed in subsequent chapters. Following this, in the main section of this chapter, we give an overview of state-of-the-art techniques and approaches that are of value in the effort to build resource-efficient fault and intrusion-tolerant systems.

2.1 System Model and Basic Architecture

This section provides background on the system model and basic architecture of state-of-the-art fault and intrusion-tolerant systems, which also serve as a basis for our own system proposals presented in Chapters 4 through 6; in case our system proposals deviate from the system model below, this is discussed in the respective chapter. Note that, in this thesis, we lay a focus on agreement-based Byzantine fault-tolerant system architectures [34, 35, 96, 97, 152, 153, 154, 161] and consequently omit an in-depth discussion of quorum-based system architectures [1, 6, 51, 113, 114] in this section.

2.1.1 System Model

In the context of this thesis, we consider distributed systems in which the server side runs an application providing a service that is used by a set of *clients*. In order to be resilient against faults, the server side comprises a group of *replicas*, usually running on different physical machines, which are also referred to as *servers*. Clients and replicas communicate via messages: To invoke an operation on the service application, the client issues a request to the server side; after the request has been processed on a replica, the replica returns the result of the operation in a reply back to the client.

2.1.1.1 Communication

System components located on different physical machines communicate by exchanging messages over a network using point-to-point connections. Reliable transmission of messages is not necessarily guaranteed: The network may fail to deliver messages, corrupt and/or delay them, or deliver them out of order. Messages are assumed to be protected by checksums in order for receivers to be able to detect corrupted messages; corrupted messages are not processed. If a sender learns or suspects that its message has not reached the receiver, the sender retransmits the message; we assume that receivers are able to identify and suppress duplicate messages. In practice, a reliable communication protocol (e.g., TCP [127]) might be used to tolerate temporary network failures and/or out-of-order message delivery.

Clients and replicas of a Byzantine fault-tolerant system authenticate messages they send over the network. The actual method of authentication (e.g., symmetric cryptography or public-key signatures) is not part of the system model and may therefore differ between systems. Receivers verify the origin of a message before processing it and drop a message if verification fails, for example, because the actual sender of a message is not the component specified in the message. System components who have to interact must be able to authenticate each other's messages. However, we make no assumptions on how this is achieved in practice (e.g., by establishing a shared secret).

2.1.1.2 Fault and Threat Model

At all times, each component of a Byzantine fault-tolerant system is assumed to be in one of two states: A *non-faulty* component operates according to specification, for example, providing correct results. In contrast, a *(Byzantine) faulty* component may violate its specification in an arbitrary way: Besides a failure by crashing, this also includes the

possibility of a component violating its specification while it continues to work. Faults in different system components are assumed to not be correlated. Note that, in practice, fault independence of system components can be improved by introducing diversity (e.g., by applying N-version programming [13, 23, 40, 148]) and consequently relying on heterogeneous implementations.
Besides addressing non-malicious hardware and software faults, the Byzantine fault model also covers intrusions, creating the need for a threat model: The standard threat model [34, 35, 96, 97, 152, 153, 154, 161] takes into account that an adversary launching an attack might have full control over the system components compromised and might therefore send arbitrary (correctly authenticated) messages on behalf of such components. A powerful adversary might even coordinate different compromised clients and replicas as part of its attack. While potentially omniscient, an adversary is assumed to have limitations: First, an adversary is computationally bound and consequently not able to break cryptographic techniques, which means that an adversary, for example, cannot send correctly authenticated messages on behalf of a non-faulty system component. Second, an adversary is not able to prevent two non-faulty system components from communicating with each other.
As further discussed in Section 2.1.1.3, the standard system model of Byzantine fault-tolerant systems assumes an upper bound on the number of faulty replicas. However, an arbitrary number of clients may be faulty in addition. Note that, in this thesis, we do not discuss in detail how to deal with faulty clients; one way to address this problem is to add clients that continuously show faulty behavior to a blacklist enforced by a (separate) access control system in order to prevent them from further using the service [34].

2.1.1.3 Safety and Liveness

The common way to assess that a Byzantine fault-tolerant system is correct with respect to its system model is by showing that the system fulfills two properties: safety and liveness [102]. Note that, in this thesis, we use the same approach as Castro and Liskov [34] according to which, in order to be safe and live, a fault and intrusion-tolerant system must behave "as a centralized implementation that executes operations atomically one at a time" (safety) and that "clients eventually receive replies to their requests" (liveness).

Safety Providing safety requires a Byzantine fault-tolerant system to return correct results to client requests as long as at most f replicas are faulty; f is a constant that has to be defined in advance. If the number of actual faults exceeds this upper bound, a system's fault-tolerance guarantees are void [109]. Due to the fact that, as discussed in Section 2.2.1, the value chosen for f affects the minimum number of replicas necessary in a system, only small values (i.e., $f = 1$ or $f = 2$) are considered practical.
In order to behave like a centralized implementation, the states of non-faulty replicas in a Byzantine fault-tolerant system need to be kept consistent. As further discussed in Section 2.1.2, for a system relying on active replication, this means that a client request that is executed on one non-faulty replica must also be processed on other non-faulty replicas, and that non-faulty replicas must handle client requests in the same order.

Liveness Consensus in a fully asynchronous system is impossible if one or more nodes may crash [67]. In consequence, a Byzantine fault-tolerant system can only ensure liveness in a partially synchronous environment [61], in which there are upper bounds on communication delays and system-component response times; however, both bounds do not have to be known to system components. With regard to network communication, we assume that a message, which is exchanged between two non-faulty components over an unreliable network (see Section 2.1.1.1), to eventually arrive at the receiver, possibly after several retransmissions [161]. Note that, in order to fulfill the liveness property under the system model used in this thesis, a fault and intrusion-tolerant system does not have to guarantee worst-case execution times for client requests. However, even in the presence of up to f replica faults, a system must eventually make progress.

2.1.1.4 Voting Certificates

One important tool to ensure safety in a fault and intrusion-tolerant system is the use of *(voting) certificates*. In the context of this thesis, a voting certificate is a proof for the correctness of a particular value; for example, a reply certificate proves that, under the fault assumptions made in a system (see Sections 2.1.1.2 and 2.1.1.3), the result returned by the replicated service is considered to be correct. Voting certificates are generated by voter components and are based on correctly-authenticated messages provided by different replicas: For example, to verify the correctness of a result, a voter collects and compares the replies of different replicas for the same request. In order to tolerate up to f faults, a reply certificate becomes *stable* as soon as the result voter has obtained $f + 1$ matching replies. At this point, the corresponding result is successfully verified as at least one of the replies received must have been provided by a non-faulty replica.

While some voters in a Byzantine fault-tolerant system prove the correctness of results, other voters are responsible for verifying messages exchanged between replicas (e.g., checkpoints, see Section 2.1.2.3). Depending on their specific tasks, voters may either be integrated with clients, which is the standard way of using result voters [34], or with replicas. In both cases, a fault in a voter can propagate to its associated system component, for example, leading to a replica accepting a faulty checkpoint, thereby itself becoming faulty. Note that, in this thesis, we do not address the problem of faulty voters infecting clients but assume that a client can trust the decision of its result voter; if this is not guaranteed in practice, additional measures are necessary to ensure the correctness of the voter's decision process [20, 47, 147].

In order for a voting certificate to become stable, a voter does not necessarily have to obtain $f + 1$ full versions of the message to verify: As an alternative, a voter is, for example, also able to successfully complete the verification process based on a single full message and f matching hashes provided by different replicas [34]; in this context, a hash is a checksum computed over the payload of the full message a replica would have sent. Making use of hashes allows a Byzantine fault-tolerant system to reduce the amount of data that has to be sent over the network.

2.1.1.5 Replica States

In order to be resilient against faults, a fault and intrusion-tolerant system must keep the states of non-faulty replicas consistent. However, the need for consistency does not require replicas to be identical, which is why, similar to Reiser et al. [130], we distinguish between two parts of a replica's state:

- The *service state* comprises the effects of all modifications made by the application that have or might have an observable influence on the output of the replicated service. A typical example for such information is a chunk of data that has been stored as the result of a client request and may later be retrieved. If the service state of a replica is inconsistent or lost, it has to be recreated based on the service states of non-faulty replicas in the system.
- The *system state*, in contrast, comprises all information of a replica's state that is not included in the service state: First, this covers all information that has no observable effect (neither direct nor indirect) on the output of the replicated service. Second, this refers to information that has an observable effect on service output but, if inconsistent or lost, could be restored by a reboot of the server. For all use cases addressed in this thesis, the operating system and middleware of a replica belong to the system state as the state of their internal data structures does not affect service output and/or can be recreated without the involvement of other replicas.

While the entire service state has to be kept consistent across all non-faulty replicas, this is only true for the parts of the system state that have an observable effect on service output. Due to the fact that, as discussed above, the latter is straight-forward as it requires no additional replica interaction, in this thesis, we focus on the former. For simplicity, if not stated otherwise, the term *state* therefore refers to the service state of a replica.
Note that making the decision which part of the state a particular piece of information belongs to requires application-specific knowledge and has to be made during development/integration time. Based on the information available during this process (e.g., the application source code), we expect a unique mapping to be feasible. However, in case of doubt, it is always safe to treat a piece of information as part of the service state, possibly at the expense of keeping it consistent unnecessarily.

2.1.2 Basic Architecture

Existing Byzantine fault-tolerant systems [34, 35, 41, 96, 97, 152, 153, 154, 161] rely on active replication [138] to provide resilience against faults. Independent of their particular realizations, the basic architecture of these systems can be divided into the following three parts (see Figure 2.1): A set of clients issue requests to the service. On the server side, a set of *agreement replicas* constitute the *agreement stage* [161], which is responsible for establishing a total order on the requests of all clients to ensure consistency. Based on the sequence determined by the agreement stage, a set of *execution replicas* constituting the *execution stage* then process the requests. Note that such an architectural division is

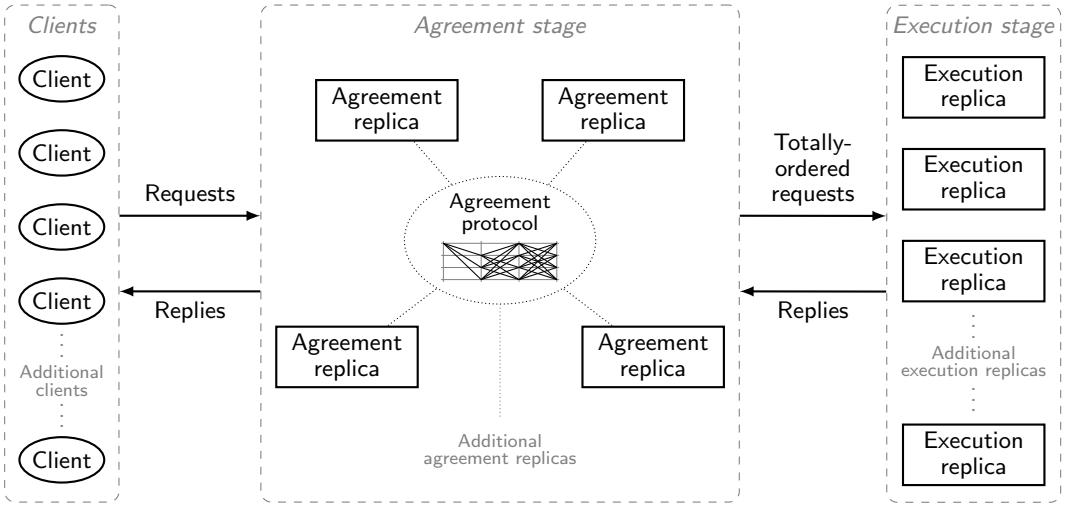


Figure 2.1: Basic architecture of an agreement-based Byzantine fault-tolerant system: Requests issued by a set of clients are processed on multiple replicas at the execution stage, respecting a global total order defined by the agreement stage of the system.

conceptual and not intended to give any hints on how to implement an actual system: While clients are usually hosted on separate physical machines, an agreement replica may, for example, be run in the same process as an execution replica.

2.1.2.1 Client

In this thesis, we use the term *client* to refer to the component at the client side that directly interacts with the server side (e.g., a library issuing requests on behalf of a user application). If not stated otherwise, we assume that the client is replication-aware, a property the user application itself does not necessarily have to provide. Each client can be identified by its system-wide unique client id that is part of every message a client sends. In addition, a client assigns a request number, which is based on a client-local counter, to every new operation invoked on the service. Note that the combination of client id and request number, the request id, can be used by system components to uniquely identify a request [118]. Amongst other things, the request id allows clients to detect and consequently drop duplicate replies; furthermore, the use of request ids enables execution replicas to ensure that each request is processed only once [65].
For simplicity, we follow the example of other authors [34, 161] and assume that a non-faulty client invokes a single operation at a time; that is, after sending a request to the agreement stage, a client blocks until it has obtained a stable result that has been verified by a voter (see Section 2.1.1.4) based on the replies of different execution replicas. Having returned the result to the user application, a client is ready to issue a subsequent request. Note that the assumption of having at most one outstanding request per client

simplifies protocol description, for example, in the context of reply caching at the agreement stage (see Section 2.1.2.2). There is no fundamental reason that would prevent one from implementing an asynchronous interaction between a client and the service in the systems presented in this thesis.
In case a client is not able to obtain a verified reply to its request within a certain (application-specific) period of time after having invoked the operation, the client sends a notification to the agreement stage indicating the request in question (see Section 2.1.2.2). Such a notification signals the server side that a fault might have occurred and consequently allows the agreement stage to initiate appropriate fault-handling procedures, which will eventually lead to a client obtaining a verified result to its request.

2.1.2.2 Agreement Stage

The agreement stage of a Byzantine fault-tolerant system comprises multiple agreement replicas, usually distributed across different physical machines, that cooperate to reliably establish a system-wide unique total order on all client requests issued to the system. The result of the agreement stage (i.e., the sequence of client requests) is then presented to the execution stage.

Agreement Protocol To ensure safety in the presence of faults, the agreement stage runs an agreement protocol which guarantees that the sequence of ordered requests is identical across all non-faulty replicas as long as the number of faulty replicas does not exceed the upper bound the agreement stage has been dimensioned for (see Section 2.1.1.3). As further discussed in Section 2.2, state-of-the-art Byzantine fault-tolerant agreement protocols differ in the amount of resources they use (e.g., the number of agreement replicas required), the performance overhead they impose (e.g., the number of protocol phases necessary), as well as the assumptions they make (e.g., use of trusted components).
Agreement protocols are organized in successive instances. The purpose of each instance is to reliably assign an *agreement sequence number* to a request. Having totally-ordered a request, the agreement stage of a Byzantine fault-tolerant system puts out an agreement certificate that includes a mapping between the request and its agreement sequence number as well as a proof that the mapping has become stable. Such a proof usually consists of a set of messages provided by different replicas indicating their commitment to the particular mapping.

Batching A key technique to improve the throughput of (Byzantine) fault-tolerant agreement protocols is *batching* [34, 69, 137]: Instead of ordering a single client request per protocol instance, applying this technique, multiple requests are combined into a batch, and a single agreement sequence number is assigned to the entire batch. The position of a specific request in the overall sequence of totally-ordered requests can then be determined by the agreement sequence number of the corresponding batch in conjunction with the individual position of the request in its batch.

Reply Cache Besides ordering client requests, the agreement stage is responsible for managing a reply cache that contains the reply to the latest executed request of each client [34, 161]. Note that the limitation to only one reply per client is a consequence of the assumption that a client has at most a single outstanding request at a time (see Section 2.1.2.1). Relying on the reply cache, non-faulty agreement replicas can ensure that they are able to provide replies that have not reached the client due to network problems. When the agreement stage receives a request whose reply is cached, replicas react by retransmitting the reply to the corresponding client.

Fault Handling As agreement replicas might not always have sufficient local knowledge to monitor the progress of request ordering, the agreement stage of a Byzantine fault-tolerant system exposes an interface that allows other system components (e.g., clients) to report on suspected faulty or malicious behavior; a typical use-case example for this interface is a *stalled-verification notification* put out by a voter indicating that it lacks enough matching values to successfully complete the verification of a message within a certain period of time. Having been informed about a possible problem, an agreement replica usually not triggers fault-handling procedures right away. Instead, due to the fact that a report about suspected faulty behavior may have been issued by a faulty system component, an agreement replica first waits for further evidence, which, for example, may be provided by additional system components.

2.1.2.3 Execution Stage

The execution stage of a Byzantine fault-tolerant system uses the totally-ordered sequence of requests provided by the agreement stage as input. Its main task is to invoke the corresponding operations on the actual service application and to pass the resulting replies on to the agreement stage, which then forwards the replies to clients.

Request Execution For fault tolerance, the execution stage consists of multiple execution replicas, each running a separate instance of the service application. To ensure consistency, execution replicas implement the same deterministic state machine [138]; that is, for the same sequence of inputs (i.e., client requests), every non-faulty execution replica produces the same sequence of outputs (i.e., replies). Thereby, execution replicas are in an identical state between processing the same two requests.
Note that the need to ensure a consistent state across non-faulty replicas is usually a limiting factor at the execution stage of (Byzantine) fault-tolerant systems: In particular, the straight-forward approach of processing requests sequentially in order to guarantee deterministic behavior may result in a significant performance penalty [90, 97]. To address this problem, a number of techniques have been proposed by different authors [18, 19, 21, 87, 97, 126] that allow execution replicas to safely process requests in parallel while preserving consistency. Kotla et al. [97], for example, proposed to execute requests in parallel if they access different parts of the service state.

Checkpoints Besides processing client requests, an execution replica periodically (e.g., based on the number of operations invoked) instructs its local service-application instance to create a snapshot of the application's current state. Such *checkpoints* serve different purposes: First, they can be used to initialize new execution replicas and/or help slow replicas that have fallen behind to catch up. Second, they are used by the agreement stage to perform garbage collection of internal agreement-protocol state [34]. However, similar to a result (see Section 2.1.1.4), a checkpoint in a Byzantine fault-tolerant system may only be used after its respective checkpoint certificate has become stable based on matching confirmations provided by different execution replicas in the system.

2.2 State of the Art

The goal of this thesis is the design and implementation of resource-efficient fault and intrusion-tolerant systems. In this section, we give an overview of existing techniques and approaches that may be used, modified, and/or extended for our purposes. Whenever suitable, we do not only present the state of the art in Byzantine fault-tolerant systems but also discuss related work in the context of crash tolerance.

2.2.1 Lower Bounds on the Number of Replicas

In general, resilience against up to f Byzantine faults requires a system to comprise a minimum of $3f + 1$ agreement replicas [125] as well as at least $2f + 1$ execution replicas [161]. To minimize the resource overhead associated with Byzantine fault tolerance, in recent years, researchers have proposed different approaches that allow a reduction of the number of replicas at both stages by relying on trusted system parts.

Use of Trusted System Components at the Agreement Stage In traditional Byzantine fault-tolerant systems that do not make use of trusted components [34, 35, 96, 153, 161] a faulty replica is able to successfully perform *equivocation*; that is, a faulty replica may use the same message id to send messages with different contents to different replicas. As multiple authors [41, 154] have shown, if faulty replicas can be prevented from performing equivocation without being detected, the minimum number of replicas at the agreement stage of a fault and intrusion-tolerant system can be reduced to $2f+1$. For this purpose, Chun et al. [41], for example, presented an attested append-only memory that provides a trusted log for recording protocol messages. With every agreement replica being able to independently access the log in order to validate messages, non-faulty replicas have sufficient information to detect if other replicas try to perform equivocation.
Levin et al. [108] have shown that in order to prevent equivocation, it is sufficient for a trusted component to provide a monotonically increasing counter. More precise, when being invoked for a message, such a trusted component is required to securely assign a unique counter value to the message and to guarantee that it will never assign the same counter value to another message. Besides signing messages this way, non-faulty

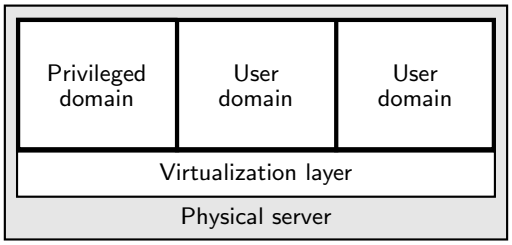


Figure 2.2: Use of virtualization on a server: A virtualization layer manages the access of virtual machines (▭, also referred to as "domains") to the resources of the physical host and enforces isolation between different domains. While service applications typically run in non-privileged user domains, a dedicated privileged domain is responsible for handling system operations.

agreement replicas must only handle messages received from other replicas in the order determined by their respective counter values; in case of gaps in the sequence of counter values, a replica is forced to block until the message with the next counter value to handle becomes available. With all non-faulty agreement replicas behaving accordingly, a faulty replica is prevented from successfully performing equivocation as sending messages with different contents would result in different counter values being assigned, which in turn could be detected, for example, due to gaps in the counter-value sequence. In MinBFT [154], Veronese et al. utilize a trusted service that builds on this principle to not only reduce the number of replicas but also the number of protocol phases required in the agreement stage of a Byzantine fault-tolerant system.

Trusted Agreement Stages While the approaches presented above only rely on trusted components at the agreement stage of a fault and intrusion-tolerant system, other authors have proposed systems in which the entire agreement stage is trusted [48, 49, 130]. In particular, such systems assume a hybrid fault model in which the agreement stage only fails by crashing while other parts of the system may be subject to Byzantine faults. As a consequence of this fault model, a minimum of $2f + 1$ replicas are sufficient at the agreement stage to tolerate up to f faults.

Virtualization-based Approaches In recent years, virtualization technology [2, 16, 93] has become an important tool for operating data centers. One of the main reasons for this development is the possibility to reduce costs by increasing server utilization [11, 52]: Instead of hosting each service application on a dedicated server, virtualization allows applications (possibly by different data-center users) to be run in isolated user domains (i.e., separate virtual machines) on the same physical machine, as shown in Figure 2.2. Isolation between user domains is enforced by the virtualization layer comprising a virtual-machine monitor. In addition, each physical server hosts a privileged domain responsible for managing system operations including, for example, the starting and stopping of user domains.

Different authors have proposed to utilize virtualization as a building block for fault and intrusion-tolerant systems: Reiser et al. [130], for example, presented VM-FIT, a system in which the privileged domain of each server hosts an agreement replica while the corresponding execution replica runs in a user domain on the same physical machine. VM-FIT relies on a hybrid fault model (see above), which assumes that execution replicas may be subject to Byzantine faults, whereas all remaining system parts, including agreement replicas, are trusted and only fail by crashing. Note that in order to be safe, such an approach requires the virtualization layer to reliably enforce the isolation of user domains; otherwise, Byzantine faults in execution replicas could propagate to other parts of the system. Due to the hybrid fault model, VM-FIT requires $2f + 1$ replicas at both the agreement stage as well as the execution stage.
In contrast to VM-FIT, the ZZ system proposed by Wood et al. [159] comprises $3f + 1$ agreement replicas that run a standard Byzantine fault-tolerant agreement protocol to order requests. Although not taking advantage of virtualization at the agreement stage, at the execution stage ZZ exploits the fact that different execution replicas can be safely co-located on the same server by running them in separate user domains. As VM-FIT, ZZ assumes the virtualization layer to be trusted.

Conclusion Reducing the number of replicas at the agreement stage of a fault and intrusion-tolerant system below $3f + 1$ requires the assumption that certain system components only fail by crashing; in such case, it is possible for a Byzantine fault-tolerant agreement stage to comprise $2f + 1$ replicas. Note that this is only a lower bound for agreement stages that actually have to tolerate Byzantine faults: For systems with hybrid fault models, which rely on crash-tolerant agreement protocols, smaller agreement stages are possible if additional assumptions on fault detection are made. In Chapter 4, we utilize this insight in SPARE, a virtualization-based fault and intrusion-tolerant system that comprises $f + 1$ replicas at the agreement stage; by co-locating two execution replicas (each running in its own user domain) on the same physical machine, SPARE only requires a total of $f + 1$ servers. Apart from that, in Chapter 5, we present a resource-efficient Byzantine fault-tolerant agreement protocol that makes use of a trusted service to prevent equivocation based on securely assigning unique counter values to messages, as proposed by Levin et al. [108].

2.2.2 Handling the Normal Case and the Worst Case Separately

In his seminal paper on the design and implementation of computer systems, Lampson pointed out that in general it is an effective strategy to handle the normal case separately from the worst case due to both having different requirements: "the normal case must be fast; the worst case must make some progress." [106] Below, we discuss a number of examples in which this methodology has been used in both crash-tolerant systems as well as Byzantine fault-tolerant systems. The fundamental insight behind these approaches is that in the normal case it is sufficient for a system to be able to detect (or at least suspect) faults, while in the worst case the system must be able to tolerate faults.

Crash-tolerant Systems Pâris [124] exploited this insight in the context of a quorum-based file system in which replicas assume one of two possible roles: While some replicas ("copies") contain both the contents of files as well as a set of file-version numbers, other replicas ("witnesses") only manage file-version numbers, without storing file contents. Based on their particular roles, replicas in the system handle a state-modifying request differently; that is, copies fully execute the write request, which includes an update of the file contents, while witnesses only increment their local file-version counter. This way, in the absence of faults, witnesses are able to spare the overhead of request processing. In case of faults, witnesses may be upgraded to copies in order to provide the system's fault-tolerance guarantees. Liskov et al. [111] introduced a related mechanism into the Harp file system that limits the active participation of witnesses to periods of node failures or network partitions. In the context of Cheap Paxos [104], Lamport et al. generalized the idea of using auxiliary nodes responsible for handling crashes of full-fledged replicas as a means to save resources in a crash-tolerant system.

Byzantine Fault-tolerant Systems Witnesses have also been utilized in the context of fault and intrusion-tolerant systems: Van Renesse et al. [80, 131] proposed a system in which, in the normal case, witnesses take part in the agreement stage but not in the execution stage. In order to guarantee liveness, their system relies on an external service for reconfiguring the roles of replicas.
Apart from that, several authors (e.g., [6, 15, 51, 100, 163]) have presented Byzantine fault-tolerant systems that are able to improve performance by exploiting benign conditions like good quality of network connections and/or a low number of faulty replicas. Zieliński [163], for example, developed a Byzantine fault-tolerant protocol that allows the agreement process of a request to be completed early if a sufficiently large number of replicas have correctly participated in the ordering of the request.
Guerraoui et al. [77] generalized the idea of building a Byzantine fault-tolerant agreement protocol that is able to dynamically adapt to changing conditions: Instead of executing a single, monolithic protocol, they proposed to compose a set of individual sub protocols, each designed to serve a particular purpose, for example, to provide high throughput in the absence of faults. Relying on such a composite agreement protocol, a system can achieve high performance if it manages to execute the sub protocol best suited for the conditions present at a certain time.
In their virtualization-based ZZ system (see Section 2.2.1), Wood et al. [159] applied an approach aimed at reducing the number of redundant request executions during normal-case operation: In the absence of faults, only $f + 1$ execution replicas, each running in a separate user domain, process client requests. If faults occur, up to f additional execution replicas are set up on demand to assist in fault handling. However, after having been activated, execution replicas in ZZ may not be prepared to step in right away due to first needing to fetch the current application state from other replicas in the execution stage.

Conclusions Handling the normal case and the worst case separately allows the implementation of computer systems to account for the particular requirements of each case. As further discussed in Chapter 3, in this thesis, we apply this concept by propos-

ing different modes of operation in order to improve the resource efficiency of fault and intrusion-tolerant systems: a normal-case operation mode, in which the system saves resources while ensuring progress in the absence of faults, as well as a fault-handling mode, in which additional resources are used to actually tolerate faults.
The fact that there are lower bounds on the number of replicas at both the agreement stage as well as the execution stage (see Section 2.2.1), does not mean that, at all times, all replicas actually have to actively participate in system operations: As shown by ZZ for the execution stage, a subset of replicas being active is sufficient for the system to make progress during normal-case operation. In Chapter 5, we extend this idea to the agreement stage of a Byzantine fault-tolerant system and present two resource-saving protocols for the normal case in which some replicas do not participate in the ordering of client requests. In case of suspected or detect faults, a protocol switch is triggered which ensures that those replicas can safely rejoin the agreement stage.

2.2.3 Recovery of System Components

As discussed in Section 2.1.1.3, fault-tolerant systems in practice are not able to tolerate an infinite number of faults. Instead, the degree of fault-tolerance guarantees provided by a system depends on the number of replicas in use. For long-running services, such an upper bound poses a problem as the number of faulty replicas is likely to eventually grow beyond any practical maximum number of faults to tolerate. In the following, we discuss approaches to address this issue by allowing system components to recover from suspected and/or detected faults [82, 122]. Applying such approaches, long-running systems do not have to be dimensioned with regard to an absolute fault threshold but instead with regard to the number of faults to tolerate within a certain period of time.

The Need for Proactive Recovery Limiting recovery efforts to system components that are provably faulty is not enough as precise detection of Byzantine faults usually not possible: For example, in case of malicious attacks, adversaries often try to hide evidence pointing to a successful intrusion in order to remain undetected. As a consequence, relying exclusively on mechanisms to reactively recover from detected faults is not always an option. To address this problem, several authors have argued for a proactive recovery of system components, for example, in the context of intrusion-tolerant firewall systems [83], grid servers [142], and quorum-based online certification authorities [162].

Proactive Recovery in Agreement-based Byzantine Fault-tolerant Systems For PBFT [34], Castro et al. presented a proactive-recovery mechanism that periodically rejuvenates replicas even if they do not show any signs of being faulty. During this procedure, a trusted recovery monitor triggers a reboot of the physical server with correct code and restarts the replica. Following this, the recovering replica changes the keys it uses to authenticate messages and fetches missing, corrupt, or outdated service-state parts from other replicas. Note that, as a replica temporarily stops to participate in system operations during reboot, the recovery procedure of a non-faulty replica in PBFT might lead to a temporary service disruption due to not enough non-faulty replicas being available.

While in PBFT only a subset of replicas recover at the same time, VM-FIT [130] (see Section 2.2.1) performs proactive recovery for all replicas simultaneously. As a result, the service temporarily becomes unavailable for a short period of time. However, leveraging virtualization allows VM-FIT to minimize the service disruption during a proactive-recovery procedure: Instead of rebooting the entire physical server, as done in PBFT, VM-FIT creates the next generation of replicas in separate virtual machines while the current generation of replicas is still active. In consequence, the period of service unavailability caused by a proactive-recovery procedure is limited to the short time it takes to switch between replica generations.
Sousa et al. [143] presented a system that comprises mechanisms for both proactive and reactive recovery. Furthermore, they proposed to reduce the impact of a recovery procedure on the availability of a service by increasing the total number of replicas in the system. This way, a majority of replicas is able to continuously process client requests without being affected by a small subset of other replicas recovering.

Conclusion Support for proactive recovery requires additional resources: If it is acceptable for a service to become unavailable for the time it takes to recover a replica, as in PBFT, this overhead can be small as only the information the replica has missed while rebooting needs to be provided by other replicas. In contrast, if availability is of major concern, additional replicas are required. In order to support long-running services at the cost of no extra physical servers, we use the approach proposed by VM-FIT as a basis for the proactive-recovery mechanism of the SPARE system presented in Chapter 4.

2.2.4 Minimizing Replication Overhead

Executing a fault-tolerant protocol for each client request and processing each request on all replicas in a system (see Section 2.1.2.3) is expensive, not only in terms of resources but also with regard to performance. In the following, we present a number of general techniques and approaches that are used in fault-tolerant (and other) computer systems to address this problem.

Active vs. Passive Replication As shown in Section 2.1.2, the basic architecture of existing fault and intrusion-tolerant systems relies on active replication [138]; that is, each client request is redundantly processed on multiple replicas that all implement the same deterministic state machine. In consequence, if some of the replicas fail, the system is still able to make progress based on the replicas remaining. In contrast, systems based on passive replication [4, 29] provide fault tolerance in a different way: Here, a single replica, the primary, processes all client requests while the other replicas, the backups, stand by in order to take over if the primary fails. However, as backup replicas do not execute client requests, and therefore miss changes to the service state, their states become outdated. One common way to address this problem is to bring backup replicas up to speed based on state updates provided by the primary, which if applied locally allow a backup replica to perform the state modifications caused by a client request [29].

Note that passive replication so far has only been applied in the context of crash tolerance due to the following problem with regard to Byzantine faults: If a primary is assumed to be able to fail in an arbitrary way, the primary may provide clients with faulty replies and/or send faulty state updates to backup replicas. As a result, faults in the primary may spread to other system components, thereby violating the assumption that faults in different system components are not correlated (see Section 2.1.1.2).

State Partitioning Another important technique for the design and implementation of systems, in which not all nodes have to execute all client requests, is state partitioning: Applying this technique, each node in the system is only responsible for the requests accessing a certain part of the service state. By assigning different state parts to different nodes, the individual load per node is reduced. State partitioning has proven to be an effective means to improve scalability, for example, in the context of large-scale file systems and distributed data storage [76, 112, 132] providing resilience against crashes. Furthermore, state partitioning has been applied to tolerate Byzantine faults: Farsite [3] and OceanStore [133], for example, are large-scale file systems that rely on different replica groups, each executing a separate Byzantine fault-tolerant agreement protocol.

Operation-dependent Request Handling Making a system resilient against faults and/or intrusions does not necessarily mean that in all cases request handling must involve full-fledged fault-tolerance mechanisms. A common optimization in this context is to distinguish between read-only requests that do not affect the state of the replicated service and state-modifying requests that lead to parts of the service state being added, deleted, and/or altered [34, 35, 42, 96, 97]. Castro et al. [34], for example, included a mechanism in PBFT that allows read-only requests to take an optimistic short path, circumventing the regular Byzantine fault-tolerant agreement protocol. Sen et al. [139] proposed to optimize for read-centric workloads by introducing a trusted cache that is able to serve read-only requests based on the reply generated by a single replica, provided that the value accessed has not changed since it has been read the last time.

Conclusions Compared with active replication, passive replication allows a system to reduce the number of times a client request has to be executed at the expense of bringing backup replicas up to speed by state updates. As applying a state update in general requires less resources than processing the corresponding client request, the resource footprints of passively-replicated systems are usually smaller than the resource footprints of their actively-replicated counterparts. Note that this is especially true for workloads with a high amount of read-only requests as such requests do not cause state modifications that need to be performed at backup replicas.
Due to the benefits with regard to resource consumption, in this thesis, we investigate passive replication as a central building block in the context of Byzantine fault-tolerant systems. In the systems presented in Chapters 4 and 5, we address the faulty-primary problem (see above) by assigning the tasks of a primary to a group of replicas instead of a single replica. This way, the correctness of replies and state updates can be verified by comparing the values provided by different replicas (see Section 2.1.1.4).

Apart from that, in the ODRC system in Chapter 6, we make use of state partitioning to increase performance. However, in contrast to Byzantine fault-tolerant systems discussed above, ODRC does not comprise different replica groups but instead applies partitioning at the granularity of state objects within a single replica group.

2.3 Chapter Summary

In this chapter, we introduced the standard system model and basic system architecture of state-of-the-art fault and intrusion-tolerant systems, which serve as a basis for our own work presented in the remainder of this thesis. Furthermore, we identified existing approaches and ideas that can be applied, modified, and/or extended in order to improve the resource efficiency of Byzantine fault-tolerant systems. In particular, this includes the use of passive replication, which so far has only been investigated in the context of crash tolerance, as well as the concept of making a clear separation between normal-case operation and fault-handling procedures. In the next chapter, we analyze the consumption of resources in Byzantine fault-tolerant systems in more detail and present our approach to improve resource efficiency in such systems.

3
Problem Analysis and Suggested Approach

In this chapter, we analyze the seminal PBFT [34] protocol in order to illustrate how resources are consumed in state-of-the-art fault and intrusion-tolerant systems. Based on the insights gained from this analysis, we then formulate a number of challenges in building resource-efficient Byzantine fault-tolerant systems. Finally, we give an overview of the general approach with which we aim to address these challenges in the systems presented in the remaining chapters of this thesis.

3.1 Problem Analysis

The goal of this section is to analyze how the resource efficiency of fault and intrusion-tolerant systems can be improved. For this purpose, we first briefly introduce and then analyze the PBFT protocol proposed by Castro and Liskov [34], which has been used in several Byzantine fault-tolerant systems [3, 35, 77, 79, 97, 161] and has also served as basis for a number of other agreement protocols [9, 41, 42, 44, 51, 96, 152, 153, 154].

3.1.1 The PBFT Protocol

PBFT requires a total of $3f + 1$ replicas to tolerate up to f faulty replicas (see Section 2.2.1). In each protocol instance, one of the replicas, the leader, is responsible for proposing a request, which is then accepted by the other replicas, the followers. As discussed in Section 2.1.1.3, in order to ensure safety, it is crucial that non-faulty replicas agree on accepting the same request in the same protocol instance despite up to f faulty replicas also actively participating in the protocol.

Preliminary Remarks This section is intended to provide necessary background for the resource-usage analysis of PBFT in Section 3.1.2, which is why we omit protocol details not relevant in this context and why, in the following, we only focus on a single protocol instance. Furthermore, we do not discuss PBFT's procedure for replacing a faulty leader, during which replicas exchange messages proving the progress of the agreement process in order to overcome potential discrepancies caused by a faulty leader.
Figure 3.1 shows the communication pattern of a PBFT protocol instance as presented in [33]. For the description below, it is only important that each arrow represents a message; the differences between solid and dotted lines become relevant in Section 3.1.2.

Protocol Description In order to agree on a request, replicas execute different protocol phases that are named after the messages sent in them (see Figure 3.1): First, having received a client request, in the PREPREPARE phase, the leader proposes the request to its followers. Next, in the PREPARE phase, the followers multicast the leader's proposal. As a faulty leader may have proposed different requests to different followers, this procedure allows replicas to confirm to each other that they have seen the same proposal.
If a replica has obtained $2f$ matching PREPARE messages (including its own) for the leader's PREPREPARE message, the request is *prepared*; at this point, a replica has proof that at least $2f + 1$ replicas have seen the same request in this protocol instance and that at least $f + 1$ of them are not faulty. This guarantees that a majority of non-faulty replicas in the system will not consider a different request for this protocol instance. However, as this so far is only the local view of a replica, an additional protocol phase, the COMMIT phase, is necessary in order to allow replicas to learn each others local views.
Having obtained $2f + 1$ matching COMMIT messages (including its own) for a request, the request reaches the status *committed*. This means that a replica can be sure that the request has been prepared on a majority of non-faulty replicas, which can prove this in

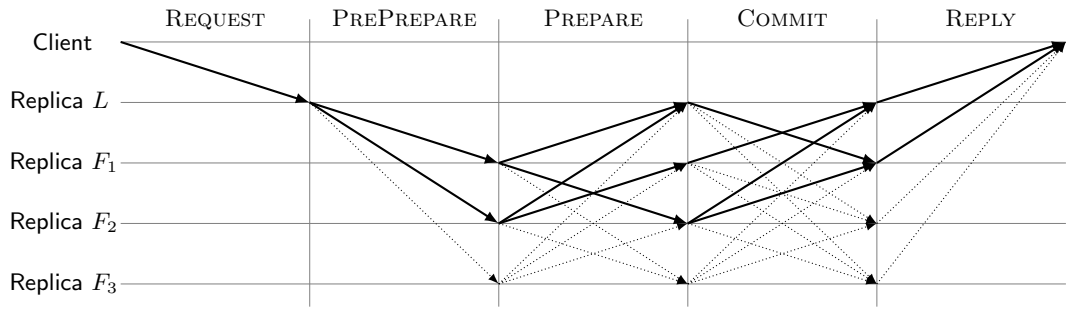


Figure 3.1: Overview of the message flow in a PBFT protocol instance that is able to tolerate one fault: A leader replica L proposes a request which is then accepted by a group of follower replicas F_{1-3}. Of all the messages sent, only a subset of messages (—) contributes to the stability of the result at the client; in contrast, the majority of messages (····) has no effect on the result.

the course of a leader-replacement procedure (i.e., the so-called *view change*). As a result, it is safe for a replica to execute the request and to send the corresponding reply to the client. The client accepts a result as soon as it has received $f + 1$ matching replies from different replicas as at least one of the replies must have been provided by a non-faulty replica and therefore contain the correct result (see Section 2.1.1.4).

3.1.2 Analysis of PBFT

In Section 2.2.2, we have identified a clear separation between normal-case operation and fault-handling procedures as a possible means to improve the resource efficiency of a fault and intrusion-tolerant system. PBFT does not apply such a separation. In the following, we show that PBFT therefore uses more resources during normal-case operation than actually required to make progress under benign conditions.

What is Necessary to Complete a Protocol Instance Under Benign Conditions? As discussed in Section 3.1.1, there are certain requirements that need to be fulfilled in order for a client to obtain a stable result to its request: Starting from the end, to prove the result correct, a client must obtain $f + 1$ matching replies from different replicas. This means that at least $f + 1$ replicas must have executed the request after it has been committed locally. In order to commit locally, a request must have been prepared on at least $2f + 1$ replicas in the system, which in turn means that the leader must have proposed the request for this particular instance to at least $2f$ followers.
Note that the solid arrows in Figure 3.1 show an example run of a PBFT protocol instance, in which all the requirements stated above are fulfilled. However, compared with a regular protocol execution of PBFT (see Section 3.1.1), this protocol run uses less resources: First, only a total of 13 messages (instead of 29 messages) are sent over the network; with messages in PBFT being authenticated, this not only results in a difference in the amount of transmitted packets, but also affects CPU usage, as less cryptographic operations are

executed. Second, as the request is only committed on replicas L and F_1 (instead of all four replicas), only those replicas actually process the request; besides a further decrease of CPU usage, depending on the service application, this could also mean a reduction in the overall number of memory and/or disk usage.

What is Necessary to Complete a Protocol Instance in Case of Faults? Unfortunately, there is no one answer to this question as the answer highly depends on what goes wrong: If, for example, a message is dropped by the network, it is enough to resend the message. On the other hand, if in the protocol run discussed above a replica provides a faulty reply to the client, the request must be executed on an additional replica, which means that it also has to be committed there. As the standard system model for Byzantine fault-tolerant systems (see Section 2.1.1) does not assume that faults can be reliably detected, it is not possible for a system to always react with the fault-handling procedures required for the actual fault scenario. PBFT addresses this problem by always running a protocol that is designed for making progress in the worst-case scenario of f replicas being faulty. In consequence, under benign conditions, more resources (illustrated by the dotted arrows in Figure 3.1) are used than are actually necessary to make progress.

Conclusion Handling the normal case requires less resources than handling the worst case. As a result, there is a potential to reduce the normal-case resource usage of fault and intrusion-tolerant systems and protocols that, like PBFT, do not distinguish between both cases [3, 9, 35, 41, 42, 44, 97, 152, 153, 154, 161].

3.1.3 Challenges in Building Resource-efficient Byzantine Fault-tolerant Systems

Based on the insights gained from the analysis of the PBFT protocol in Section 3.1.2, we can formulate a number of challenges in the context of building resource-efficient Byzantine fault-tolerant systems:

- **Low Resource Usage During Normal-case Operation:** In the absence of faults, a resource-efficient system should reduce its resource footprint to the minimum required for making progress without losing the ability to safely initiate fault-handling procedures.
- **Appropriate Resource Overhead in the Presence of Faults:** Once triggered, fault-handling procedures should not immediately lead to a scenario in which the maximum number of replicas is involved; instead, a resource-efficient system should try to save resources even during fault handling.
- **Benign Resource-saving Mechanisms:** Although an important goal, saving resources must never have the highest priority in a resource-efficient system; that is, it has to be ensured that mechanisms for optimizing resource usage do not endanger the safety and liveness of a system in the presence of faults.

In addition to the resource-related challenges above, a resource-efficient Byzantine fault-tolerant system should also address the common goal of providing good performance.

3.2 Suggested Approach

In order to address the challenges stated in Section 3.1.3, a resource-efficient Byzantine fault-tolerant system needs to be able to dynamically adapt its resource usage to potentially changing conditions. We propose to address this problem by relying on different operation modes: one for normal-case operation and one for performing fault handling.

Normal-case Operation Mode While in normal-case operation mode, a fault and intrusion-tolerant system reduces system operations to the minimum necessary for ensuring safety and liveness under benign conditions. At the execution stage, for example, this includes processing client requests on only $f + 1$ execution replicas as $f + 1$ matching replies are sufficient for a voter to successfully perform result verification in the absence of faults (see Section 2.1.1.4). As a consequence, running in normal-case operation mode allows a system to save resources compared with existing approaches in which each request is processed on $2f+1$ [41, 130, 152, 154] or even $3f+1$ [34, 35, 96, 97, 153, 161] execution replicas. While saving resources in normal-case operation mode, a system is not required to provide means for tolerating faults, intrusions, and/or network problems. However, a system must have a mechanism at its disposal that allows it to safely switch into fault-handling mode in case the presence of one or more of such problems has been suspected or detected.

Fault-handling Mode While in fault-handling mode, making progress and returning to normal-case operation mode is the primary goal of a system; saving resources is only a secondary goal. In consequence, a system, for example, may process a client request for which result verification is incomplete on additional execution replicas, thereby generating enough replies for the voter to successfully prove the corresponding result correct. Note that, depending on the particular use case, the fault-handling mode may be divided into a number of sub modes with different resource-consumption characteristics. In order to complete a pending result verification, a system, for example, may first try to process a request on a single additional execution replica and only rely on further execution replicas in case this measure does not solve the problem.

Switching between Operation Modes Introducing different operation modes into a Byzantine fault-tolerant system creates the need for a safe and efficient mechanism to perform mode switches. For example, if an execution replica crashes while the system is in normal-case operation mode, another execution replica must step in as soon as possible in order to keep the service available. While performing this task is rather straight-forward for stateless applications, stateful execution replicas first need to obtain the current state of the service before they are able to assist in fault handling. To solve this and similar problems related to mode switches in an efficient manner, we propose to use passive replication (see Section 2.2.4) as a central building block in the context of Byzantine fault-tolerant systems: Due to the fact that the state of a passive replica is periodically brought up to speed by applying state updates, preparing a passive replica to assist in fault handling in the course of a mode switch requires only a small overhead.

3.3 Research Questions and Objectives

In Section 3.2, we have introduced our approach to build resource-efficient fault and intrusion-tolerant systems. In this section, we present an overview of the research questions that arise when applying the approach to different parts of a Byzantine fault-tolerant system as well as in different environments. In particular, we are interested in investigating whether the approach is flexible enough to be applied to improve the resource efficiency of a fault and intrusion-tolerant system in different ways:

- **Minimizing a System's Resource Footprint:** Can our approach be used to build a system that provides the same service as another system but is more resource efficient due to achieving the **same performance** by utilizing **less resources**?
- **Increasing System Performance:** Can our approach be used to build a system that provides the same service as another system but is more resource efficient due to achieving a **better performance** by utilizing an **equal amount of resources**?

In the following, we summarize the research questions and objectives guiding the development of the systems presented in Chapters 4 through 6.

Resource-efficient Virtualization-based Replication The observation that virtualization has become one of the central building blocks of today's data centers [11, 52] raises the question whether this technology [2, 16, 93] can also be utilized to minimize the resource footprint of a fault and intrusion-tolerant system. In this context, two differences between virtualized and non-virtualized environments are of particular interest: First, virtualization offers the possibility to reduce the number of servers required in a fault and intrusion-tolerant system by co-locating subsets of execution replicas on the same physical machines (see Section 2.2.1); with each replica running in a separate virtual machine, the virtualization layer ensures isolation between them. Second, in comparison to physical machines, virtual machines can be activated more quickly. Utilizing these properties, in Chapter 4, we investigate how to apply virtual machines to implement a resource-efficient normal-case operation mode (see Section 3.2) in which only a minimum number of execution replicas are active while all other execution replicas are kept in a resource-saving passive mode. Furthermore, we examine the effectiveness of virtual machines as a tool for implementing an efficient operation-mode switching mechanism.

Passive Byzantine Fault-tolerant Replication While in Chapter 4 we focus on the execution stage, in Chapter 5, we investigate how to design a normal-case operation mode that affects an entire system, including the agreement stage. To this end, we present an approach that allows a subset of replicas to remain passive in the absence of faults. In contrast to active replicas, passive replicas neither participate in the agreement of client requests nor execute them. Instead, passive replicas are brought up to speed by verified state updates provided by the active replicas in the system. If faults are suspected or detected, a reconfiguration mechanism ensures a safe transition to a resilient fault-handling protocol during which passive replicas are activated. To show the flexibility of the approach, we present two different instances of it, with and without trusted components.

On-demand Replica Consistency In contrast to Chapters 4 and 5, in which we address the challenge of minimizing the resource footprint of a fault and intrusion-tolerant system, in Chapter 6, we focus on the use of different operation modes to increase system performance during normal-case operation. In this context, we examine ways of improving the resource efficiency of a system without relying on trusted components, thereby taking a fundamentally different approach than previous works on resource-efficient Byzantine fault-tolerant replication (see Section 2.2.1). Furthermore, of particular interest in this chapter is the question whether providing Byzantine fault tolerance inherently results in a performance penalty that causes each fault and intrusion-tolerant service to perform worse than its respective unreplicated non-fault-tolerant equivalent. Going one step further, in Chapter 6, we also consider the question whether it is possible to reinvest the resources saved while being in normal-case operation mode, meaning to utilize them to provide better performance than an equivalent unreplicated service. Note that, unlike the approaches pursued in Chapters 4 and 5, the system design investigated in Chapter 6 uses operation modes at the granularity of state objects, not replicas: In this system, an execution replica is only responsible for processing requests that access a certain subset of state objects; as such a practice leads to parts of an execution replica's state becoming outdated, the system provides a mechanism to ensure the consistency of replicas on demand, for example, in the course of switching to fault-handling mode.

4

Resource-efficient Virtualization-based Replication

Recent years have shown a shift in the way data centers are operated: Instead of dedicating a server to run applications on behalf of a single user, virtualization technology [2, 16, 93] is applied to host services of different users on the same physical machine, resulting in a significant increase of server utilization and a reduction of costs [11, 52]. As a consequence of this development, more and more (business) critical services with strong dependability requirements are run in virtualized environments. Unfortunately, most state-of-the-art systems and approaches for ensuring high availability in such environments [53, 146, 156] so far only provide resilience against crashes, leaving services vulnerable to Byzantine faults.

In this chapter, we address this problem by presenting a replication architecture for critical services that is designed to tolerate arbitrary faults in the most vulnerable parts of such systems: the virtual machines running the service application. As our main contribution in this chapter, we investigate how to take advantage of the special properties of today's data-center environments by combining the benefits of virtual machines with the resource efficiency of passive replication to reduce the overhead for Byzantine fault tolerance. In addition, we show how to provide an efficient virtualization-based proactive-recovery mechanism in order to support dependability for long-running network-based services. Furthermore, we discuss the integration of heterogeneous execution replicas targeted at minimizing the probability of correlated replica failures. Finally, to evaluate our approach, we present the case study of a web-based multi-tier auction system, a typical example of a business-critical service with high dependability requirements.

4.1 Resource-efficient Long-term Dependability as an Infrastructure Service

Deploying and maintaining long-running stateful services with high dependability requirements in today's data centers involves significant manual efforts, in particular if a service needs to be resilient against intrusions and other Byzantine faults. To address this problem, we present SPARE, a system designed to provide an infrastructure service ensuring long-term dependability for service applications that run in virtualized environments. In the following, we outline the most important goals of SPARE, and give a brief introduction on how they are addressed in the system's design.

4.1.1 Resilience against Byzantine Faults in User Domains

Ensuring the availability of a service running in a virtualized environment involves protecting it against a wide spectrum of faults including server crashes, software malfunctions, and malicious intrusions. However, commercial tools available today [146, 156] as well as most approaches proposed by academia [28, 53] so far only provide resilience against crashes of system components. As a result, manual efforts are required to make an existing crash-tolerant service resilient against Byzantine faults, which, for example, can be achieved by relying on a Byzantine fault-tolerant middleware [25, 34, 42] that is run alongside the service application inside the same virtual machines [45].
SPARE builds on the ideas of VM-FIT [130] and ZZ [159] and addresses these problems by providing an infrastructure service that frees data-center users of the need to take care of fault tolerance themselves. Resilience against crashes of system parts is achieved by introducing redundancy (i.e., multiple physical servers as well as execution replicas). In addition, in order to be able to tolerate Byzantine faults in user domains (i.e., separate virtual machines, see Section 2.2.1) running the service application, before returning the result of a request to the client, SPARE verifies the correctness of the result based on a comparison of the replies provided by different execution replicas.

4.1.2 Resource Efficiency

Making a service resilient against Byzantine faults by using existing approaches comes with a high resource overhead: Without support of the underlying infrastructure, at least $3f + 1$ physical servers (see Section 2.2.1), each hosting an execution replica of the service in a virtual machine, are required in order to tolerate up to f faults [34]. Relying on special-purpose infrastructure services, this number can be reduced to $2f + 1$ [130, 152]. Furthermore, independently of the number of servers, most existing systems that are resilient against Byzantine faults [34, 41, 49, 130, 143] (including VM-FIT) suffer from an additional drawback: The use of plain active replication leads to a high resource usage during normal-case operation due to permanently keeping enough execution replicas active to tolerate faults (see Section 3.1). ZZ [159] constitutes an exception due to using only $f + 1$ execution replicas in the absence of faults and creating up to f additional execution replicas during fault handling. However, the agreement stage of ZZ is distributed across $3f + 1$ servers and consequently has a large resource footprint.

SPARE addresses these problems by using only $f + 1$ physical servers, thereby minimizing the resource footprint of the overall system. In addition, SPARE partially applies passive replication and saves resources by comprising a minimal setting in periods during which no fault-handling procedures are necessary. Only in situations where the outputs of additional execution replicas are actually required to tolerate a fault, the system spends further resources by increasing the number of active execution replicas.

4.1.3 Efficient Fault Handling

Ensuring high availability for critical services implies a fast response to occurring faults. In particular, fault events should not slow a service down to a point where it is rendered unusable. Instead, in the optimal case, fault handling is transparent to (human) clients. Existing systems that are able to tolerate Byzantine faults [34, 41, 49, 130, 143] (including VM-FIT) rely on active replication and consequently keep the state of a sufficiently large number of execution replicas up to date at all times in order to ensure progress in the presence of faults.

The design decision to minimize the resource footprint during normal-case operation prevents SPARE from applying the same approach as systems purely based on active replication. Even worse, although passive replication offers the potential of saving resources, it also comes with a drawback compared to active replication: Not keeping backup execution replicas up to date at every moment in time usually leads to increased fault-handling latencies. In ZZ [159], for example, the fact that backup execution replicas first have to acquire the service state before being able to assist in fault handling may result in latencies of multiple seconds.

SPARE addresses these problems by combining the advantages of virtual machines (e.g., fast activation) with an efficient mechanism to bring execution replicas up to speed based on state updates. As a result, the system is able to save resources during normal-case operation while still responding quickly to faults. Note that, as all fault and intrusion-tolerant systems considered in this thesis (see Section 2.1.1.3), SPARE however does not guarantee upper bounds on fault-handling latencies.

4.1.4 Long-term Resilience

When a critical service runs continuously for a long period of time (potentially years), it is likely that the number of actual faults eventually exceeds the upper bound of faults the service has been dimensioned for at deployment time. As discussed in Section 2.2.3, proactive recovery [34] periodically replaces execution replicas with clean, non-faulty versions and is therefore an effective means to cope with faults and/or intrusions. However, applying this technique usually either requires additional replicas [143] or, as in VM-FIT [130], leads to service disruptions due to active execution replicas being required to assist in the recovery procedure.

SPARE improves on existing approaches by relying on a proactive-recovery mechanism that only involves passive execution replicas during normal-case operation. This way, active execution replicas are able to process client requests without interruption until being replaced at the end of the recovery process.

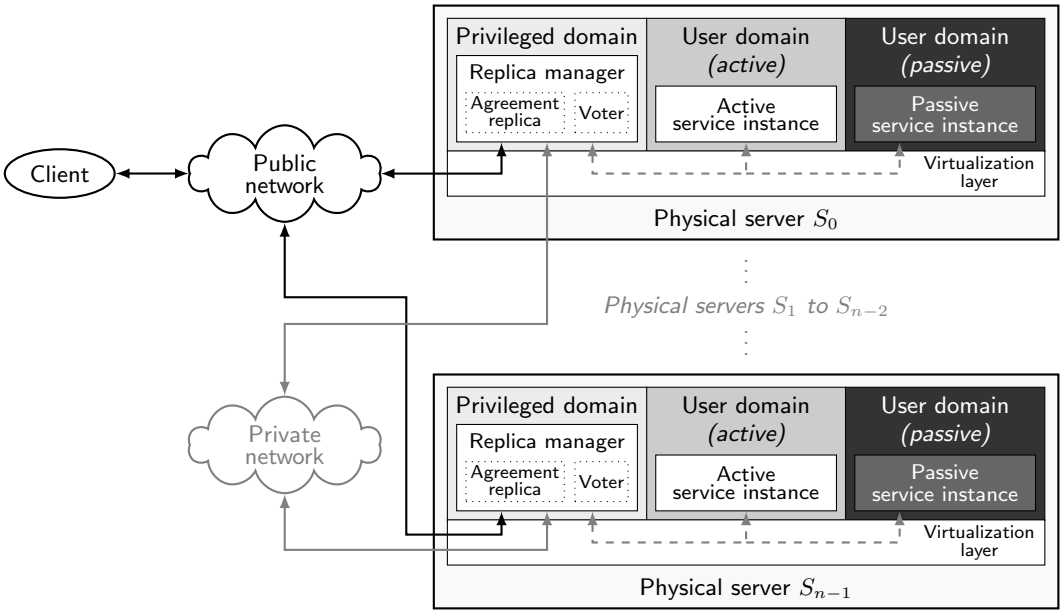


Figure 4.1: Overview of the SPARE architecture: Each server hosts an application-independent replica manager in its privileged domain as well as two execution replicas, an active and a passive one, each in its own user domain. While clients access the service through a public network (—), replica managers on different servers communicate using a separate private network (—); independently, domains on the same server exchange messages via a local network interface (- -).

4.2 The SPARE Architecture

For SPARE, we modify and extend the VM-FIT [130] architecture to provide long-term dependability as an infrastructure service in a resource-efficient manner. In this section, we discuss assumptions on SPARE's execution environment and service applications; furthermore, we present the fault model of SPARE as well as the architecture of its replicas.

4.2.1 Environment

The SPARE architecture is designed for application in a single data center where latencies between participating physical servers are low. In the following, we use the term *cell* to refer to all servers hosting different replicas of the same service. Despite being hosted in the same data center, servers of the same cell should be part of independent fault domains (e.g., different racks with redundant networking and power [32]) in order to prevent a single cause (e.g., a power outage) from bringing down the entire system.
As shown in Figure 4.1, replicas belonging to the same SPARE cell are connected via two types of network [45, 119, 143]: a *public network* (e.g., the Internet) through which clients invoke the service application, and a completely separate *private network* used only for SPARE-internal communication between replicas. While the public network

```
1 /* Execution of service requests */
2 [REPLY, UPDATE] processRequest(REQUEST request);

4 /* Application of state updates */
5 void applyUpdate(UPDATE update);

7 /* State transfer for proactive recovery */
8 STATE getState();
9 void setState(STATE state);
```

Figure 4.2: Overview of the functionality required from a service application in order to be integrated with SPARE (pseudocode): Besides providing means to retrieve and apply state updates, which reflect the state modifications caused by processing client requests, a service instance must comprise methods for getting and setting its application state.

might be open to arbitrary clients, access to the private network is limited to SPARE replicas only. Taking these characteristics as well as today's data-center infrastructures into account, it is feasible to assume a partially synchronous system model [61] for the public network and a synchronous system model for the private network [45, 130]; that is, in the private network replica crashes can be detected within a certain period of time [53, 66, 143, 146, 156]. In consequence, $f + 1$ physical machines in a SPARE cell are sufficient to tolerate up to f server crashes.

4.2.2 Service Application

SPARE has been developed for use with arbitrary network-based service applications which fulfill the following properties: First, to ensure consistency, each instance of the application that serves as an execution replica implements the same deterministic state machine (see Section 2.1.2.3). Second, as shown in Figure 4.2, having processed a request modifying the service state (see Section 2.1.1.5), an instance not only returns a reply but also a state update that reflects all relevant changes (L. 2); for read-only requests the state update is empty. Third, when supplied with a state update, an instance deterministically applies the corresponding modifications to its state (L. 5). Fourth, in order to support proactive recovery of execution replicas, an instance provides methods for retrieving and setting its state (L. 8 – 9).
As a consequence of these requirements, legacy applications may have to be ported in order to be integrated with the SPARE infrastructure; in general, the same is true for all infrastructures relying on passive replication. However, to facilitate integration, unlike existing approaches [146, 156], SPARE does not require service applications to be modified to use an external storage area network. Instead, application instances may manage their state both in memory as well as on disk, exactly as they do in a non-virtualized environment. In Section 4.10, we study the costs of integrating a multi-tier web application with SPARE in detail.

	System component	
Crashes	Privileged domain	SPARE logic
		Operating system
	Hardware	
Byzantine faults	User domains	Service application
		Middleware
		Operating system
	Client	

Figure 4.3: Overview of the SPARE fault model: As state-of-the-art commercial tools for achieving high availability [146, 156], SPARE is resilient against crashes. However, in addition, SPARE is also able to tolerate Byzantine faults in user domains as well as malicious behavior of clients.

In addition to the requirements mentioned above, applications will benefit to a special degree from SPARE's resource savings if they also provide both of the following characteristics: First, the application workload comprises a large fraction of read-only requests not having any effect on the service state when being processed. Second, the size of a state update emitted by an application instance is much smaller than the size of the corresponding client request that led to the update. Note that both characteristics lead to a reduction of the overhead necessary for bringing passive replicas up to speed due to minimizing the number and size of state updates that have to be sent over the network.

4.2.3 Fault Model

SPARE relies on the same hybrid fault model as VM-FIT [130], as shown in Figure 4.3: In accordance with the current best practice in industry [30, 55, 84, 146, 156], we assume a non-hostile operation environment that only fails by crashing. In contrast, processes running in user domains, and the service application in particular, may fail arbitrarily.

Servers SPARE assumes the server-side system components not running inside a user domain, including the virtual-machine monitor and the privileged domain, to only fail by crashing. Besides being the common approach in today's data centers, we consider treating these components as part of the trusted computing base to be justified because both the virtualization layer as well as the system components in the privileged domain are usually thoroughly tested and well maintained as, unlike the service applications deployed by users, they are under direct control of the data-center operator.
Apart from the system components that are assumed to only fail by crashing, in contrast to traditional virtualized environments, SPARE is designed to tolerate Byzantine faults in the virtual machines running services on behalf of users: the user domains; besides hosting the service application, user domains in SPARE also comprise their own operating system and middleware instances. Due to running arbitrary and possibly untested code, these system parts are particularly vulnerable to faults and intrusions. In consequence, system components executed in user domains will benefit from the additional resilience against Byzantine faults.

As any other fault-tolerant system, SPARE is not able to tolerate an arbitrary number of faults; instead, at deployment time, an upper bound f needs to be finalized. Note that although SPARE uses different techniques to address different categories of faults, with regard to this upper bound, we do not distinguish between crashes and Byzantine faults; that is, the crash of a physical server is considered to be a single fault, and so is the crash of a system component running in the privileged domain or the misbehavior of a process in a user domain caused by an intrusion.

Clients The upper bound of at most f faults a SPARE cell is able to tolerate exclusively covers faults of server-side components. In addition, an arbitrary number of clients may behave in a Byzantine way and, for example, send manipulated service messages in order to try to corrupt system parts that run in user domains. Note that we only consider faulty behavior of clients whose effects are limited to these areas; protecting the overall system against a malicious client breaking the isolation property of a user domain, for example, is outside the scope of SPARE. Given these assumptions, a SPARE cell is able to ensure safety (see Section 2.1.1.3) as long as the number of server-side faults, some or all possibly triggered by actions of Byzantine clients, does not exceed f.

4.2.4 Replicas

As depicted in Figure 4.1, a physical server in SPARE is host to multiple virtual machines comprising different parts of the system. In the following, we present the division of tasks between the privileged domain and the two user domains in detail.

Privileged Domain On each server, the privileged domain runs an application-independent *replica manager*, which combines both an agreement replica (see Section 2.1.2.2) as well as a voter (see Section 2.1.1.4). Besides, the replica manager is responsible for additional tasks including interception and propagation of client requests, management of state updates, and proactive recovery of execution replicas (see Section 2.2.3).
SPARE treats not only the agreement-replica part of a replica manager but in fact the entire agreement stage as a black box and makes no assumptions on how the totally ordered sequence of requests (i.e., the output of the agreement stage, see Section 2.1.2.2) is established. In consequence, different crash-tolerant ordering protocols (e.g., Paxos [103]) can serve as a basis for the SPARE agreement stage. To reduce implementation overhead, one might also rely on an existing group communication framework (e.g., Spread [10] or JGroups [85]) for this purpose.

User Domains In addition to the privileged domain, a SPARE server hosts two user domains each comprising an execution replica of the service application. Execution replicas never interact directly with clients or SPARE components other than their local replica manager, which acts as a relay for service messages. As user domains do not have to be accessible from the public or private network, to ensure isolation, they are connected to the privileged domain via a local network interface.

During normal-case operation, only one of the user domains is *active*; that is, the virtual machine is running and its execution replica is processing client requests. In the absence of faults, replies from the active execution replicas of all $f + 1$ physical servers in the cell are sufficient to prove a result correct (see Section 2.2.2). Therefore, SPARE saves resources by keeping the other user domain on each server in a *passive* state, in which the corresponding execution replica does not provide any service, but is only woken up from time to time to update its state. As described in Section 4.4, passive user domains are only activated in case of suspected or detected faults when replies from additional execution replicas are needed to verify the result of a request.
Relying on virtual machines to host execution replicas offers several benefits: First, the isolated environment provided by the underlying virtual-machine monitor limits the effects of Byzantine faults in a service-application instance to the virtual machine the execution replica runs in. Second, virtualization allows SPARE to co-locate active and passive execution replicas on the same physical server. As a result, the system is able to tolerate f Byzantine faults in user domains using only a minimum number of $f+1$ servers. Third, starting/unpausing a virtual machine is much faster than booting/resuming a physical machine [81]. In consequence, the process of activating additional execution replicas to tolerate faults is sped up [159], minimizing service disruption. Fourth, support for the proactive recovery of execution replicas can be provided without additional physical servers when utilizing virtualization (see Section 4.5). Fifth, the use of virtual machines helps system builders to improve fault independence of execution replicas by facilitating the challenge of integrating heterogeneous implementations (see Section 4.6).

4.3 Virtualization-based Passive Replication

In this section, we describe how SPARE handles client requests during normal-case operation while utilizing virtualization to minimize its resource footprint at the same time. Below, we only focus on the basic measures that enable the system to tolerate Byzantine faults in execution replicas; particular fault-handling procedures are presented and discussed in detail in the next section.

4.3.1 Normal-case Operation

In order to use the service application hosted by a SPARE cell, a client establishes a network connection to one of the servers; to distribute communication overhead across machines, different clients may be linked to different servers, for example, by using a load balancer. As shown in Figure 4.4, when a client invokes a service operation by sending a request o, the replica manager of the server the client is connected to intercepts the request and introduces a $\langle \text{REQUEST}, id_{RM}, o \rangle$ message into the agreement stage; id_{RM} is the replica manager's unique id (i.e., in the example in Figure 4.4: $id_{RM} = R_0$).
Besides establishing a total order, the agreement stage is responsible for propagating each client request to all agreement replicas. Once a request is ordered, each replica manager on a non-faulty server therefore receives an agreement certificate $\langle \text{AGREED}, id_{RM}, s, o \rangle$

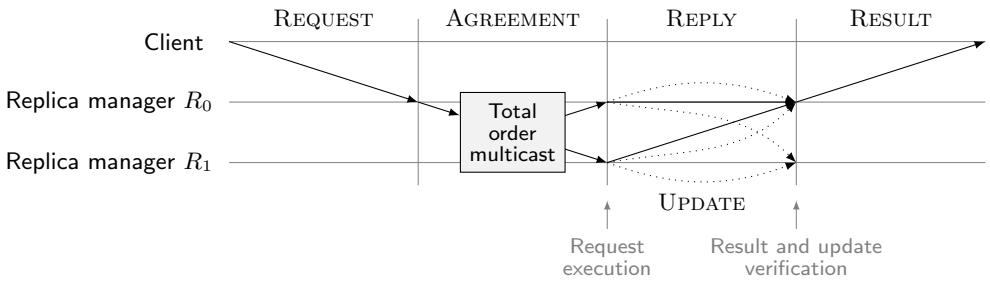


Figure 4.4: Message flow of requests and replies (—) as well as state updates (····) in SPARE for a cell that is able to tolerate one fault: Before returning it to the client, the replica manager that has initially received the request verifies the result based on the replies provided by different execution replicas; in contrast, state updates are verified by each replica manager independently.

from the agreement stage; s is a sequence number indicating the position of request o in the totally-ordered sequence. At this point, all replica managers hand over the request to the execution replica running in their respective active user domains.
Having processed a client request, an active execution replica id_E returns its corresponding reply v to the local replica manager, which in turn adds it to its local reply cache and forwards a $\langle \text{REPLY}, id_E, s, v \rangle$ message to replica manager id_{RM}; that is, the replica manager that has initially received request o with sequence number s. To verify the result for request o, replica manager id_{RM} collects the replies from different replica managers (including its own) and hands them over to its local voter. On successful verification, id_{RM} sends the stable result back to the client that has issued the request.

4.3.2 Passive Execution Replicas

In the absence of faults, replies from the $f + 1$ active execution replicas are sufficient to successfully verify the result to a request. However, to actually tolerate up to f faults under less ideal circumstances, a total of at least $2f + 1$ replies are required. The traditional approach [34, 41, 49, 130, 161] to address this problem is to generate the deciding replies by relying on additional active execution replicas, each running on its own physical server, which consequently leads to an increased resource usage during normal-case operation. In contrast, SPARE combines virtualization and passive replication to keep backup execution replicas available on the physical servers that are already there; to save resources, the backup replicas are only activated when their output is actually needed.

The Need for State Updates In order to be able to create a consistent reply to a client request o on demand, a non-faulty passive execution replica must have the same application state as a non-faulty active execution replica at the time it is going to process o. SPARE solves this problem by providing each passive execution replica with a sequence of state updates that allows the replica to reproduce state changes triggered by client operations without executing the requests itself. To prevent passive execution replicas

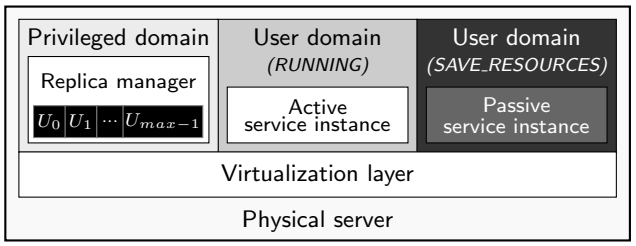
from being contaminated during this procedure, which would render them useless for the purpose of producing additional independent replies, replica managers verify state updates before handing them over.

Providing and Verifying State Updates The sequence of verified state updates is established as follows: When an active execution replica id_E processes a state-modifying request with sequence number s, it not only provides a reply v but also a state update u that reflects the request's impact on the application state. In the next step, each replica manager distributes both the update and the reply of its respective local execution replica in an $\langle \text{UPDATE}, id_E, s, u, v \rangle$ message between all other replica managers (see Figure 4.4). Having $f + 1$ UPDATE messages available, during normal-case operation, each replica manager is therefore able to verify the correctness of the state update by voting independently. In consequence, there is no additional interaction with other replica managers required to learn the outcome of the verification process of a state update; in Section 4.8 we discuss additional implications as well as an alternative solution for the problem of update verification. Note that in the approach presented above, as a result of including the reply, an UPDATE message only becomes stable if both the reply and the update have been verified. At this point, it is therefore safe (see Section 4.7) to apply the state update to passive execution replicas.

Updating Passive Execution Replicas Upon successful verification, a replica manager does not forward a state update to its local passive execution replica right away: As the user domain of a passive replica is kept in a resource-saving mode, it needs to be woken up temporarily in order to be able to update its execution replica. To minimize this overhead, SPARE reduces the overall number of operation mode switches for the corresponding virtual machine by applying state updates in batches. Thus, a replica manager first inserts a verified state update into an intermediate buffer. When the buffer size reaches a certain threshold U_{max}, the replica manager wakes up the local passive user domain (see Figure 4.5b) and hands the buffered batch of state updates over to the execution replica (see Figure 4.5c), thereby relying on the sequence numbers of updates to preserve the order determined by the agreement stage. Having received a confirmation from the execution replica that all state updates in the batch have been applied, the replica manager instructs the virtual-machine monitor to resume the resource-saving mode for the passive user domain (see Figure 4.5d).

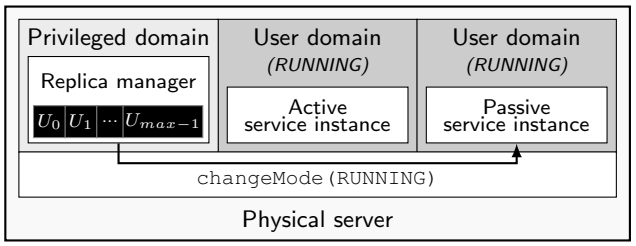
4.4 Fault Handling

The previous section presented the processing steps of SPARE during normal-case operation when all active replicas provide correct replies. In the following, we describe the mechanisms the system uses to tolerate detected or suspected faulty behavior of components. Note that in this context we omit a discussion of internal fault-handling procedures of the agreement stage as SPARE uses this component as a black box (see Section 4.2.4).



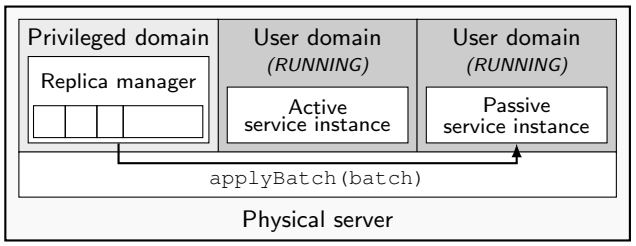
Global data structures		
UPDATEBUFFER	`upds`	Update buffer
REPLICA	`pr`	Passive replica

(a) Initial situation prior to update



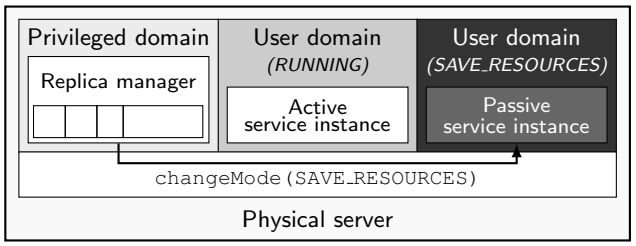

```
/* Check whether update buffer is full. */
if(upds.size() == U_max) {
  /* Wake up passive replica. */
  pr.changeMode(RUNNING);
}
```

(b) Phase 1: Preparation




```
/* Create state-update batch. */
BATCH batch := upds.collect();
/* Apply updates to passive replica. */
pr.applyBatch(batch);
```

(c) Phase 2: Update




```
/* Wait until update is complete. */
while(!pr.updComplete()) wait;
/* Resume resource-saving mode. */
pr.changeMode(SAVE_RESOURCES);
```

(d) Phase 3: Conclusion

Figure 4.5: Overview of the three-phase process for bringing a passive execution replica up to speed (simplified pseudo code): In order to apply the batch of buffered state updates, the passive execution replica temporarily has to discontinue resource-saving mode.

4.4.1 Suspected vs. Detected Faults

Replica managers verify the correctness of a result/update by comparing the replies/updates provided by different execution replicas. With the system being designed for tolerating at most f faults, verification is successful as soon as $f+1$ matching replies/updates are available, because at least one of those replies/updates must be correct due to having been created by a non-faulty execution replica. As a result of SPARE's approach to save resources by relying on only a minimum of $f+1$ active execution replicas, the verification process of a result *stalls* in case of faults or request-execution delays; that is, a replica manager is temporarily not able to successfully complete the verification process due to having obtained less than $f+1$ matching replies/updates. To address this problem, a key requirement for subsequent fault-handling procedures is to produce additional replies/updates in order to enable the replica manager affected to continue and eventually complete the stalled verification process.
Based on the effects observed by the replica manager that performs the voting, we distinguish between *suspected* and *detected* faults: If a replica manager obtains less than $f+1$ replies/updates within a certain period of time, it suspects the execution replicas that failed to contribute a reply/update to be faulty. Note that in such a scenario the replica manager has no definitive proof that the suspected execution replicas are actually faulty as their provision of replies/updates just might have been delayed. In contrast, a replica manager is able to detect the presence of a fault if replies/updates obtained from different execution replicas diverge: In such case, the mismatch indicates right away that (at least) one of the replies/updates must be faulty. However, to identify which execution replicas provided faulty replies/updates, a replica manager first has to wait until it learns the outcome of the verification process (i.e., the correct result/update).
Suspected and detected faults both cause the verification process of a result/update to stall. Thus, the immediate reactions of a replica manager to their occurrences are the same: The replica manager requests additional replies/updates from passive execution replicas to ensure progress.

4.4.2 Stalled Result Verifications

SPARE's fault-handling mechanism has been designed to provide a replica manager facing a stalled result verification with the minimum number of additional replies it needs to be able to successfully complete verification: Instead of activating all passive execution replicas at once, the number of replicas to be included in tolerating the fault depends on the progress that has already been made in the verification process.

Basic Fault-handling Mechanism If the verification of a result stalls due to a suspected or detected fault, the replica manager id_{RM} responsible for performing the voting inserts a $\langle \text{STALLED}, id_{RM}, s, \mathcal{E}, count \rangle$ notification into the agreement stage comprising the agreement sequence number s of the request o_s whose result is in question. In addition, the STALLED message includes $\mathcal{E}$, a set containing the ids of all execution replicas that

Global data structures		
MESSAGESTORE	msgs	Store for relevant messages (e.g., replies, requests, and control messages)
UPDATEBUFFER	upds	Buffer storing state updates that have not yet been applied
REPLICA	ar	Local active execution replica
REPLICA	pr	Local passive execution replica

```
S1  void handleStalled(STALLED stl) {
S2    /* Resend reply if available. */
S3    if(ar.id ∉ stl.ℰ) {
S4      REPLY rpy := msgs.getRpy(stl.s);
S5      if(rpy != null) {
S6        send(rpy, stl.id_RM);
S7      }
S8    }
S10   /* Store notification using seq. nr. as key. */
S11   msgs.add(stl.s, stl);
S13   /* Submit offer to help. */
S14   OFFER ofr :=
          new OFFER(pr.id, stl.s);
S15   send(ofr, stl.id_RM);
S16 }
```

```
O1  void handleOffer(OFFER ofr) {
O2    /* Store offer using seq. nr. as key. */
O3    msgs.add(ofr.s, ofr);
O5    /* Check whether to activate passive replica. */
O6    OFFERs ofrs := msgs.getOfrs(ofr.s);
O7    STALLED stl := msgs.getStl(ofr.s);
O8    if(|ofrs| > stl.count) return;
O9    if(ofr.id_E != pr.id) return;
O11   /* Activate local passive execution replica. */
O12   pr.changeMode(RUNNING);
O13   while(!pr.ready()) wait;
O14   upds.flush(ofr.s − 1);
O15   REQUEST o_s := msgs.getReq(ofr.s);
O16   Forward request o_s to replica pr;
O17 }
```

Figure 4.6: Basic mechanism to initiate fault handling in SPARE (simplified pseudo code): When informed about a stalled verification via a STALLED message, replica managers use OFFER messages to negotiate on which server to activate passive execution replicas.

have already contributed a reply, as well as a parameter $count = f + 1 - m$, which indicates the minimum number of additional replies that are expected to be required in order to successfully complete result verification; the replica manager calculates the value of $count$ by subtracting the maximum number of matching replies currently available m from the verification threshold of $f + 1$.

As shown in Figure 4.6 (`handleStalled` method), upon receiving a STALLED notification, a replica manager checks whether the id of its local active execution replica is included in $\mathcal{E}$ (L. S3). If this is not the case although the replica has already processed request o_s, the replica manager retransmits the corresponding reply to id_{RM} (L. S4 – S6). Independently, the replica manager prepares an $\langle \text{OFFER}, id_E, s \rangle$ message and introduces it into the agreement stage to signal that its local passive execution replica id_E is ready to assist in tolerating the fault (L. S14 – S15). Following this, the agreement stage takes care of ordering the OFFER messages for request o_s and distributing them between all non-faulty replica managers.

Upon receiving an OFFER message (see Figure 4.6, `handleOffer` method), if its own OFFER message is among the first $count$ offers in the output of the agreement stage, a replica manager's local passive execution replica has been selected to take part in fault handling (L. O3 – O9). In such case, the replica manager performs the following steps:

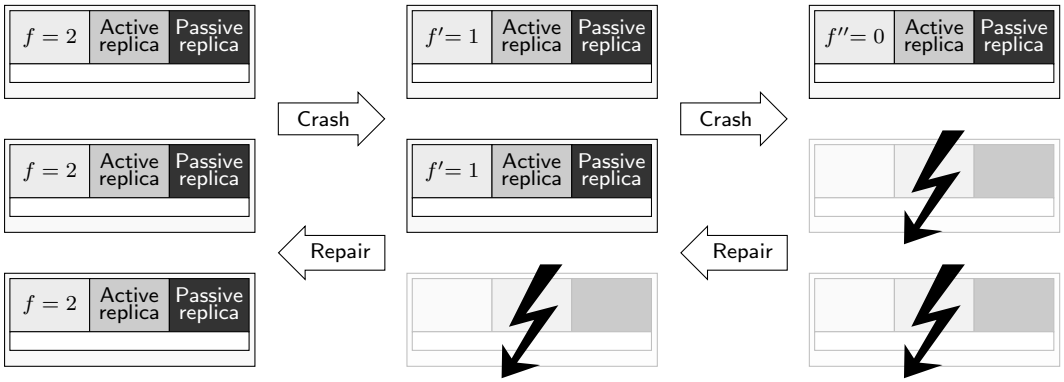
First, it wakes up the passive user domain (L. O12 – O13). Next, it flushes the update buffer (see Section 4.3.2) and forwards all state updates with agreement sequence numbers lower than s to the passive execution replica (L. O14). Having applied the verified state updates, the replica has the same application state as a non-faulty active execution replica before having processed request o_s. In the next step, the replica manager activates the replica by issuing request o_s (L. O15 – O16) and waits for the (now former) passive execution replica to return a reply. When the reply becomes available, the replica manager forwards it to the replica manager responsible for the verification of request o_s.

Extended Fault-handling Mechanism Enabling a replica manager to use the $count$ parameter of a STALLED notification to ask for a specific number of additional replies allows SPARE to save resources even during fault handling; the alternative would have been to activate all passive execution replicas upon a stalled verification. However, it is not guaranteed that a single STALLED notification automatically leads to a successful verification in all cases: For example, the reply provided by a passive execution replica might be faulty. Also, the value of the $count$ parameter might have been too low in order to produce enough correct replies; such a scenario occurs when the majority of matching replies currently available indicates the same but wrong result. To ensure progress in such situations, a replica manager protects each STALLED notification it issues with a timeout that triggers if the verification process has not been successfully completed within a certain period of time. On timeout expiration, a replica manager starts another round of fault handling by introducing another STALLED notification, with an updated value for $count$, into the agreement stage in order to get replies from additional passive execution replicas. This process continues until the result corresponding to the request in question has been successfully verified.

4.4.3 Stalled Update Verifications

The verification process for a state update in SPARE, as described in Section 4.3.2, is similar to the steps necessary to prove the correctness of a reply. In consequence, the fault-handling procedure triggered for state updates upon a stalled verification is basically the same as for replies (see Section 4.4.2): A replica manager that is not able to verify a state update within a certain period of time distributes a STALLED notification via agreement stage indicating the agreement sequence number of the corresponding request that caused the state modification. Following this, OFFER messages are used to select a set of passive execution replicas responsible for tolerating the fault by actively processing the request in order to provide additional state updates.

The main difference in fault handling between replies and state updates is related to the fact that verification for a particular state update is not only performed once by a single replica manager but by all non-faulty replica managers independently. As a result, multiple STALLED notifications issued by different replica managers may be distributed for the same agreement sequence number. In order to prevent unnecessary activations of passive execution replicas, in such case, replica managers only act on the first notification they receive and ignore all others issued during the same round of fault handling.



(a) All servers working. **(b)** One server has crashed. **(c)** Two servers have crashed.

Figure 4.7: Reaction to server crashes in SPARE by example of a cell dimensioned to tolerate a total of $f = 2$ faults: When a replica manager detects the crash of another server, it adjusts the number of remaining faults to $f' = 1$ (first crash) and $f'' = 0$ (second crash), respectively; as soon as a server has been repaired, the value is increased again.

4.4.4 Server Crashes

Compared to fault-tolerant systems based on pure active replication [34, 41, 130, 161], the crash of a physical server in SPARE is more severe, as not only one but two execution replicas (i.e., in the initial configuration, an active and a passive execution replica) are affected. As a consequence, special measures have to be taken to tolerate such faults as well as to ensure a safe client failover.

Server-side Fault Handling With more than one execution replica being affected directly, as a temporary result of server crashes, replica managers in a SPARE cell at first may not be able to make progress due to receiving too few (i.e., less than $f + 1$) replies or state updates from different execution replicas to complete a verification process: In the worst case of f server crashes, for instance, only a single physical server remains; as it hosts two execution replicas, without additional measures, progress would not be ensured for SPARE cells that are dimensioned to tolerate more than one fault (i.e., $f > 1$).
To address this problem, as shown in Figure 4.7, when detecting the crash of a physical server, a replica manager lowers the stability threshold for result and update verification processes to $f' + 1$, with $f' = f - f_{crashed}$ and $f_{crashed}$ being the total number of physical servers that have crashed. Note that this is safe due to the fact that, if $f_{crashed}$ faults have already occurred, the system is only required to tolerate $f' = f - f_{crashed}$ additional (Byzantine) faults in order to provide the dependability guarantees defined at deployment time; for this purpose, $f' + 1$ matching replies/updates from different execution replicas are sufficient to verify correctness.
Lowering the stability threshold in the presence of server crashes enables SPARE to make progress: With a remaining cell size of $f'+1$ physical servers, there is a sufficient number of execution replicas available to tolerate at most f' faults, comparable to a SPARE cell that has been dimensioned for f' faults in the first place. After a crashed server has been repaired or replaced, the stability threshold for verification processes is increased again.

Client Failover As the consequence of a server crash, the replica manager a client is connected to might no longer be able to return the result of a request to the client. However, replica managers running on other servers are prepared to step in by drawing on their local reply caches (see Section 4.3.1). Note that SPARE imposes no restrictions on how the switch to another replica manager is implemented. In order to achieve a seamless switch, for example, techniques for transparent TCP connection failover [5, 53, 95, 115, 140] could be applied, which allow to substitute the server side of a link while the connection remains open. Once the switch has been completed, the replica manager that took over initiates the standard verification process for the result in question. The replica manager is able to do so, as it not only has access to the reply of its local active execution replica but also to a set of state updates, and consequently to the corresponding replies included in them (see Section 4.3.2), provided by the execution replicas on other physical servers. On the successful completion of the verification process, the replica manager finally returns the correct result to the client.

4.4.5 Returning to Normal-case Operation

Having tolerated the stalled verification of a reply or state update, fault-handling procedures leave a SPARE cell in a state in which, in addition to the active execution replicas, one or more former passive execution replicas are running. With more replicas active than actually required for normal-case operation, continuing with this setting would result in resources being used unnecessarily. In the following, we discuss different measures to approach this problem. The overall idea behind them is, if circumstances permit, to keep the $f + 1$ execution replicas active that most likely operate correctly, and to instruct the remaining replicas to go to resource-saving mode.

Retirement and/or Demotion of Active Execution Replicas If fault handling has been triggered after a replica manager detected a fault based on a mismatch (see Section 4.4.1), successful completion of the verification process not only reveals the correct reply/update but also the identity of the execution replicas that provided faulty versions. A reasonable approach in such case is to assume that the source of the faults is of permanent nature (e.g., a corrupted application state) and will likely lead to continued faulty behavior in the future, if the affected execution replica is kept activate. Based on this assessment, SPARE may decide to retire a faulty execution replica, which includes shutting down its user domain and freeing all resources the replica held.
In order to force the retirement of an execution replica, a replica manager must have a definitive proof for the faulty behavior of the replica; suspecting a fault (see Section 4.4.1) is not enough as it may lead to a non-faulty replica erroneously being destroyed, possibly leaving the system with an insufficient number of non-faulty execution replicas and therefore vulnerable to actual faults. Instead, the safe strategy for active execution replicas suspected of faulty behavior is to demote them to passive replicas; that is, to put them into resource-saving mode and from then on modify their application state using state updates.

Besides knowledge about suspected or detected faults, the decision which execution replicas to keep active may also be affected by additional aspects, for example, load balancing considerations: Whenever possible, at the end of fault-handling procedures, SPARE tries to reach a configuration in which each physical server only hosts a single active execution replica. Therefore, SPARE first checks whether it has a sufficient number of (presumably) non-faulty replicas on different servers available, before keeping a former passive execution replica active on a server that already hosts another active replica.

Switching to Normal-case Operation Independent of the particular strategy applied to decide which execution replicas to demote and/or retire, SPARE always uses the same mechanism to implement the switch to normal-case operation. In addition to STALLED and OFFER messages (see Section 4.4.2) it relies on a $\langle \text{CONVICTED}, id_E \rangle$ notification a replica manager propagates via agreement stage as soon as it has proof that an execution replica id_E has actually provided a faulty reply or update. In combination with knowledge about server crashes as well as the strategy based on which decisions about activations, demotions, or retirements of execution replicas are made, these messages allow each replica manager to keep track of the state (i.e., active, passive, or retired) of all execution replicas in the system. Using the agreement stage to distribute this information ensures that all replica managers process the messages in the same order and therefore draw consistent conclusions.

4.5 Proactive Recovery

At deployment time, an upper bound f is defined specifying the maximum number of faults a SPARE cell must be able to tolerate at the same time; depending on this value, one can calculate the minimum size of the cell of physical servers (i.e., $f + 1$, see Section 4.2). However, being designed to offer dependability for long-running service applications (see Section 4.1), simply providing enough execution replicas to tolerate f faults is not enough: As faults accumulate over time, the system sooner or later is likely to reach its limit for any practical cell size. To address this problem, SPARE periodically recovers execution replicas, thereby cleaning them from the effects of potential faults and/or intrusions. Due to being performed proactively, this procedure not only constitutes a means against faults and intrusions that have already resulted in observable failure (e.g., faulty replies) but, as discussed in Section 2.2.3, also ones that so far remained undetected.

4.5.1 Basic Approach

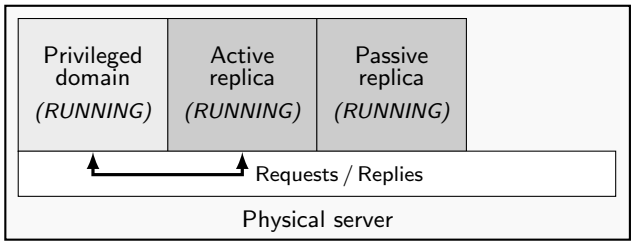
Proactive recovery for stateful services requires a system to create new execution replicas (i.e., the next *generation* of replicas) based on the latest state of the current generation of execution replicas. Note that, using the state model discussed in Section 2.1.1.5, for this procedure, only the service state actually has to be transfered between execution replicas of different generations. In contrast, the system state of a next-generation execution replica is created during its installation process and/or at boot time, as a preparation prior to the service-state transfer.

In actively-replicated fault and intrusion-tolerant systems [35, 130], participation in the state transfer puts additional load on the execution replicas processing client requests. To address this problem, passive execution replicas play a major role in the proactive-recovery procedures of SPARE, reducing the overhead for active execution replicas to a minimum. Furthermore, SPARE exploits virtualization to reduce the amount of data to be sent over the network by creating next-generation execution replicas on the same physical servers as their predecessors.
SPARE recovers execution replicas at the level of virtual machines; that is, not only the service-application instance of a replica is included in the recovery process but also the operating system and middleware running in its user domain. To set up a next-generation execution replica, first, its system state is created by starting a new user domain from a clean virtual-machine image that contains the operating system, middleware, and application software. In a second step, the service state of the next-generation execution replica is set to reflect the current state of the application. Once a new execution replica has been fully set up, the roles of execution replicas running on the same physical server are reassigned: The current passive replica is promoted to be the new active execution replica and the next-generation replica becomes the new passive execution replica. Finally, the former active execution replica is destroyed and all of its resources are freed.
In the following, we present and discuss two mechanisms used for proactively recovering execution replicas in SPARE: a *lightweight mechanism* and a *resilient mechanism*. The lightweight mechanism reduces the recovery overhead in the absence of faults by only relying on passive execution replicas to create the next generation of execution replicas. In contrast, the resilient mechanism is designed to operate in the presence of faults; it is more expensive than the lightweight mechanism due to active execution replicas also taking part in recovery procedures.

4.5.2 Lightweight Recovery Mechanism

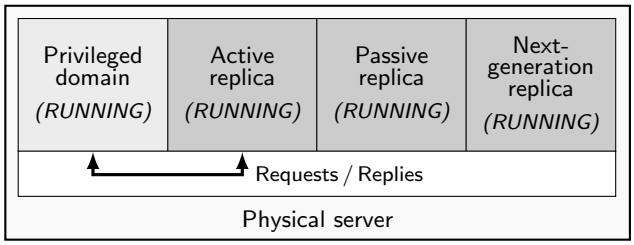
SPARE's lightweight recovery mechanism minimizes the overhead for proactive recovery by only relying on the service state of passive execution replicas to initialize next-generation execution replicas. Being an optimistic approach, a lightweight recovery procedure may not complete successfully; in such case, at the end of the next recovery period, recovery is performed using the resilient mechanism presented in Section 4.5.3.

Initiation of Recovery Procedure SPARE defines the interval between two recovery procedures as a number of service invocations: Having verified $U_{recovery}$ state updates, a replica manager triggers a new recovery procedure. Note that, due to this number being a system-wide constant, recovery procedures are triggered at the same point in logical time on all servers. To minimize overhead, $U_{recovery}$ should be selected as a multiple of the update-buffer size U_{max} (see Section 4.3.2). This way, as shown in Figure 4.8, the execution of a proactive-recovery procedure can be initiated after the second phase of the updating process of a passive execution replica: Instead of putting the passive execution replica back into resource-saving mode after having applied the update batch, during recovery, a replica manager keeps the replica running (see Figure 4.8a).



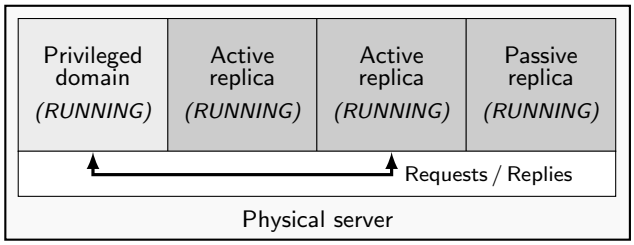
Global data structures	
REPLICA ar	Active replica
REPLICA pr	Passive replica

(a) Situation after phase 2 of the updating process (see Figure 4.5c)



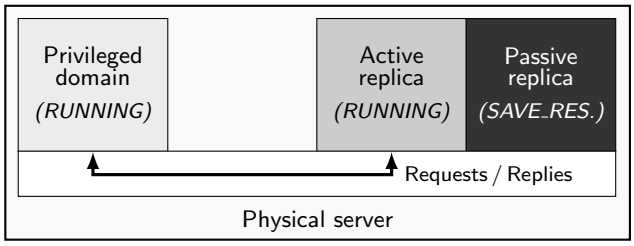

```
/* Create next-generation replica. */
REPLICA nr := new REPLICA();
/* Get and verify passive-replica state. */
STATE prState := pr.getState();
if(!prState.verify()) abort;
/* Set state of next-generation replica. */
nr.setState(prState);
```

(b) Phase 1: State transfer




```
/* Wait until replica nr is ready. */
while(!nr.stateReady()) wait;
/* Switch replica roles. */
REPLICA oldAR := ar;
ar := pr;
pr := nr;
```

(c) Phase 2: Replica switch




```
/* Put pr into resource-saving mode. */
pr.changeMode(SAVE_RESOURCES);
/* Destroy former active replica. */
oldAR.destroy();
```

(d) Phase 3: Cleanup

Figure 4.8: Overview of the lightweight mechanism for proactively recovering execution replicas in SPARE (simplified pseudocode): Having brought the passive execution replica up to speed in the course of an update process, a next-generation execution replica is created based on a verified copy of the passive replica's state. If state verification fails recovery is aborted. Otherwise, the passive execution replica is activated and takes over request processing. Furthermore, the next-generation execution replica becomes the new passive replica. At the end of the procedure, the user domain of the former active execution replica as well as all of its data are destroyed.

Creation of Next-generation Execution Replicas In order to set up a next-generation execution replica, a replica manager creates and starts an additional user domain on its local physical server using a clean virtual-machine image that contains all required software (e.g., operating system, middleware, service application). Note that SPARE makes no assumptions on how such an image is provided; to facilitate integration, for this purpose, one could therefore draw on storage components for virtual-machine images already available in today's data-center infrastructures [119]. As soon as the new user domain has completed booting and its instance of the service application is running, the next-generation execution replica is ready to receive the service state (see Figure 4.8b).

State Transfer During recovery, SPARE minimizes the amount of data that needs to be transmitted over the network by taking advantage of the fact that multiple execution replicas run on the same physical server. Instead of retrieving a full copy of the service state from an execution replica on another server, a replica manager relies on the state of the local passive execution replica to update the application instance of the next-generation execution replica. Note that, besides reducing network overhead, selecting the passive execution replica to provide the state offers additional benefits: First, the active execution replica is not involved in the process and therefore able to continue processing requests without experiencing additional load. Second, as further discussed in Section 4.5.4, a passive execution replica in general is more likely to comprise a correct copy of the state than an active execution replica.
Having retrieved the service state from its local passive execution replica id_E, a replica manager tries to verify the correctness of the state copy (see Figure 4.8b). For this reason, the replica manager computes a hash h_{state} over the state contents and distributes a $\langle \text{STATE}, id_E, s, h_{state} \rangle$ message between all replica managers in the cell; s is the agreement sequence number of the state update that triggered the current proactive-recovery procedure. As soon as the replica manager has obtained $f + 1$ matching state hashes for the same sequence number from different passive execution replicas, the state becomes stable. In consequence, it is safe to use the state contents to bring the next-generation execution replica up to speed.
Note that in the presence of faults a replica manager might not be able to prove the correctness of its local state copy, for example, due to one passive execution replica providing a faulty state hash. In such case, the replica manager aborts the recovery procedure and puts the next-generation execution replica into resource-saving mode until the end of the subsequent recovery period. At this point, recovery is performed using the resilient mechanism presented in Section 4.5.3, eventually leading to a properly updated next-generation execution replica.

Switch of Execution-replica Roles Once state transfer has completed successfully on a server, the local replica manager is able to abandon its current active execution replica and instead make use of the next-generation execution replica. As shown in Figure 4.8c, this is done by the replica manager assigning new roles to execution replicas: The current passive execution replica is promoted to be the new active execution replica and

exclusively processes client requests from then on. In addition, the next-generation execution replica becomes the new passive execution replica and is consequently brought up to speed by state updates.
Note that, unlike the creation of the state copy at the beginning of a recovery procedure, the switch of execution-replica roles does not necessarily have to occur at the same point in logical time on all servers. However, as in the context of fault handling, it must be ensured that a passive execution replica has applied all previous state updates prior to taking over request execution from the active execution replica.

Cleanup Having reassigned the roles of execution replicas, as shown in Figure 4.8d, a replica manager puts the new passive execution replica into resource-saving mode. In order to conclude the recovery procedure, the replica manager then garbage-collects the former active execution replica, which includes destroying the corresponding user domain and freeing all resources (e.g., disk space) the replica had allocated. In consequence, a proactive-recovery procedure results in the initial setting, which is a server hosting both an active execution replica and a passive execution replica. However, thanks to the recovery, the system has been cleared from potential detected, suspected, and even unsuspected faults in the former active execution replica.

4.5.3 Resilient Recovery Mechanism

SPARE's optimistic lightweight recovery mechanism presented in Section 4.5.2 only succeeds if all of the $f + 1$ passive execution replicas provide correct copies of the current service state. As this may not always be the case, the system relies on a resilient recovery mechanism that, on the one hand, is resilient to faults but, on the other hand, also more expensive as active execution replicas have to participate in the process.

Recovery Procedure Due to the fact that the resilient recovery mechanism is in most parts identical to the lightweight recovery mechanism, namely the initiation, the creation of next-generation execution replicas, the switch of execution-replica roles, as well as the cleanup (see Section 4.5.2), in the following, we concentrate on the main difference: the state transfer. Instead of only the passive execution replicas taking part in this phase, the resilient recovery mechanism also requires active execution replicas to provide their service state. As a result, each replica manager creates and distributes two STATE messages, one for each of its local execution replicas. With a total of $2f + 2$ state hashes from different execution replicas becoming available in the cell, each replica manager will eventually be able to obtain $f+1$ matching hashes and consequently learn the hash of the correct service state. In case the correct hash corresponds to one of the two state copies provided by local execution replicas, a replica manager directly concludes the state transfer by handing over the verified copy to its local next-generation replica. Otherwise, a replica manager first has to fetch the correct version of the state contents from another server in the cell.

Comparison to Lightweight Recovery Mechanism Being an optimistic mechanism, the lightweight recovery procedure does not complete successfully in the presence of faults. Furthermore, due to only relying on passive execution replicas, there might be cases in which the mechanism cannot be applied, for example, if all execution replicas on a server are active as the result of fault-handling procedures.
In contrast, the resilient recovery mechanism is always applicable and guaranteed to make progress even if up to f of the participating execution replicas provide faulty states. However, this resilience and flexibility comes at a cost: First, requiring active execution replicas to create a consistent snapshot of their current state conflicts with their main task of processing requests (see Section 2.1.2.3), therefore impairing performance. Second, concurrently managing state copies from two execution replicas instead of one results in additional resource usage on a server, for example, leading to significantly increased memory consumption in case of large service states. Third, in the unlikely event that none of the local execution replicas has provided a correct state copy, the full contents of the state have to be transferred over the network in order to initialize the next-generation execution replica.

4.5.4 Discussion

In the following, we discuss a number of problems that arise when applying the proactive-recovery mechanisms discussed in Sections 4.5.2 and 4.5.3 in practice.

Selection of Recovery Mechanism If a lightweight recovery procedure is aborted due to faulty hashes preventing a successful verification of the service state, SPARE postpones the initialization of next-generation execution replicas until the end of the subsequent recovery period. Note that an immediate switch to the resilient recovery mechanism is not possible as this would require active execution replicas to go back in time in order to create a consistent snapshot of their service state. Considering the delay in the presence of faults, the lightweight recovery mechanism should only be applied if it is likely to succeed. In the following we discuss criteria for making this decision.
Having caused fault-handling procedures due to showing suspicious or detectably faulty behavior, active execution replicas may be demoted to passive execution replicas (see Section 4.4.5). In such cases, there is an increased risk that those execution replicas will provide faulty states during recovery. However, in the absence of prior fault-handling procedures, the probability of comprising a corrupted state is very low if a passive execution replica has never been active: Such an execution replica has never processed a single client request but has only been brought up to speed using verified state updates. Given that replica managers in SPARE prevent clients from directly communicating with user domains (see Section 4.2.4), it is feasible to assume that the service state of such a passive execution replica could not have been manipulated by a malicious client. In consequence, the state could have only been corrupted or lost as the result of a non-malicious fault, for example, a disk failure.

Taking these considerations into account, replica managers in SPARE only apply the lightweight recovery mechanism if it is likely to complete successfully. Furthermore, replica managers use knowledge about detected faults to specifically eliminate faulty execution replicas by modifying the reassignment process of roles during recovery: for example, if an execution replica has been retired due to a fault, it is replaced by the next-generation execution replica, regardless of whether the execution replica was the next in line to be removed or not.

Evolution of Execution Replicas Although offering a solution to clear a system from detected, suspected, and even unsuspected faults in the service state of execution replicas, there is one category of faults plain proactive recovery is not able to address: bugs in the software running in the user domain (e.g., the service application). If, for example, a malicious client manages to exploit a vulnerability in the application code to take over an execution replica, the proactive-recovery procedure clears the effects of the intrusion. However, it is likely that the adversary will succeed in also taking over the new active execution replica within a short period of time using the same approach as before.

To address this problem, SPARE supports applying the concept of evolving execution replicas [23], allowing the software of execution replicas to change from generation to generation. By creating a next-generation execution replica based on a virtual-machine image to which the latest patches have been applied, software bugs can be eliminated over time, thereby reducing the probability that an adversary exploits the same vulnerability twice.

In general, the fact that next-generation execution replicas in SPARE are completely built from scratch during a proactive-recovery procedure greatly facilitates an evolution of execution replicas. This is especially true for software updates that only include internal code modifications not leading to any observable effects. However, in cases where updates change the representation or contents of the service state, additional measures (e.g., in the form of conversion routines) have to be taken in order to allow SPARE to initialize next-generation execution replicas with a verified copy of the state.

Implementation Hints Although the presentation of SPARE's proactive-recovery mechanisms in Sections 4.5.2 and 4.5.3 suggests a strictly sequential process, a practical implementation should parallelize some of the tasks in order to increase efficiency: For example, due to the fact that a user domain takes multiple seconds to boot, the next-generation execution replica should be started a sufficiently long time before the updating process at the end of which the passive execution replica is promoted to become the new active execution replica. Furthermore, for large service states, instead of computing a single hash over the entire state provided by an execution replica, a replica manager should compute hashes over parts of the state. This way, verification of the first parts can already be initiated while retrieval of other parts is still in progress. Note that splitting the service state into parts also allows replica managers to increase efficiency when fetching a full state copy as different parts may be transferred from different servers [24, 88].

4.6 Fault-independent Execution Replicas

SPARE tolerates Byzantine faults in user domains by verifying the output of execution replicas: In order for a reply, state update, or state copy to be treated as correct, a total of at least $f + 1$ execution replicas must have provided the same value. As discussed in Section 2.1.1.2, fault independence of execution replicas (i.e., the absence of correlated faults [23, 40, 148]) is an important property in this context: If a single fault, for example, leads to more than f execution replicas returning the same faulty reply, the system would not be able to uphold its safety guarantees. In the following, we elaborate on the extent to which fault independence of execution replicas is required in SPARE. Furthermore, we discuss the integration of heterogeneous replica implementations in the context of N-version programming [12, 13, 38] aimed at improving fault independence.

4.6.1 Eliminating Harmful Correlated Failures of Execution Replicas

Below, we analyze particular characteristics of SPARE with regard to fault independence of execution replicas and propose measures to reduce the probability of correlated faults.

Analysis of SPARE-specific Characteristics Although perfect fault independence of execution replicas already is notoriously difficult to achieve in practice [94], there are fundamental reasons limiting the spectrum of faults with regard to which execution replicas in SPARE can be independent: First, being designed for application in a single data center (see Section 4.2.1), a SPARE cell is not resilient against disasters that affect the entire site (e.g., earthquakes); adapting the system to be run in a geo-replicated environment is possible but outside the scope of this thesis (see Section 7.3). Second, an active execution replica and its co-located passive execution replica can never be fully fault independent due to being hosted on the same physical server: If the server crashes both execution replicas become unavailable.
Note that, even though both fault scenarios mentioned above result in correlated failures of execution replicas, neither of them poses a threat to the safety of SPARE: In the first case, the crash of all execution replicas prevents the system from making any progress at all; however, it does not cause SPARE to treat faulty replies, state updates, or state copies as correct. Regarding the second case, SPARE reacts to the crash of a server by executing a custom fault-handling procedure that ensures the continuous availability of the service, as discussed in Section 4.4.4.
Taking these factors into account, SPARE is able to uphold its safety guarantees as long as all of the following requirements are met: First, crashes of servers belonging to the same cell are not correlated while the data center is operational. Second, execution replicas hosted on the same server fail independently while the server is running. Third, failures of execution replicas hosted on different servers are never correlated.

Improving the Fault Independence of Execution Replicas In order to ensure that the requirements for fault-independent execution replicas presented above are met with high probability, we propose the following measures: First, as discussed in Section 4.2.1, the

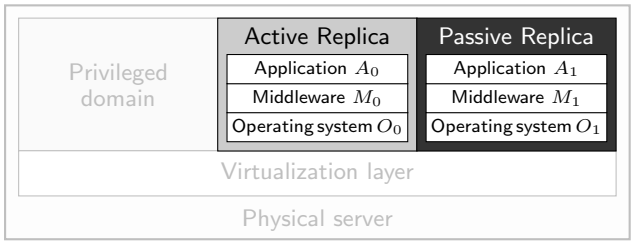


Figure 4.9: Example for introducing diversity into execution replicas: Although all providing the same service, different execution replicas in SPARE may rely on different operating systems, middleware components, and service implementations in order to improve fault independence.

servers of a SPARE cell should be placed in different racks with redundant networking and power [32]; this way, in case of a partial power outage, for example, servers crash independently. Second, on each server, the states of execution replicas should be stored on different storage media in order for active and passive execution replica to be independent with regard to disk failures. Third, as discussed in Section 4.6.2 in more detail, heterogeneous implementations should be used to minimize the probability of execution replicas sharing the same vulnerabilities [12, 13, 38, 71].

4.6.2 Utilizing Heterogeneous Execution-replica Implementations

Heterogeneous implementations are a means to reduce the probability of correlated faults in execution replicas, as discussed in Section 2.1.1.2. In the following, we present the extensions required to apply N-version programming to execution replicas in SPARE.

Introducing Diversity Due to the fact that execution replicas in SPARE not only comprise an instance of the service application but instead span an entire user domain (see Section 4.2.4), diversity can be introduced on multiple levels: Besides being equipped with its own service implementation, each execution replica may rely on a different operating system and middleware (see Figure 4.9). Note that, to minimize the costs for development and integration, generic components on all levels can be diversified by relying on already existing software (e.g., web servers or databases), following the concept of opportunistic N-version programming [35].

The Need for a Uniform State Representation Integrating existing software components often introduces an additional problem: Although all providing the same functionality, different implementations are likely to use different representations to manage the service state. As a result, state updates put out by two heterogeneous execution replicas might not match even though both execution replicas operate correctly, leading update verifications to stall indefinitely. A similar problem can occur during a proactive-recovery procedure if formats differ for state copies that essentially comprise the same contents.
One solution to implement verification in the face of such problems could be to make voters aware of different data formats. However, as this would require application-specific

code to be introduced into the replica manager, this approach may introduce vulnerabilities in the privileged domain. Furthermore, it does not work in cases where voters verify the correctness of a value based on hashes (e.g., during recovery, see Section 4.5.2).
In consequence, SPARE addresses the issue of diverse data formats by requiring execution replicas to include conversion routines that allow them to translate data between their replica-specific representation and a uniform representation that is identical across all execution replicas [35]. Note that SPARE makes no assumptions on the format of such a uniform state representation as long as it allows the system to perform a byte-by-byte comparison of messages from different execution replicas.

4.7 Safety and Liveness

In this section, we discuss why SPARE remains safe and live (see Section 2.1.1.3) as long as the number of faulty execution replicas does not exceed the threshold a particular cell is dimensioned for. To guarantee safety, the system has to ensure that faults in some components do not propagate to other, non-faulty system parts; in addition, SPARE protects clients by only returning results that have been verified based on the replies provided by different execution replicas. To guarantee liveness, the system must always keep enough passive execution replicas available to be able to make progress in the event of verification processes for results and/or updates stalling.

4.7.1 Containment of Faults

As other virtualization-based systems [28, 45, 130], SPARE relies on the virtual-machine monitor to enforce isolation between virtual machines running on the same physical server. In particular, this includes the virtual-machine monitor preventing an active execution replica from corrupting its co-located passive execution replica or the replica manager executed in the privileged domain by breaking out of its virtual machine. Nevertheless, a malicious execution replica might still try to cause damage by emitting faulty replies and/or state updates. In order to also contain faults in such cases, SPARE takes a number of measures discussed in the following to protect both replica managers as well as other execution replicas.

Protection of Replica Managers Replica managers in SPARE never interpret, let alone process, any requests, replies, or state updates provided by clients and execution replicas. Being an application-independent component, a replica manager treats all messages exchanged between clients and the application as chunks of bytes. With respect to the verification of results and state updates, this means that verification is realized based on a byte-by-byte comparison of messages (in the uniform representation, see Section 4.6.2), without performing any checks on semantic equivalence. As a consequence, replica managers are protected from vulnerabilities in message deserialization routines and higher-level application-specific procedures. Besides verification, as discussed in Section 4.8.1, the only operation replica managers may invoke on replies and state updates is the computation of hashes. Here, too, messages are not interpreted but remain in their serialized form throughout the process.

Protection of Execution Replicas With the virtual-machine monitor enforcing isolation between virtual machines, the only possibility for an execution replica to interact with another one is through sending state updates. In order to prevent a faulty active execution replica from corrupting the application state of a passive execution replica this way, all state updates forwarded to passive execution replicas are first verified by a replica manager comparing the updates from different active execution replicas. During this process, faulty state updates will be sorted out due to not receiving the majority of votes required for becoming stable (i.e., $f + 1$). After a successful verification, passive execution replicas may safely deserialize and interpret state updates in order to apply them. Although verification allows SPARE to protect execution replicas from being corrupted via state updates, a fault may still occur as the result of processing a client request. As discussed in Section 4.6, the measures taken to achieve fault-independent execution replicas are targeted at confining such a fault to a single execution replica.

4.7.2 Ensuring System Progress

Verifying results and state updates is an effective measure to guarantee safety in a fault-tolerant system. However, in order to be of practical use, such systems must also ensure that progress is made, both in the absence as well as the presence of faults, as long as there are clients invoking operations on the service. Systems relying on traditional active replication [34, 41, 49, 130, 161] provide this property by constantly keeping more execution replica active than necessary during normal-case operation. In contrast, SPARE minimizes the number of active execution replicas in the absence of faults, requiring passive execution replicas to assist in fault handling. In the following, we discuss liveness aspects that arise from this design decision.

Overcoming Stalled Result Verifications Having sent a request to a service replicated using SPARE, a client expects to receive a verified result in return. However, in case of faults, the replica manager responsible for proving the result correct at first might not receive enough matching replies from different active execution replicas in order to successfully complete the verification process right away. For a SPARE cell dimensioned to tolerate at most f faults, ensuring liveness means to guarantee that even in such cases eventually $f + 1$ matching replies from different execution replicas become available. With a cell comprising a total of $2f + 2$ execution replicas ($f + 1$ of them initially active, the other $f+1$ initially passive), a set of $f+2$ non-faulty execution replicas (i.e., one more than actually required to make progress) remains if f replicas crash or are otherwise subject to faults.

Note that in order for SPARE to overcome a stalled result verification, it is not enough to guarantee the availability of a certain number of passive execution replicas. In addition, it is also crucial to ensure that all of those execution replicas are actually capable to participate in fault-handling procedures. As we do not force execution replicas to provide a roll-back mechanism allowing the recreation of past application states, this requirement demands in particular that passive execution replicas must not already have applied the update corresponding to the result whose verification has stalled. In other words, in

order to assist in fault-handling procedures for a request o, a passive execution replica at this point must not have performed the state modifications triggered by processing request o. In SPARE, this is ensured by including replies in their corresponding UPDATE messages (see Section 4.3.2). This way, an UPDATE cannot become stable without result verification for the same request also completing successfully. As a consequence, if result verification for request o stalls due to a fault, it is guaranteed that none of the passive execution replicas has already applied the state update for request o as the update could not have been verified successfully; therefore, all passive execution replicas are eligible to participate in fault handling.

Overcoming Stalled Update Verifications As explained in Section 4.3.2, a state update in SPARE is verified by each replica manager independently, based on the updates provided by different active execution replicas. On the one hand, voting separately offers the benefit of not having to notify other replica managers about the outcome of the verification process. On the other hand, this approach might temporarily lead to different replica managers drawing different conclusions on verification progress: In case of delays, for example, some replica managers might be able to successfully complete verification, while others might stall due to not receiving enough replies before their timeouts trigger (see Section 4.4.3). As a result of the subsequent fault-handling procedures, one or more passive execution replicas may be (unnecessarily) activated; however, all replica managers in the cell will be able to verify the state update eventually, either due to the original messages finally being processed or thanks to the additional updates becoming available during fault handling.

4.8 Optimizations and Tradeoffs

In this section, we present optimizations that allow SPARE to further reduce its resource footprint and/or increase performance. Furthermore, we discuss important tradeoffs with regard to the design of the system's fault-handling mechanisms.

4.8.1 Use of Hashes

As in other systems (see Section 2.1.1.4), a voter in SPARE does not require $f + 1$ full replies or state updates to perform a successful verification; instead, a single full version of the correct value is sufficient as long as the voter is able to obtain at least f correct reply/update hashes in addition. SPARE exploits this fact to reduce the network bandwidth consumption of the communication between replica managers: For large replies and state updates, only one replica manager provides a full version, while all others include hashes in their REPLY and UPDATE messages. If possible, the replica manager that contributes the full reply/update is the one that also performs the verification, as in such case the large message does not have to be transmitted over the network.
Note that the use of reply and state-update hashes introduces an additional problem to be addressed: With voting being performed based on hashes, a successful verification does

not automatically mean that a replica manager also has the correct full reply/update available; instead, it might have only learned the correct hash. Such a scenario occurs, for example, if the full version of a reply or state update obtained by a replica manager is faulty. To solve this problem, replica managers that have only contributed a hash must be prepared to provide the corresponding full replies/updates on demand.

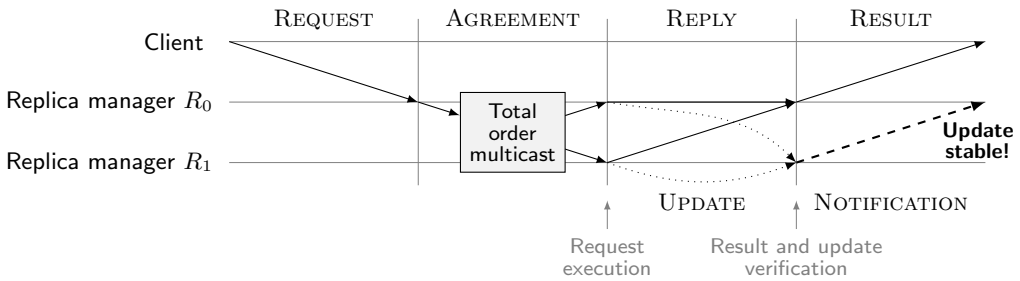
4.8.2 Batching

Section 4.3.2 has already presented batching as a possible way to minimize the overhead of bringing passive execution replicas up to speed in SPARE. In this context, the size of the buffer, in which verified state updates are stored before being collectively forwarded to the execution replica, has an impact on the duration of fault-handling procedures: Using a smaller buffer size, less state updates have to be applied during fault handling before a passive execution replica is able to process the request for which verification has stalled. As a result, it takes less time until the replies/updates become available that are necessary to decide the vote. On the downside, a smaller buffer size leads to shorter update cycles of passive execution replicas causing additional resource overhead due to the corresponding virtual machines being woken up more frequently. In summary, the buffer size allows one to trade off fault-handling latency for resource efficiency.
Besides optimizing the updating procedure of passive replicas, batching can also be used to reduce the overhead for both agreement and verification in SPARE: Instead of inserting each request into the agreement stage independently, a replica manager may collect requests from different clients and hand them over in a batch (see Section 2.1.2.2). With the agreement stage treating the batch as one large message, multiple requests are ordered within a single round of agreement. Note that a similar approach can also improve update verification: In this case, a replica manager may batch the state updates it receives from a local execution replica before distributing them to other replica managers in order to be verified.

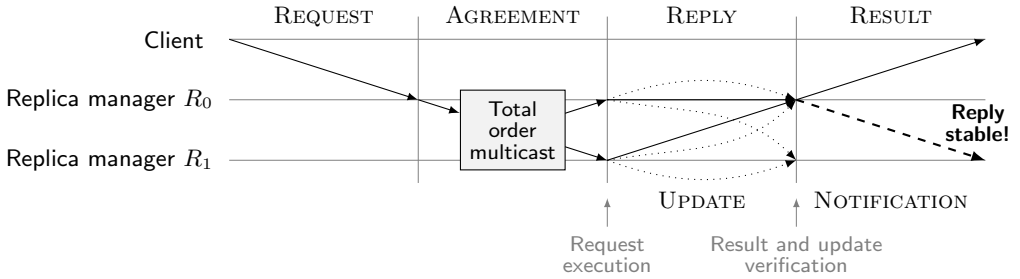
4.8.3 Update Verification

In the following, we discuss alternative solutions to the problem of verifying the correctness of a state update and compare them to the approach used in SPARE.

Independent Update Verifications As described in Section 4.3.2, state updates in SPARE are verified by each replica manager independently. Note that this approach trades off latency for bandwidth: With replica managers performing redundant update verifications, there is no additional communication step required for a replica manager to learn the outcome of the voting process. However, this comes at the cost of having to distribute the state updates of all execution replicas between all replica managers. As discussed in Sections 4.8.1 and 4.8.2, the overhead for sharing such information can be significantly reduced in practice by a combined use of hashes and state-update batching.



(a) Protocol variant performing update verification on only a single replica manager.



(b) Protocol variant not requiring a state update to comprise the hash of its corresponding reply.

Figure 4.10: Message flow of requests and replies (—), state updates (····), and notifications (- -) for variants of the protocol presented in Figure 4.4: At the cost of some replica managers having to wait an additional communication step to make progress these alternatives (a) reduce verification overhead and (b) allow replica managers to handle replies and state updates separately.

An alternative solution to address the problem of verifying state updates could have been to perform voting for each state update on a single replica manager only, which would then notify all other replica managers about the outcome, as shown in Figure 4.10a. To balance load, the replica manager responsible for verifying a particular update might, for example, be selected in a round-robin fashion. Besides minimizing the number of messages to be sent, such an approach would also prevent scenarios in which some replica managers successfully complete the verification process for a state update while others might stall temporarily (see Section 4.7.2). On the downside, the additional interaction necessary to propagate the verification result may cause a prolonged fault handling. Furthermore, a failover mechanism would be required to ensure progress in case the replica manager selected to perform verification crashes during the process.

Inclusion of Replies in State Updates Another design decision with respect to update verification was to include replies (or in the optimized case: their hashes) in the corresponding state updates (see Section 4.3.2). This way, state updates can only be successfully verified if the result of the operation that led to the state modification also has become stable. In consequence, during the updating process, no extra measures have to be taken to fulfill the requirement discussed in Section 4.7 demanding that an update is only applied to a passive execution replica when it is guaranteed that the replica will not be needed to participate in fault handling for the corresponding request.

An alternative solution could have been to not include the reply (hash) in the state update but instead instruct the replica manager performing reply verification to distribute a notification informing all other replica managers about the result becoming stable (see Figure 4.10b). Having received such a notification it would be safe for a replica manager to apply the update to its local passive execution replica. Note that such an approach may enable more efficient implementations for use-case scenarios in which execution replicas provide replies and state updates through different channels (see Section 4.10.2 for an example), as it allows replica managers to handle replies and updates independently.

4.9 Integration with Existing Infrastructures and Services

In this section, we describe how the concepts of SPARE can be realized using Xen [16], a state-of-the-art virtualization technology widely used in today's data centers and cloud infrastructures (e.g., Amazon EC2 [7]). Furthermore, we present a set of implementation details that are crucial for the integration of existing network-based services in practice.

4.9.1 Xen

The SPARE architecture imposes a number of requirements on the virtualization technology in use, for example, with regard to the privileges of different virtual machines or the flexibility of network configurations. In the following, we discuss how those requirements match with features provided by the Xen virtual-machine monitor [16]. Note that, due to relying on basic virtual-machine–monitor functionality and not using any Xen-specific features, other virtualization technologies (e.g., VMware [155]) could also serve as basis for a SPARE implementation.

4.9.1.1 Domains

The SPARE server architecture distinguishes between a privileged domain hosting application-independent system parts (i.e., the replica manager) and user domains comprising instances of the service application (see Section 4.2.4). In contrast to execution replicas in user domains, the replica manager in the privileged domain is aware of running in a virtualized environment. Not only that, the replica manager also requires privileges to modify its environment: Amongst other things, this includes the activation of passive execution replicas during fault handling (see Section 4.4.2) as well as the creation of new user domains in the context of proactive recovery (see Section 4.5).
Xen addresses these differences by providing two categories of domains: a privileged *Domain 0*, which is implicitly started at the boot time of the physical server, and non-privileged *DomU*s, which must be explicitly managed by the Domain 0. As desirable for fault-independent execution replicas (see Section 4.6), each DomU may comprise its own operating system, middleware, and application instance. Furthermore, due to SPARE not imposing any restrictions, DomUs may be either paravirtualized or fully virtualized, widening the range of operating systems to be deployed in user domains.

4.9.1.2 Network

As discussed in Section 4.2.1, the SPARE architecture relies on three different types of networks: a public network through which clients and servers exchange requests and results, a private network used for communication between replica managers, and an internal network on each physical host that allows the privileged domain to interact with user domains. In order to guarantee the safety properties of SPARE, the networks have to be isolated from each other; for example, clients must not have direct access to user domains and execution replicas running in user domains must not be able to send messages to replica managers residing on other physical servers.
Using Xen, this isolation can be implemented by relying on multiple *software bridges* managed in the privileged domain [26, 157, 160]; a software bridge acts as a switch for different physical or virtual network devices. For the public and private network, separate bridges should be used, each connected to its own physical network device. Furthermore, one has to ensure that only the virtual network device of the privileged domain is added to those bridges as, with the exception of the replica manager, no other component should have direct access to physical network devices. For setting up the internal network, a third software bridge may be used which connects the virtual devices of both privileged domain and user domains; that is, such a bridge does not include any physical network devices.

4.9.1.3 Resource-saving Modes for User Domains

As discussed in Section 4.3.2, SPARE reduces the resource usage of passive execution replicas by keeping their corresponding user domains in resource-saving mode. Xen offers two different mechanisms that can be used for this purpose [39]: *pause/unpause* and *suspend/resume*. Pausing a user domain in execution causes it to lose its status of being runnable and consequently leads to the virtual-machine monitor not scheduling the domain from this point on; as a result, a paused user domain still resides in memory but does not consume any CPU. In contrast, the suspend/resume mechanism goes a step further by releasing the memory allocated by a user domain after having stored the domain's current state on disk; therefore, a suspended user domain neither consumes CPU nor memory. In conclusion, if resource usage were the only concern, Xen's suspend/resume mechanism would be the means of choice to implement the resource-saving mode of passive execution replicas.
However, besides minimizing the resource footprint of a user domain, SPARE imposes another important requirement on a resource-saving mechanism: The transition between resource-saving mode and normal-case operation must be fast. Note that this property is especially crucial during fault handling, for example, in the event of a stalled result verification: The longer it takes a passive execution replica to leave resource-saving mode, the later it is able to process the request in question in order to provide an additional reply. Analyzing the pause/unpause and suspend/resume mechanisms in Xen, we found significant differences in latencies: While unpausing a user domain in our test environment (see Section 4.11.1) requires only about 210 milliseconds, resuming a suspended

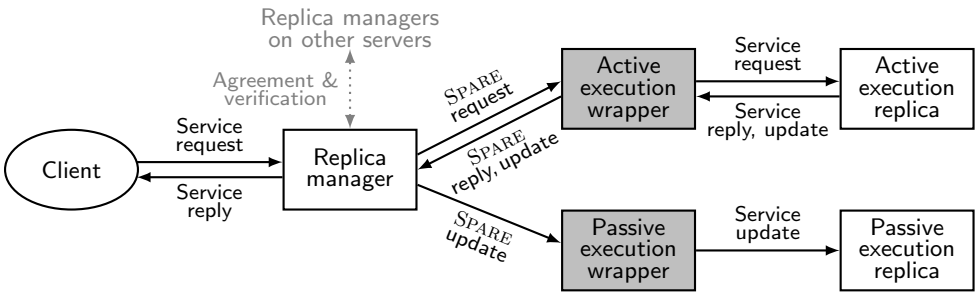


Figure 4.11: Introduction of execution wrappers to manage the interaction between a replica manager and both execution replicas: Depending on design considerations, an execution wrapper can either be integrated with the replica manager in the privileged domain or running along side its execution replica in the user domain.

user domain takes more than 27 seconds; that is, a factor of more than two orders of magnitude longer. Given these numbers, using the suspend/resume mechanism in SPARE is not feasible as it would lead to fault-handling latencies of almost half a minute. In our current prototype, we therefore rely on paused user domains as basis for the resource-saving mode of a passive execution replica. Nevertheless, we do not want to rule out that this decision might change in the future as evolution in software and/or hardware, might speed up the process of resuming a suspended user domain; for example, the use of solid-state drives instead of hard drives is likely to decrease the overhead for making the user-domain state persistent. Furthermore, customizing the suspend/resume mechanism to the particular requirements of SPARE could also brings additional benefits.

4.9.2 Integration of Service Applications

Many existing network-based applications have not been developed to be run in a replicated environment. In consequence, integrating such services with SPARE inherently requires modifications to their original implementations. This section presents system components and mechanisms of SPARE designed to facilitate this adaptation process.

4.9.2.1 Execution Wrapper

Although network-based services in general use message passing to interact with clients, different applications rely on different communication patterns and/or require different methods of authorization before a client is actually allowed to use the service. As a result, in order to be as transparent to execution replicas as possible, the SPARE infrastructure needs to properly imitate the behavior of clients of the particular application. To address this problem, we extend the approach proposed in [35] and introduce a system component, the *execution wrapper*, whose main responsibility is to manage the interaction between replica managers and their execution replicas. As shown in Figure 4.11, each execution replica is assigned with its own execution wrapper.

Basic Architecture An execution wrapper in SPARE consists of two parts: a generic part, which handles communication with the replica manager, and an application-specific part, which knows how to interact with the execution replica. The generic part of an execution wrapper treats all application-related messages (i.e., requests, replies, and state updates) as chunks of bytes, without interpreting, processing, or modifying them. It communicates with the replica manager using SPARE-internal messages that, besides containing the original requests, replies, or state updates, also comprise relevant meta data (e.g., the id of the execution replica that provided a reply, or the sequence number of the client request a state update corresponds to). In contrast, the application-specific part of an execution wrapper is allowed to append or even rewrite service messages; such modifications, for example, may include the handover of deterministic timestamps (see Section 4.9.2.2) as well as data-format conversions necessary to support heterogeneous execution-replica implementations (see Section 4.6.2).

Implementation Alternatives Besides identifying the application-specific tasks required from an execution wrapper, another important decision has to be made during the adaptation process of a service application: where to place the execution wrapper. As integrating the execution wrapper with the service implementation can be considered too expensive for most applications, there are basically two different possibilities remaining to introduce an execution wrapper into the SPARE architecture: First, by integrating it with the replica manager in the privileged domain or, second, by co-locating the execution wrapper with the execution replica in the user domain.
Combining both replica manager and execution wrapper in a single component, on the one hand, offers the advantage of achieving improved latency thanks to not requiring an additional indirection. On the other hand, due to including the execution of service-specific code (i.e., the application-specific part of the execution wrapper) in the privileged domain, such an approach is only suitable for execution wrappers that do not interpret service messages. Otherwise, the execution wrapper would represent a potential vulnerability endangering the containment of faults (see Section 4.7.1). Note that there are no such restrictions when co-locating the execution wrapper with the execution replica in the user domain: In this case, the virtual-machine monitor ensures isolation regardless of whether a fault occurs in the service application or in the execution wrapper. However, this flexibility comes at the cost of an additional communication step between replica manager and execution wrapper.

4.9.2.2 Support for Consistent Timestamps

As discussed in Section 2.1.2.3, active replication requires execution replicas to implement the same deterministic state machine. In practice, many implementations of service applications violate this principle (see Section 4.10.2.1 for an example) by relying on timestamps generated by the local system clock. Due to the fact that it is neither guaranteed that the system clocks of different user domains are synchronized nor that all execution replicas in the cell will process a particular request at the same point in (physical) time, the use of different timestamps could lead to inconsistencies, possibly causing the states of non-faulty execution replicas to diverge.

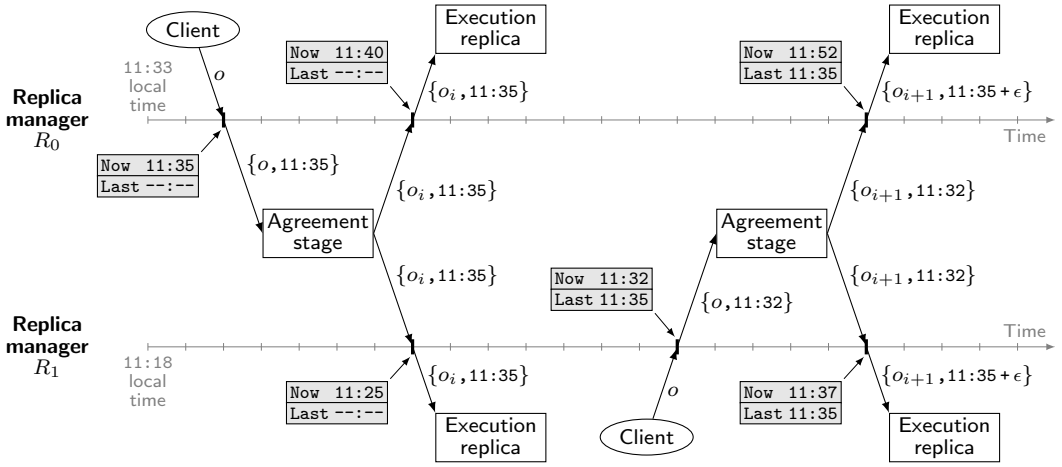


Figure 4.12: Overview of SPARE's mechanism for providing deterministic timestamps: Before introducing a client request o into the agreement stage, a replica manager attaches a timestamp reflecting its current local time. As local clocks may differ between servers, to ensure monotonicity, replica managers check (and if necessary adjust, see request o_{i+1}) the timestamp of an ordered request prior to forwarding it to the execution replica.

In order to ensure deterministic timestamps, we adapt the standard technique proposed to address this problem in Byzantine fault-tolerant systems [34, 35, 161]: The approach utilizes the agreement stage to distribute a timestamp attached to each request, which may then be used by all execution replicas. As shown in Figure 4.12, in SPARE, before inserting a request into the agreement stage, a replica manager attaches a timestamp reflecting the current state of its local system clock (represented by the `Now` value in Figure 4.12) to the message; this way, all replica managers in the cell will receive both the request as well as the timestamp associated. However, due to the lack of perfectly synchronized system clocks, this technique alone does not guarantee monotonically increasing timestamps. Therefore, each replica manager locally stores the timestamp t_i assigned to the latest request ordered o_i (represented by the `Last` value in Figure 4.12) and compares it to the timestamp t_{i+1} of the subsequent request o_{i+1} in the output of the agreement stage. If $t_i < t_{i+1}$, a replica manager stores t_{i+1} and forwards request o_{i+1} to the local active execution replica. However, if $t_i \geq t_{i+1}$, a replica manager assigns a new timestamp $t_x = t_i + \epsilon$ to request o_{i+1} in order to ensure monotonicity; the replica manager then stores t_x and hands request o_{i+1} over to the execution replica. With this algorithm being deterministic, all non-faulty replica managers in the SPARE cell adjust timestamps in a consistent manner.

Note that a similar approach can be applied to deal with other determinism-related problems, for example, the consistent use of random numbers. In all such cases, replica managers can decide on deterministic values (e.g., timestamps) and attach them to client requests. However, it lies within the responsibility of execution wrappers (see Section 4.9.2.1) to ensure that execution replicas actually make use of these values.

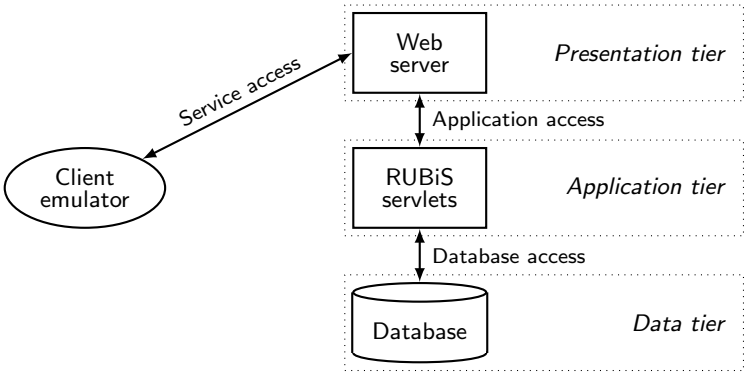


Figure 4.13: Basic architecture of RUBiS: Client requests (e.g., searches for items) are received by a web server and processed by a set of servlets, potentially leading to read or write operations being issued on a relational database storing the application state.

4.10 Case Study: RUBiS

In this section, by means of the *Rice University Bidding System (RUBiS)* [120], we investigate how to integrate a multi-tier application (i.e., a typical use-case example for SPARE) with our system. RUBiS is a benchmark for middleware infrastructures designed to emulate an eBay-like [62] auction system; as basis for our prototype, we use the RUBiS version relying on Java servlets. In the following, we present an overview of the application's architecture and discuss the modifications necessary to integrate it with SPARE.

4.10.1 Overview

The RUBiS benchmark is composed of two main parts: First, the server side provides the core functionality of an auction system (i.e., creating auctions, browsing for items, placing bids, uploading comments etc.) via a website interface. Second, on the client side, a benchmarking tool emulates the behavior of human clients accessing the auction system's website using a browser.

Auction System Figure 4.13 shows an overview of the multi-tier architecture of the RUBiS auction system located on the server side: Client browsers establish connections to the upper tier and send their requests to a Jetty web server, where they are then processed: Queries accessing static content (e.g., a retrieval of the cover page of the auction system's website) are answered directly by the web server, without the involvement of another tier. In contrast, requests to dynamic content (e.g., a search for items currently available) are handled by Java servlets responsible for generating a custom reply whose content depends on the application's current state. As the state of the auction system is managed in a MySQL database in the lower tier, in order to process a request, servlets issue one or more SQL statements to query and/or update information about users, items, auctions, or comments.

Client Emulator The RUBiS client emulator is a benchmarking tool designed to conduct automated experiments on the auction system. In order to evaluate realistic workload patterns, the tool emulates the behavior of a human client interacting with the auction system through a browser; that is, the client emulator takes into account that, besides using the functionality provided by the system (e.g., to search for items or to place bids), a client also takes time to read the information retrieved. In consequence, instead of immediately issuing a subsequent request after having received a reply, the tool includes intervals during which a client does not use the auction service at all. To vary the load on the system, the client emulator allows one to conduct experiments with multiple clients in parallel, all running their own sessions. During their sessions, clients do not execute the same predefined sequence of steps. Instead, the client emulator utilizes a probabilistic approach to determine the next interaction to initiate for each client individually.

4.10.2 Integration with SPARE

In the following, we present a possible way to integrate RUBiS and similar multi-tier applications with SPARE in order to allow them to benefit from the long-term dependability provided by the system. Note that the approach discussed below requires the introduction of a RUBiS-specific execution wrapper (see Section 4.9.2.1) as well as changes to the auction system; in contrast, the client emulator can remain unmodified.

4.10.2.1 Deterministic Execution Replicas

Active state-machine replication, in general, requires execution replicas to be deterministic, as discussed in Section 2.1.2.3. Analyzing the components comprising the RUBiS server side, we found that the original implementations of some of the Java servlets do not satisfy this property as they all rely on timestamps generated by the local system clock. In consequence, when a request is processed by multiple execution replicas, each of them might use a different timestamp, for example, to select the deadline of an auction. However, such inconsistencies must not happen as they can lead to non-faulty execution replicas behaving differently: Continuing the example, one replica might conclude that a winning bid has been received before the deadline of an auction, while another replica might announce a different client to be the winner of the same auction due to having selected an earlier deadline.
To ensure consistent behavior of non-faulty RUBiS execution replicas, we modify the affected servlets to make use of the deterministic timestamps provided by SPARE (see Section 4.9.2.2). Servlets receive the timestamp as part of the HTTP header of a client request to which the RUBiS execution wrapper has appended an additional line containing the timestamp for the request.

4.10.2.2 Creation of State Updates

The RUBiS benchmark has not been designed with service replication in mind. As a result, the auction system lacks specific functionality to extract its current application state. However, implementing the concept of a multi-tier architecture, RUBiS stores all relevant application state in the lower tier (i.e., the database).

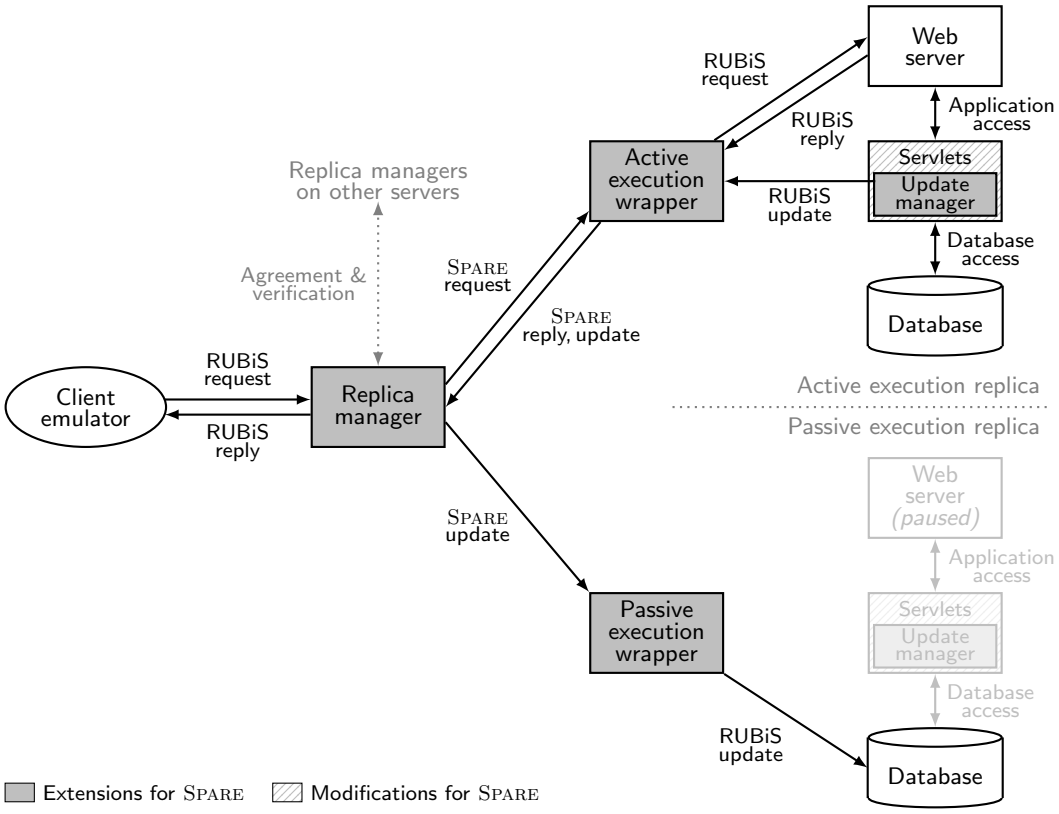


Figure 4.14: Overview of the RUBiS integration into SPARE: Interaction with both execution replicas is handled by application-specific execution wrappers; only service-level requests and replies are exchanged via the presentation tier (i.e., the web server), while state updates are directly extracted from/applied to lower tiers.

Keeping Track of State Modifications To enable active RUBiS execution replicas in SPARE to provide state updates, we introduce a module between upper and lower tier, the *update manager*, which is able to keep track of state modifications. As depicted in Figure 4.14, the update manager intercepts all calls from the web server to the database. Depending on the command of the SQL statement, the update manager distinguishes between two different categories of calls: For read-only queries (e.g., `SELECT` statements), the update manager acts as a relay; that is, it issues the call to the database and returns the corresponding result to the web server without further action. Similar to other network-based applications (see Section 4.12.3), the majority of the workload in RUBiS consists of such queries. However, there is also a small fraction of calls leading to state changes in the database (e.g., `INSERT` or `UPDATE` statements), for which the update manager needs to take additional measures in order to generate state updates.

Generating Updates for State Modifications Upon intercepting a state-modifying call, the update manager performs the following steps: First, it forwards the call to the lower tier and waits until the database provides a result. Next, based on the SQL statement of the original call, the update manager creates and executes a database query which reads the entries inserted/updated by the state-modifying call. This way, the update manager learns relevant changes to the database and is therefore able to create a state update (i.e., an `INSERT` or `UPDATE` statement) which will reproduce the modifications when being applied to the database of a passive execution replica. Finally, the update manager returns the reply of the original call to the web server in the upper tier.
Note that, instead of issuing an additional query for each state-modifying call, an alternative (and more efficient) solution to create state updates in RUBiS might have been to derive them directly from the SQL statements of the calls intercepted. We decided not to pursue this approach as it comes with a major drawback: If the update manager created a state update based on the original state-modifying call, it would skip an essential part of processing; that is, the execution of the call by the database. As a result, in case of a fault during this last step of execution, an update would not reflect the actual state modification but a change the execution replica never went through. Instead, by retrieving the effects of a call after its execution has fully completed, our approach ensures that state updates only comprise changes that have manifested in the database.
Creating and performing the read-after-modification query for state updates in RUBiS is straight-forward and efficient, as state-modifying calls to the database do not involve complex operations spanning multiple tables but only modify at most two tables each; for example, in the lower tier, modifications for inserting an item are limited to a table storing information about all auctions, whereas registering a new user just leads to an additional entry in the users table. Furthermore, most of the calls only update a single database row or field which greatly facilitates the task of determining their effects using a separate query. Note that, for applications for which such an approach is not feasible, more sophisticated techniques [134, 135] can be used to extract write sets.

4.10.2.3 Transmission of State Updates

Having created an update reflecting a modification to the auction system's state, the update manager of an active execution replica needs to hand it over to the local RUBiS execution wrapper. One way to solve this problem would have been to attach the update to the result of the database call, thereby returning it to the web server in the upper tier, which in turn would have to be responsible for propagating the update; for example, as part of its reply (which is completely different from the database result forwarded by the update manager) to the client. However, as such an approach would have required major refactoring, we developed an alternative solution.
Instead of taking the detour via the web server, an update manager in our system directly transmits state updates to its local execution wrapper using a dedicated network connection between them (see Figure 4.14). Note that such an approach, on the one hand, achieves a clear separation of concerns, but, on the other hand, introduces the following problem: With replies to client requests and state updates being delivered independently,

Original State-update Batch

```
U1 INSERT INTO comments VALUES
       ('[ID of user A]', '[Comment C_A]');
U2 INSERT INTO comments VALUES
       ('[ID of user B]', '[Comment C_B]');
U3 UPDATE items
       SET quantity = [Quantity]
       WHERE id = [ID of item X];
U4 UPDATE users
       SET rating = [Rating]
       WHERE id = [ID of user C];
U5 INSERT INTO comments VALUES
       ('[ID of user D]', '[Comment C_D]');
```

Optimized State-update Batch

```
O1 INSERT INTO comments VALUES
       ('[ID of user A]', '[Comment C_A]'),
       ('[ID of user B]', '[Comment C_B]'),
       ('[ID of user D]', '[Comment C_D]');
O2 UPDATE items
       SET quantity = [Quantity]
       WHERE id = [ID of item X];
O3 UPDATE users
       SET rating = [Rating]
       WHERE id = [ID of user C];
```

Figure 4.15: Comparison of an original and an optimized state-update batch in RUBiS (simplified example): SQL statements may be combined (e.g., to use only a single call O1 to insert multiple data rows U1, U2, and U5) or reordered (e.g., the insert U5 with respect to the update U3) if they affect different parts of the application state.

the execution wrapper needs to be provided with information on which state update corresponds to which request; this mapping, for example, is crucial to determine which state updates to compare during verification. To address this problem, we extend the execution wrapper to add a unique id (i.e., the agreement sequence number) to the HTTP header of each client request before issuing the request to the active execution replica. Inside the execution replica, we enable the update manager to access this information during the execution of the request. This way, the update manager is able to assign the unique id to the corresponding state update, which in turn allows the execution wrapper to resolve the mapping between requests and updates.

4.10.2.4 Update of Passive Execution Replicas

During the process of updating a passive execution replica (see Section 4.3.2), a replica manager flushes the content of its buffer and transfers a batch of verified state updates (i.e., a list of SQL statements) to the RUBiS execution wrapper. In order to reproduce the state modifications reflected in the batch, the execution wrapper eventually forwards it to the local update manager. However, before actually handing it over, the execution wrapper performs a number of optimizations on the batch (see Figure 4.15) to speed up the updating process. In particular, optimizations include the combination of a set of insertions to a single statement that on execution adds multiple entries to the database; using only one statement is significantly more efficient than issuing a call for each entry to be inserted [27]. Note that during the optimization of a batch, SQL statements may be reordered without losing correctness, if they are *independent* [97], for example, due to operating on separate parts of the database (e.g., different tables, as depicted in Figure 4.15).

The modifications performed to optimize a batch of state updates in RUBiS are not specific to this application but can be applied to all batches consisting of SQL statements. In general, there are two major aspects to take into consideration: First, the overhead of optimizing a batch should correspond to the time saved by using the optimized version instead of the original one. Second, in order to ensure correctness, it is crucial that the resulting optimized batch leads to the exact same database state the original batch would have produced if it had been executed.

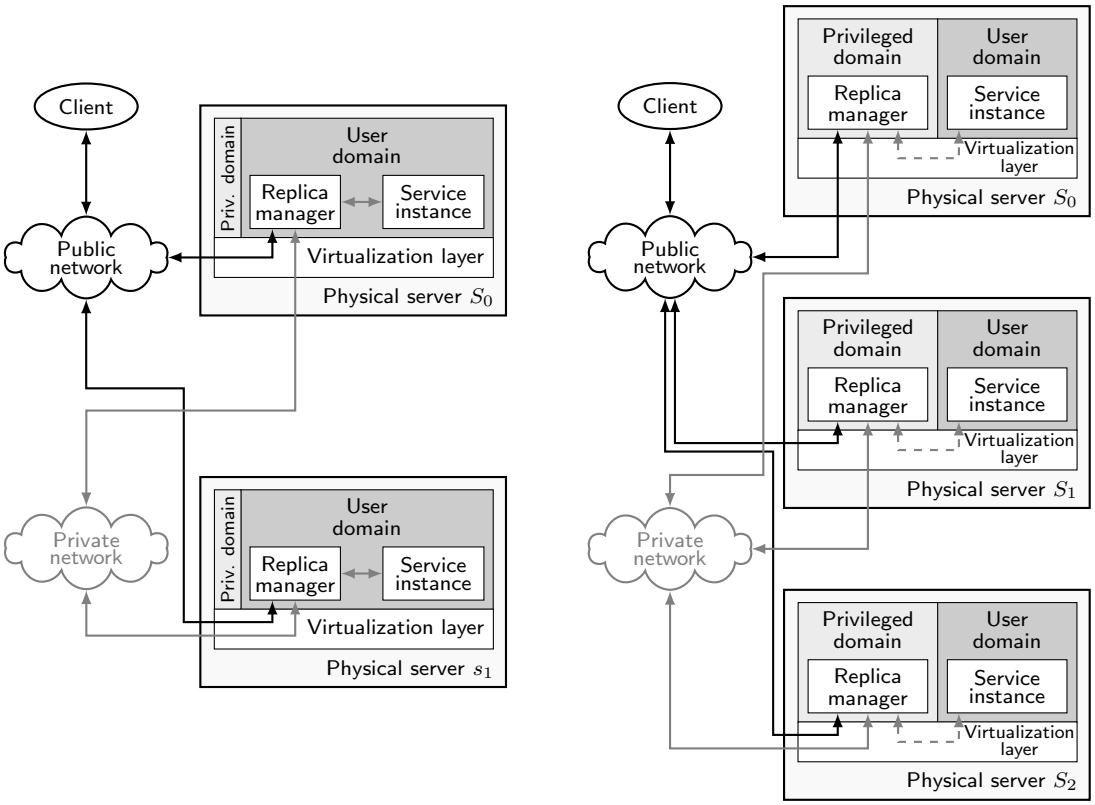
4.11 Evaluation

In the following, we use the RUBiS benchmark presented in Section 4.10 to evaluate both the performance and resource footprint of SPARE. For comparison, we also perform experiments with two related system configurations: one that can only handle crashes of execution replicas, and another that is able to tolerate Byzantine faults in execution replicas but, unlike SPARE, relies completely on active replication.

4.11.1 Environment

We perform our experiments on a cluster of 8-core servers (2.3 GHz, 8 GB RAM) which are all connected with switched Gigabit Ethernet. Each machine is running a Xen 4.0.1 virtual-machine monitor with a Ubuntu 10.04 (2.6.32-18 kernel) privileged domain; for user domains, Debian 7.0 (2.6.39.4 kernel) is used as operating system. In our prototype, we draw on the crash-tolerant Paxos [103] protocol for reliable ordering in the agreement stage. All RUBiS clients are executed on a separate physical server in the cluster which does not host any replicas. Clients are linked to replica managers via TCP connections; using static load balancing, we ensure that all replica managers are contacted by an equal number of clients. During the experiments, we not only measure throughput performance but also collect information on the resource footprint of the server side; in particular, we are interested in the CPU, disk, and network usage of replica managers as well as both active and passive execution replicas.

System Configurations In our evaluation, we compare SPARE against two other approaches targeting fault tolerance for network-based services, in the following referred to as **CRASH** and **APPBFT** (see Figure 4.16). CRASH represents a typical way for an application provider to make a service resilient against crashes in a virtualized environment (e.g., an Infrastructure-as-a-Service cloud) without support of the infrastructure provider: Lacking the possibility to execute own code in the privileged domain of a server, all system components in CRASH are run in the user domain. As the system is only designed to tolerate crashes, $g + 1$ execution replicas are sufficient to provide safety in the presence of at most g faults in the environment addressed (see Section 4.2.1). Furthermore, in contrast to SPARE, there is no need to perform reply verification in CRASH: With the fault model assuming that all replies generated by execution replicas are correct, a replica manager is allowed to return the first reply available as a result to the client. Tolerating only crashes, the measurement results obtained from experiments with



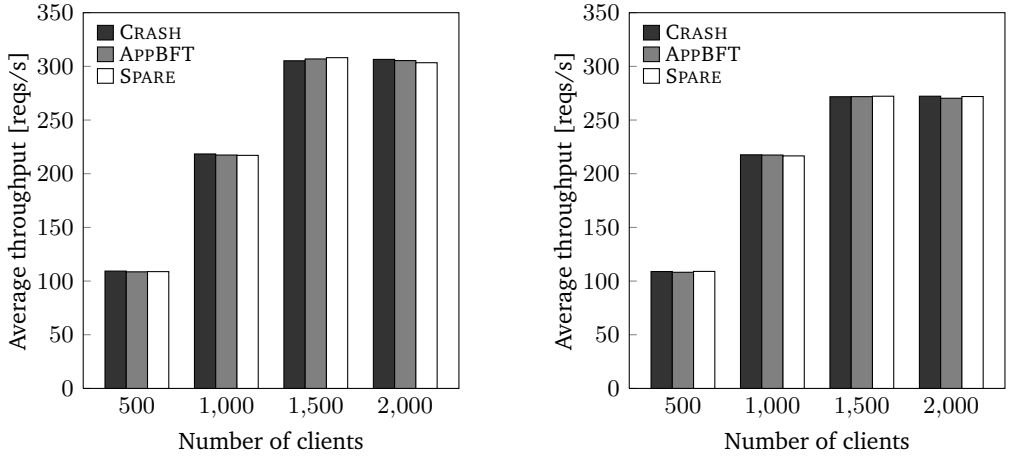
(a) CRASH architecture **(b)** APPBFT architecture

Figure 4.16: Overview of the architectures of CRASH and APPBFT cells dimensioned for tolerating a single fault: CRASH comprises two active execution replicas and is resilient to crashes of system components, whereas APPBFT requires three active execution replicas and, like SPARE, is able to cope with a Byzantine fault in an execution replica.

CRASH serve as a lower bound for the resource footprint achievable in SPARE. However, the overhead necessary for resilience against Byzantine faults in execution replicas is expected to prevent SPARE from actually being as resource-efficient as CRASH.

In contrast to CRASH, APPBFT is based on the same fault model as SPARE (see Figure 4.3) and also provides the same fault-tolerance guarantees; that is, APPBFT is able to tolerate up to f Byzantine faults in user domains while being resilient to crashes in the remaining system. However, unlike SPARE, APPBFT relies on traditional active replication and is therefore distributed over $2f + 1$ physical servers, each hosting its own execution replica. Given these characteristics, APPBFT implements the system design proposed by VM-FIT [130]. Evaluating APPBFT allows us to assess the effects of passive replication on the resource savings of SPARE.

Throughout the evaluation, we use system configurations that are able to tolerate a single fault (i.e., $f = g = 1$); as a result, our settings comprise two physical servers for



(a) Small database **(b)** Large database

Figure 4.17: Results of the RUBiS benchmark during a four-minute runtime session for CRASH, APPBFT, and SPARE in dependence of the number of concurrent clients and the size of the database: All three configurations achieve matching throughput performance.

both CRASH and SPARE, and three physical servers for APPBFT. In order to minimize the impact of implementation-specific factors on measurements, the prototypes of CRASH and APPBFT have been implemented reusing as much of the SPARE code base as possible. This way, differences in experiment results are expected to be caused by actual differences in system designs, not by heterogeneous realizations.

RUBiS Configurations User domains in our evaluation each host an instance of the server side of the modified RUBiS benchmark presented in the case study in Section 4.10; that is, an execution replica comprises both a Jetty web server executing a set of Java servlets to process client requests as well as a MySQL database storing information about registered users and item auctions. Before starting a test run, all replicas are initialized with the same database state. As database size affects the processing time of requests (i.e., the larger the database, the more records have to be searched), we repeat our experiments with two different initial application states: a *small database* containing about 100,000 users and 500,000 bids, and a *large database* storing about a million users and five million bids.

4.11.2 Performance

We evaluate the throughput performance of CRASH, APPBFT, and SPARE during normal-case operation by configuring RUBiS clients to execute the default *bidding mix* usage scenario for six minutes, including a runtime session of four minutes. To address different load scenarios, we vary the number of clients concurrently accessing the service from 500 to 2000; note that 2000 clients saturate the service in our experimental setting.

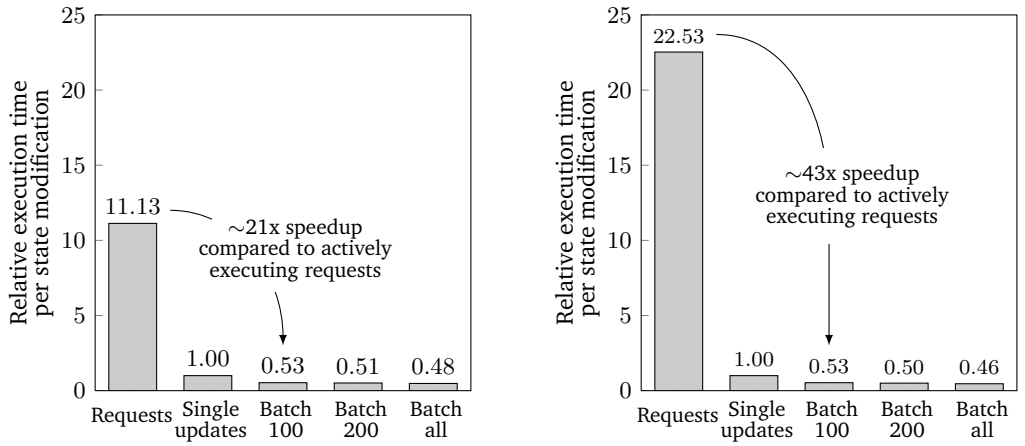
Figure 4.17 presents the results of the RUBiS experiments; they show that in all scenarios, the throughput realized for AppBFT and SPARE is within 1% of the throughput realized for CRASH. There are mainly two reasons leading to these results: First, for experiments with more than 1,500 concurrent clients, system performance is dominated by costly database operations performed by the execution stage. Therefore, agreement and verification overhead only has little effect on overall throughput. Second, with all three system configurations relying on the same protocol for request ordering, differences in performance are also negligible in cases where the impact of the agreement stage is increased (i.e., experiments with 1,000 and less clients).
With SPARE achieving similar results as AppBFT, the measurements also show that the need to maintain additional passive execution replicas in SPARE comes at no extra cost in terms of throughput performance in RUBiS. In the following section, we investigate the difference between active and passive execution replicas in more detail.

4.11.3 Active vs. Passive Execution Replicas

The main intent of introducing passive replication into SPARE was to achieve a smaller resource footprint compared to a system purely based on active replication (i.e., AppBFT). In the following, we evaluate how resource consumption differs between active and passive execution replicas in order to be able to assess whether the use of passive replicas can actually lead to a reduced resource footprint of the overall system.

4.11.3.1 Processing Requests vs. Applying State Updates

As discussed in Section 4.2.2, a crucial factor for the resource efficiency improvements possible through passive replication is the assumption that for the particular application in question actively processing a state-modifying request is significantly more costly than applying the corresponding state update. To investigate whether this assumption holds for RUBiS, we record all state-modifying requests issued by clients during a benchmark run, as well as their corresponding state updates, and replay both individually on a separate, newly initialized execution replica; that is, instead of applying a read/write workload as done in the experiments in Section 4.11.2, the RUBiS instance in this experiment has to cope with a write-only workload consisting of either a set of requests or the same number of state updates. Besides submitting state updates individually, we also conduct additional experiments in which we apply updates in batches of different sizes (see Sections 4.3.2 and 4.10.2.4).
Figure 4.18 presents the outcome of this group of experiments; for better comparison, we normalized the numbers to the time it took to apply a single state update to the RUBiS instance. Our results show that processing requests takes more than an order of magnitude longer than performing the same state modifications by applying updates, proving that our assumption holds for RUBiS. The reason for the significant overhead accompanied with executing requests is the fact that all tiers of the RUBiS service (see Figure 4.13) are involved during processing: After having been received by the web server in the presentation tier, a request is executed by a RUBiS servlet in the application tier, which then



(a) Small database **(b)** Large database

Figure 4.18: Comparison of relative execution times per state modification for different methods of bringing a RUBiS instance up to speed: Applying updates is significantly faster than executing the corresponding requests; batching updates further improves efficiency.

issues a SQL statement to the database. In contrast, applying a state update in SPARE only requires a single call to the database (see Figure 4.14), thereby bypassing unnecessary processing steps in upper tiers, amongst others for example: unmarshalling the HTTP request, executing the application logic, issuing (read-only) database queries necessary to check request validity as well as to assemble the state-modifying SQL statement, and creating an HTTP reply for the client. In consequence, an update is a much more efficient means to perform a state modification than a request.

Besides illustrating that state updates in passive execution replicas are less costly than state modifications in active execution replicas, the results presented in Figure 4.18 also show that making use of batches can further speed up the updating process: By combining multiple SQL statements (see Section 4.10.2.4 and Figure 4.15), for example, it is possible to reduce the execution time per state modification by almost half when applying 100 updates at once. Our results also indicate that increasing the batch size beyond 100 only provides limited additional improvements: Combining all state updates issued during the entire RUBiS benchmark run to a single batch reduces the relative execution time per state modification to 0.48 for the small database (0.46 for the large database), a decrease of only 10% (15%) compared to a batch size of 100.

4.11.3.2 Continuous Execution vs. Sporadic Wake-ups

During normal-case operation active execution replicas are running without interruption, continuously processing client requests. In contrast, as discussed in Sections 4.3.2 and 4.9.1.3, passive execution replicas in SPARE are primarily kept in resource-saving mode and only periodically woken up in order to be brought up to date. Figure 4.19

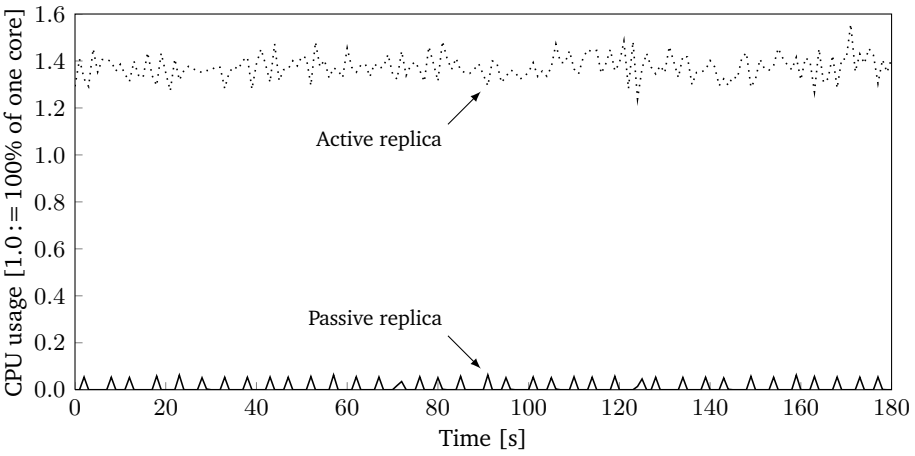


Figure 4.19: CPU-usage comparison (excerpt) between an active execution replica and a passive execution replica hosted on the same physical server during a small-database RUBiS experiment with 2000 concurrent clients: While the active user domain is continuously running, the passive user domain only consumes processor time when its execution replica is woken up from resource-saving mode in order to be brought up to date.

shows that this approach has a significant impact on CPU usage: While being paused, a user domain hosting a passive execution replica is not scheduled by the virtual-machine monitor and consequently does not consume any processor time, as opposed to a user domain hosting an active execution replica, which is always running. Having been unpaused to apply a batch of state updates, the passive execution replica becomes runnable only for the duration of the updating process.
For our experiments presented in Section 4.11.2 we use a batch size of $U_{max} = 100$ resulting in a single updating procedure to take a total of about 480 milliseconds to complete; this includes about 210 milliseconds to wake up the user domain, about 32 milliseconds for preparations (i.e., optimizing and serializing the update batch, establishing a connection to the database etc.), about 28 milliseconds for applying the batch, and finally about 210 milliseconds for putting the user domain back into resource-saving mode. Note that a large fraction of the total time is spent on tasks whose durations are independent of the size of the update batch, for example, unpausing/pausing the user domain. Therefore, increasing batch size would lead to an improved update-to-overhead ratio and consequently an increase in resource efficiency. However, as further investigated in Section 4.11.5 and discussed in Section 4.12, such a reconfiguration would also prolong SPARE's handling of faulty execution replicas.

4.11.3.3 Comparison of Resource Usage Characteristics

In the following, we analyze differences in usage characteristics between active and passive execution replicas for three key resource types (i.e., CPU, network, and disk) and investigate how they contribute to reducing the resource footprint of SPARE.

CPU Based on our evaluation results, in the previous sections, we have concluded that passive execution replicas have a lower CPU usage than active execution replicas due to processing updates instead of requests (see Section 4.11.3.1) and spending most time in resource-saving mode instead of running without interruption (see Section 4.11.3.2). In addition to these two, there is another crucial factor allowing passive execution replicas to save CPU resources: As further discussed in Section 4.12, in the domain of network-based services SPARE was designed for, workloads are usually dominated by read-only requests not modifying the state of the application [63, 70, 107]. In RUBiS, for example, most requests issued by clients only perform queries on the database (e.g., searching for items) or do not access the database at all (e.g., loading the cover page of the website). As read-only requests do not lead to any state updates, only active execution replicas must provide resources for processing them; passive execution replicas, in contrast, are not affected by such requests at all. As a result of all three factors mentioned above, a passive execution replica in SPARE is able to save more than 99% in CPU usage compared to an active execution replica.

Network Analyzing the network usage of active and passive execution replicas, we see that the same factors that are responsible for a decrease in CPU usage of passive execution replicas also lead to a reduction in the amount of data to be transferred: While interaction with active execution replicas includes the exchange of both a request and a reply, passive execution replicas exclusively operate on updates, which are usually much smaller than application-level messages, as they only contain the state modifications to be performed. For the RUBiS use case, our evaluation shows that the combined size of a state-modifying request and its reply is on average 13 times larger than the corresponding state update. In addition, as discussed above, the fact that most of the workload is read-only means that for a large fraction of operations no updates have to be sent at all, further reducing the network usage of a passive execution replica.

Disk While passive execution replicas allow a system to minimize its resource usage of both CPU and network, there is one resource type which offers only small savings: persistent storage; that is, the amount of data written to disk is nearly the same for both active and passive execution replicas. This is caused by the fact that a passive execution replica is required to independently manage its own full copy of the application state, which needs to be kept up to date by reproducing all relevant modifications that have also been performed by the active execution replica. One opportunity to optimize resource efficiency for passive execution replicas with regard to disk writes arises when multiple updates in a batch affect the same state part. Such a scenario, for example, may occur when the same database entry is updated more than once within a short period of time. In this case, the active execution replica must invoke multiple write operations (i.e., one for each update) whereas for the passive execution replica it is sufficient to only modify the database entry once (i.e., applying the last update which supersedes the others). In our experiments with RUBiS the effect of such optimizations on the disk usage of passive execution replicas were negligible.

Summary Taking the differences in usage characteristics between active and passive execution replicas into account, SPARE's use of passive replication has the potential to significantly minimize the consumption of CPU and network resources of the overall fault-tolerant system. In the following section, we evaluate the extent of the resource savings possible in more detail.

4.11.4 Resource Footprint

During the course of the experiments discussed in Section 4.11.2, we have collected detailed information about CPU, network, and disk usage of virtual and physical machines in the three system configurations. In this section, we assemble this data to present and compare resource footprints of CRASH, APPBFT, and SPARE for the different scenarios evaluated. Note that due to the fact that the three system configurations provide matching performance in all cases (see Figure 4.20), the results of the resource usage measurements do not require any normalization and can therefore be directly used for comparison. We start our discussion by focusing on the average resource usage of a single physical server in each of the three systems. In the next step, we compare overall resource footprints that take the total number of physical servers in each system configuration into account; that is, they consider that CRASH and SPARE in our evaluation setting both comprise two physical servers, while APPBFT is distributed over three physical servers.

4.11.4.1 Resource Footprint of a Physical Server

Although performing similar tasks, the characteristics of servers, including the number of hosted user domains as well as the load distribution between local virtual machines, differ in the three system configurations evaluated. Our measurement results presented in Figure 4.20 show the impact of these specific characteristics on the consumption of different resources types (i.e., CPU, network, and disk) for a single physical server. For CRASH and APPBFT, the CPU resource footprint of a server comprises the resource usage of both the privileged domain and the user domain running the active execution replica; for SPARE, the footprint also includes the CPU usage of the user domain hosting the passive execution replica. Note that for network usage the numbers presented reflect the amount of data transmitted to other servers using the physical network card; communication between user domains hosted on the same server is not included, as it does not affect the external network. Furthermore, for disk usage we only report the amount of data written to disk by user domains, as none of the three system configurations comprises a component in the privileged domain that manages data on disk.

General Observations The results in Figure 4.20 show that for CPU, network, and disk, the amount of resources consumed is mainly dependent on the system throughput realized: With more client requests being processed, more data is exchanged over the network, and more state modifications need to be made persistent within the same period of time. Focusing on CPU usage, the numbers for APPBFT and SPARE furthermore

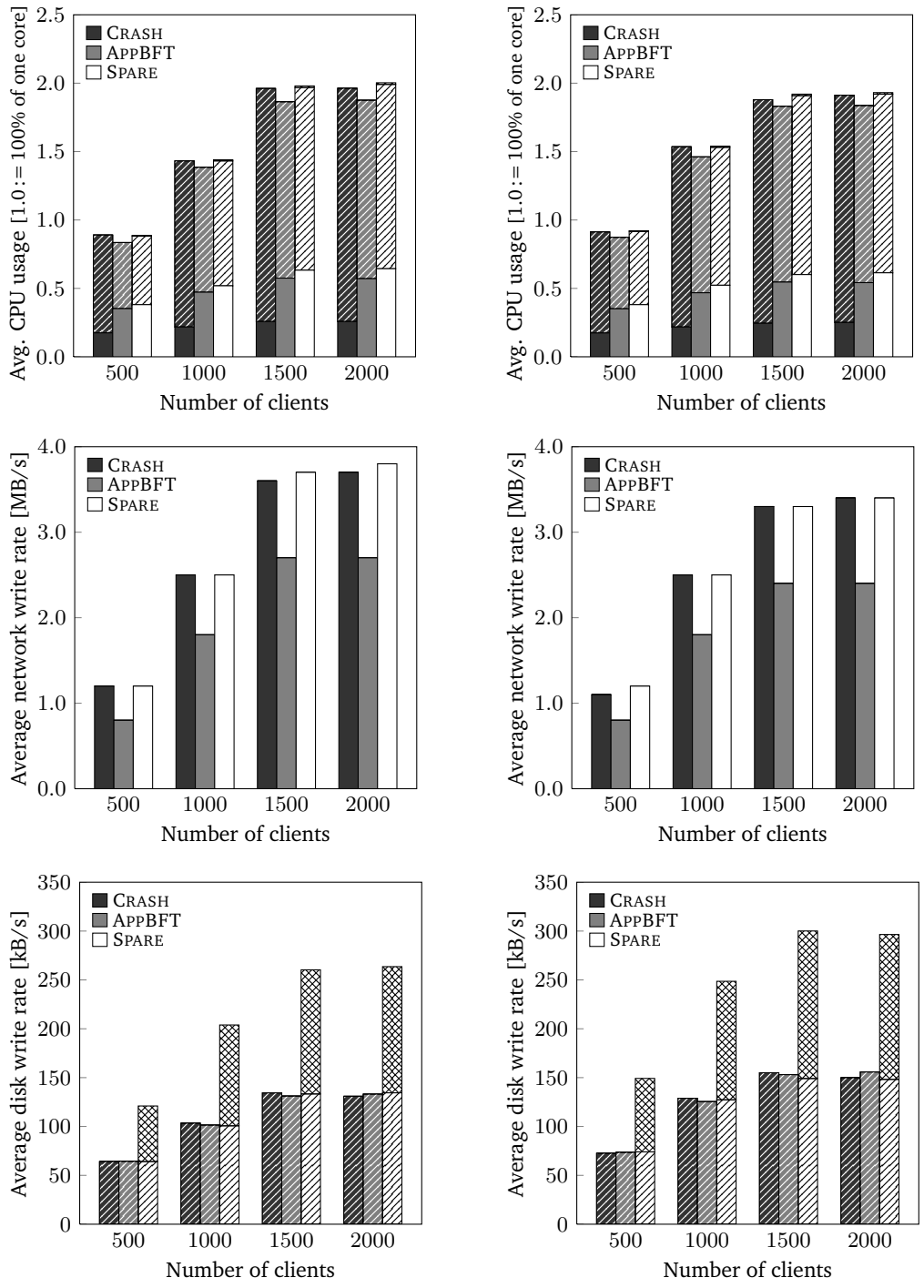


Figure 4.20: Comparison of the server resource footprints of CRASH, APPBFT, and SPARE for the privileged domain (▭), the active user domain (▨), and the passive user domain (▩) for the RUBiS benchmark using a small database (left) and a large database (right), respectively: Usage of CPU, network, and disk is mainly dependent on the throughput realized.

reveal a desirable property of a fault-tolerant system: Most processor time is spent on operations in the application running in the active user domain and not on replication-related overhead (e.g., request ordering and reply verification) in the privileged domain hosting the replica manager.

Comparison between SPARE and CRASH The measurement results presented in Figure 4.20 show that a SPARE server requires 1% more CPU and 2% more network resources compared to a server in CRASH. The reason for the two system configurations consuming similar amounts is the fact that usage of these resource types is dominated by the overhead for handling communication with clients (i.e., receiving requests and sending replies), which is equal in both cases. However, the location in which resource consumption for this task arises differs between CRASH and SPARE, as the CRASH replica manager runs in the same virtual machine as the active execution replica (see Section 4.11.1). As a result, the privileged domain in CRASH only consumes the resources (i.e., CPU in particular) necessary to manage the virtualization overhead for the active user domain. In general, differences in resource usage between SPARE and CRASH are worth to be investigated in more detail as they reflect the costs of maintaining a passive execution replica in SPARE: For CPU and network usage, as mentioned above, the overhead is almost negligible, which shows that exchanging and voting on state updates is a very efficient means to bring passive execution replicas up to speed. Regarding disk writes, the use of passive replication is more costly leading to 96% more data to be made persistent due a passive replica maintaining its own copy of the service state (see Section 4.11.3.3).

Comparison between SPARE and APPBFT Our measurement results show that a SPARE server consumes more resources than an APPBFT server for all three resource types evaluated. The increased resource demand mainly stems from two factors: First, an APPBFT server in our experimental setting, which is able to tolerate a single fault (i.e., $f = 1$), must handle only a third (i.e., $\frac{1}{2f+1}$) of all client connections, whereas a SPARE server is connected to half (i.e., $\frac{1}{f+1}$) of all clients. Second, managing and updating the passive execution replica in SPARE, as discussed in Section 4.11.3.3, requires a small amount of network capacity for the transfer of state updates as well as disk space for an additional copy of the application state. In total, both factors account for about 6% more CPU (i.e., 4% in the privileged domain, 1% in the active user domain, and less than 1% in the passive user domain), about 38% more network, and about 97% more disk usage per physical server. Note that the 38% increase in network usage is less than one might expect at first glance for a SPARE server which has to handle 50% more client connections than an APPBFT server: The difference is accounted for by the fact that SPARE's agreement stage only needs to forward each client request to f instead of $2f$ other servers.

Summary A comparison of the server resource footprints shows that the ability to tolerate Byzantine faults in execution replicas in SPARE comes at the cost of an increased resource consumption compared with a system configuration that only provides resilience against crashes (i.e., CRASH). Furthermore, when limiting the focus to a single server,

SPARE's reduced cell size (i.e., $f + 1$ servers instead of $2f + 1$) and the design choice to co-locate passive execution replicas with their active counterparts on the same physical machine even lead to a server resource footprint larger than that of a server in a comparable system configuration relying on traditional active replication (i.e., APPBFT).

4.11.4.2 Overall Resource Footprint

The measurement results presented in Section 4.11.4.1 have shown that a physical server in SPARE obtains more resources than a physical server in CRASH and APPBFT. To put these values into perspective, we now discuss the overall resource footprints of the three system configurations, which respect differences in cell sizes; that is, the system configurations evaluated in our experiments comprise two servers (i.e., $g + 1$ and $f + 1$, respectively) for both CRASH and SPARE, and three servers (i.e., $2f + 1$) for APPBFT. We calculate the overall resource footprint of a system configuration for each resource type by multiplying the average usage of a single physical server from Figure 4.20 with the respective cell size. The resulting overall resource footprints are presented in Figure 4.21.

Comparison between SPARE and CRASH With a CRASH cell comprising the same number of physical servers as a SPARE cell but providing only resilience against crashes, its overall resource footprint serves as a lower bound for the resource usage of SPARE. Furthermore, due to the structural similarities of both system configurations, differences between their resource footprints reveal the overhead of SPARE's mechanisms for tolerating Byzantine faults in execution replicas; that is, the management of an additional, passive execution replica per server as well as the verification of replies by voting. Our results show that the combined resource consumption of both mechanisms is modest for two of the types evaluated: Compared with CRASH, SPARE uses 1% more processor time and sends only 2% more data over the network. As discussed in Section 4.11.4.1, disk usage constitutes an exception: Maintaining a separate copy of the application state on persistent storage leads to SPARE writing 96% more data to disk than CRASH.

Comparison between SPARE and APPBFT Providing identical fault-tolerance guarantees, a comparison of the overall resource footprints of SPARE and APPBFT reveals the resource-saving effects of SPARE's central design decision: the use of passive replication in combination with a reduced cell size of only $f + 1$ instead of $2f + 1$ physical servers. An analysis of the overall resource footprints presented in Figure 4.21 shows that the smaller number of servers is of particular importance in this context: With less execution replicas taking part in request processing, SPARE uses 30% less CPU than APPBFT. For the same reason, and due to the fact that the overhead for exchanging state updates is small, 8% less data needs to be transferred over the network in SPARE. In contrast, persistent storage is the only resource type for which consumption in SPARE shows a different picture: Due to all of the $2f + 2$ (i.e., $f + 1$ active and $f + 1$ passive) execution replicas maintaining their own copies of the application state, disk writes are increased by 31% compared with APPBFT, which needs to keep a total of $2f + 1$ copies up to date.

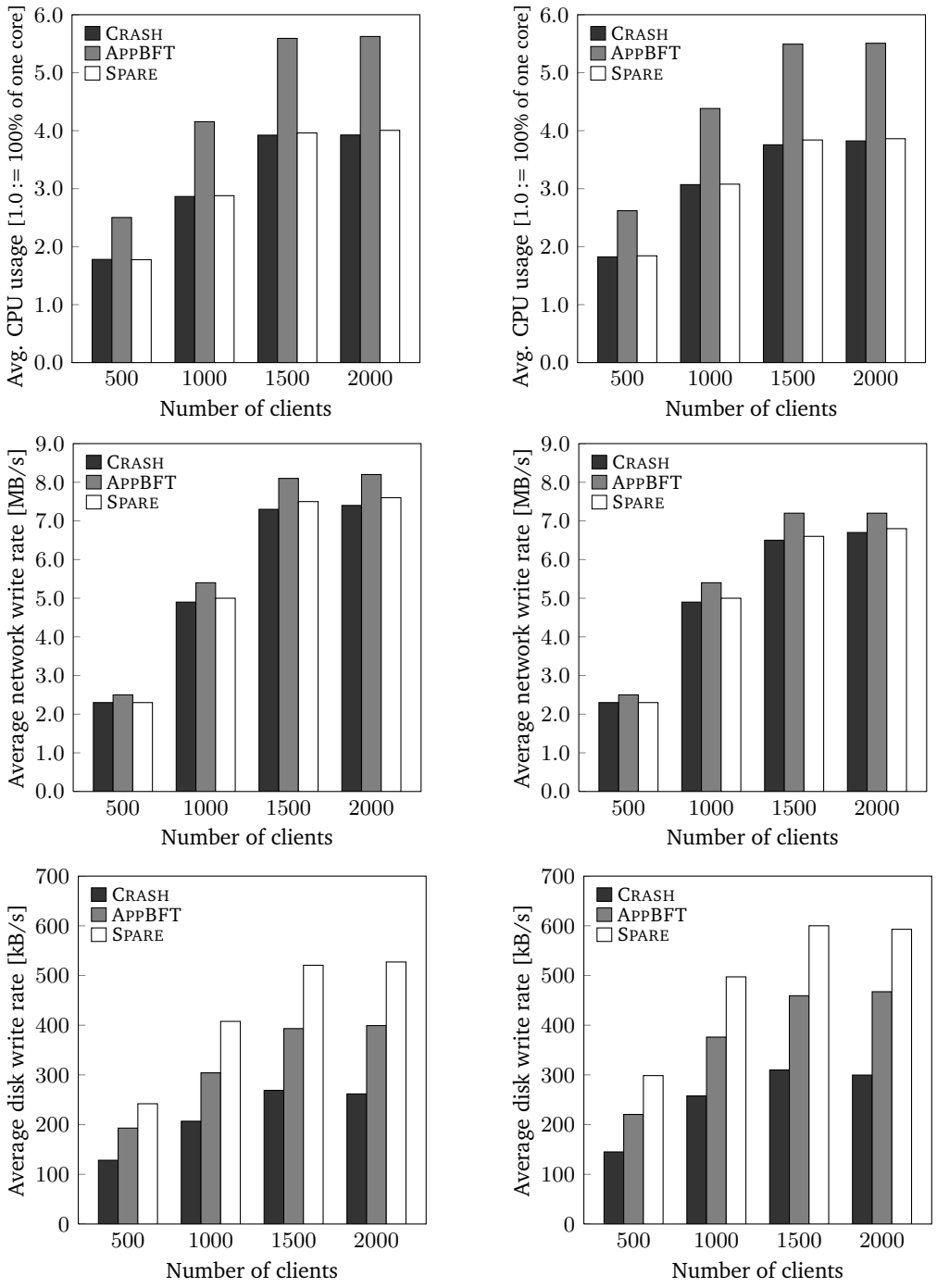
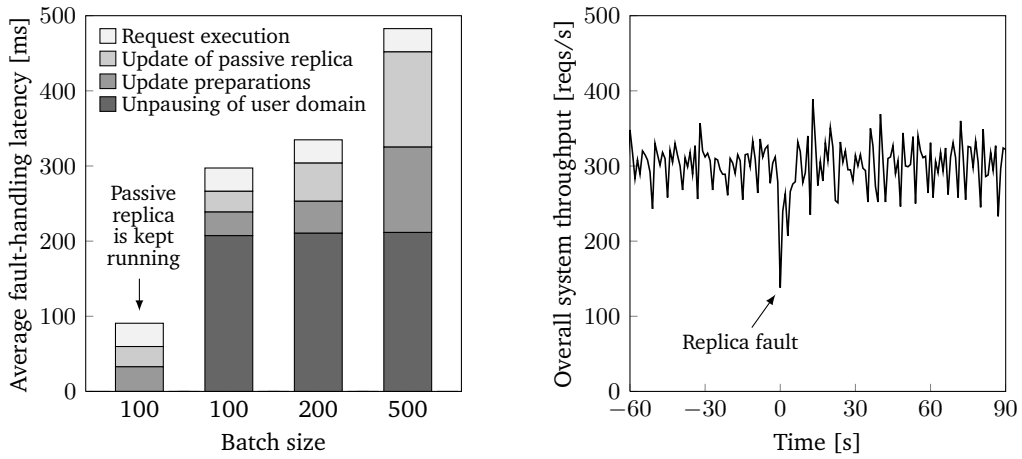


Figure 4.21: Comparison of the overall resource footprints of CRASH, APPBFT, and SPARE for the RUBiS benchmark using a small database (left) and a large database (right), respectively: The results show that, with the exception of disk writes, providing Byzantine fault tolerance for service applications in SPARE requires a negligible resource overhead compared with traditional systems that are only resilient against crashes; furthermore, the use of passive replication allows SPARE to save resources compared with systems relying on plain active replication.



(a) Fault-handling latency of SPARE for different update-batch sizes during small-database RUBiS experiments with 1500 clients.

(b) Impact of a replica fault on throughput during a small-database RUBiS experiment with 1500 clients and a batch size of 500 updates.

Figure 4.22: Evaluation of SPARE's fault-handling mechanism: When an active execution replica crashes, it takes the system a few hundred milliseconds to tolerate the fault.

Besides contributing to a reduction in CPU and network usage, SPARE's small cell size also leads to savings for another resource type not discussed so far: power. With the power consumption of today's state-of-the-art servers not being proportional to the machines' current workloads [17, 37], the small load increase induced by passive replication on each of the $f + 1$ SPARE servers is expected to have negligible effects compared with requiring f additional physical machines in APPBFT. As a result, SPARE has the potential to reduce total power consumption by almost a third, thereby not only improving resource efficiency but also allowing providers to save on energy costs.

Summary A comparison of the overall resource footprints of the three system configurations evaluated shows that with the exception of persistent storage, which we discuss in more detail in Section 4.12.2, passive replication in SPARE can be realized with little resource overhead; for CPU and network usage, for example, SPARE comes close to the lower bound provided by CRASH. Furthermore, our results prove that the use of passive execution replicas, the application of resource-saving modes for virtual machines, and the small cell size of the overall system in particular, allow SPARE to significantly reduce its resource footprint while providing the same fault-tolerance guarantees as the traditionally-replicated APPBFT.

4.11.5 Fault Handling

The results presented in Section 4.11.4 show that the use of passive replication allows SPARE to reduce the amount of resources required to provide resilience against Byzantine faults in execution replicas. However, the increase in resource efficiency comes at

the cost of having to activate passive execution replicas first before the system is prepared to actually tolerate a fault. In the following, we investigate SPARE's fault-handling latency (i.e., the time it takes to tolerate a fault after it has been suspected/detected) as this factor is crucial for the practicality of the system.

Experiment Description To evaluate SPARE's fault handling, we manipulate an active execution replica in order for it to crash during an experiment with 1,500 RUBiS clients concurrently accessing the service that has been initialized with the small database. As the fault-handling latency depends on the number of updates to be applied before a passive execution replica is able to assist in tolerating a fault, we conduct experiments with different batch sizes U_{max}. Note that, in all cases, we create a worst-case scenario for SPARE by ensuring that the fault is triggered at a time when the virtual machines of passive execution replicas are in resource-saving mode and the update buffer is almost completely filled, containing $U_{max} - 1$ state updates (see Section 4.3.2). In consequence, fault-handling procedures for a request whose reply verification has stalled in our experiments include: First, waking up the user domain of a passive execution replica. Second, applying $U_{max} - 1$ state updates. Third, executing the request affected by the fault in order to provide the deciding reply that allows the stalled verification process to complete successfully (see Section 4.4.2).

Correlation between Batch Size and Fault-handling Latency Figure 4.22a shows the fault-handling latencies of SPARE for different batch sizes ranging from 100 to 500. For comparison, we also present the result of an experiment with a batch size of 100 in which passive execution replicas are not put into resource-saving mode but kept running after having been updated. As a consequence, in this case, there is no need to unpause the user domain of the passive execution replica before being able to progress with applying state updates, resulting in an overall fault-handling latency of about 91 milliseconds. In contrast, the fault-handling latency in all other experiments is prolonged by the about 210 milliseconds it takes in our test environment to wake up the virtual machine comprising the passive execution replica (see Section 4.11.3.2).
Once the virtual machine of a passive execution replica is running, the batch size dictates how fast the remaining fault-handling procedures can be completed in worst-case scenarios. There are two main reasons why larger batches lead to higher fault-handling latencies: First, preparation of the updating process, including, for example, the serialization and optimization of the batch, is more costly for a large batch compared to a small batch. Second, with more state modifications to perform, applying a large batch to a passive execution replica takes more time. Note that the results for the update phase of the fault-handling process in Figure 4.22a confirm our findings presented in Section 4.11.3.1: With execution times per state modification decreasing only slightly for batch sizes above 100, applying 499 state updates in a batch takes about five times longer than processing a batch of 99 state updates, prolonging fault handling by about 100 milliseconds, just in this phase.

Impact of a Replica Crash on Performance Despite differences in durations for different batch sizes, our evaluation shows that the overall fault-handling latency of SPARE is moderate in all the cases evaluated: When fault-handling procedures are initiated, it takes the system a total of about 300 to 500 milliseconds to make progress by activating a passive execution replica to tolerate the fault. In consequence, a fault causes only a minor disruption in system performance as illustrated by the measurement results presented in Figure 4.22b, taken during the course of a small-database experiment with 1,500 clients and a batch size of 500: With a fault-handling latency of about half a second, in this scenario, system throughput drops to 138 requests per second (i.e., about 45% of normal) directly after the crash of the active replica. However, after a short warmup phase of about five seconds, in which throughput increases, the activated passive execution replica has fully replaced the crashed replica. Note that the crash of the active execution replica has been transparent to clients: All requests issued have been successfully processed by SPARE and none of the client connections timed out.

4.12 Discussion

In the following, we discuss a number of aspects that arise from the evaluation results presented in Sections 4.11.2 to 4.11.5. In particular, we focus on the problem of selecting the right batch size for state updates as well as on SPARE's elevated usage of persistent storage compared with other system configurations. Furthermore, based on our evaluation results for RUBiS, we investigate whether similar resource savings are also possible for other network-based services.

4.12.1 Trading off Resource Savings for Fault-handling Latency

Our evaluation has shown that SPARE achieves its goal of saving resources thanks to relying in parts on passive replication. While passive replication is an effective means to reduce the resource footprint of a fault-tolerant system, it usually comes with a drawback compared with active replication: an increase in fault-handling latency caused by the fact that a passive execution replica first has to be updated before being able to assist in tolerating a fault. As a general rule: The more a passive execution replica has fallen behind, the longer it takes to bring the replica up to speed. Therefore, on the one hand, frequent updates of passive execution replicas are desirable in order to achieve a low fault-handling latency. On the other hand, frequent updates increase the overhead for passive replication resulting in a larger resource footprint. In summary, there is a tradeoff between a low fault-handling latency and high resource savings.

In SPARE, the means of choice to balance this tradeoff is the size of the state-update batch (see Section 4.8.2): Larger batch sizes lead to passive execution replicas being woken up less frequently from resource-saving mode but also result in higher fault-handling latencies (see Section 4.11.5); reducing the batch size improves fault-handling latency at the cost of more frequent updates. Note that the overhead for unpausing/pausing

the user domain of a passive execution replica sets a lower bound on the practical size of an update batch: With such an operation taking about 210 milliseconds (see Section 4.11.3.2), at some point, reducing the batch size will not increase the update frequency any further. Instead, to achieve an optimal fault-handling latency, one would keep passive execution replicas running and not put the replicas back into resource-saving mode after they have been updated.

4.12.2 Disk Overhead for Passive Execution Replicas

As discussed in Section 4.11.3.3 and underlined by the results in Sections 4.11.4, persistent storage constitutes an exception among resource types: It is the only type of which SPARE requires more than APPBFT due to the fact that each passive execution replica is required to operate on its own full copy of the application state. As a result, the comparison of overall resource footprints in Section 4.11.4.2 revealed an increase of 31% in the total amount of data written to disk for the RUBiS multi-tier application.

Is SPARE's Disk Overhead Likely to Be a Problem in Practice? There are mainly three reasons why the additional disk usage of passive execution replicas in SPARE is not likely to be a major drawback in practice: First, taking into account that disk space is inexpensive and abundantly available in today's data center infrastructures [32, 75], an increase by less than a third can be considered acceptable. Second, a comparison of access characteristics shows that disk writes of passive execution replicas in SPARE are expected to cause less overhead than disk writes of active execution replicas: While active execution replicas invoke separate write operations each time they process a state-modifying client request, passive execution replicas only change the service state in the course of applying an update batch, offering the potential to handle all modifications in a single write operation [24]. Third, not all applications behave in the same way as RUBiS with regard to disk usage: While all data written to disk in RUBiS constitutes an essential part of the service state, many other applications make use of temporary files (e.g., to store intermediate results that would not fit into memory [54]) in the course of processing a request. Thus, although such files are written by the active execution replica, they are not part of the service state and should therefore be left out of the state updates applied to the passive execution replica. Note that the same approach may, for example, be used for debug logs produced by active execution replicas if the logs are not considered essential.

Can Data Deduplication Reduce SPARE's Disk Overhead? In an effort to reduce the disk overhead in SPARE, one might consider applying the concept of data deduplication [117, 129] to minimize the storage footprint for the service states of both execution replicas hosted on the same physical server. Such an approach would exploit the fact that both copies of the service state are very similar: As a result of the deduplication process, the amount of data actually written to disk would be reduced by storing chunks that are included in both copies only once. Note that, however, making use of data deduplication

is not possible in SPARE as it would create a dependence between both execution replicas that might result in correlated failures (see Section 4.6). Independently, if execution replicas rely on heterogeneous implementations using different internal state representations (see Section 4.6.2), it is unlikely that the application of data deduplication would result in significant savings in the first place.

4.12.3 Transferability of Results

In our evaluation, we analyzed the throughput performance and resource footprint of SPARE using the RUBiS benchmark as application scenario. Based on the results and insights obtained from these experiments, in the following, we discuss implications for other network-based services.

Throughput Performance For all experiments conducted with RUBiS, the three system configurations evaluated (i.e., CRASH, APPBFT, and SPARE) achieved similar throughput, proving that for this use case the overhead for maintaining passive execution replicas in SPARE has no observable (negative) effect on performance. As discussed in Section 4.11.2, the fact that database operations dominate processing times in RUBiS is responsible for differences in the agreement stages of system configurations not resulting in differences in throughput performance. Note that this does not necessarily have to be the case for all applications: Replicating services with very short processing times, for example, shifts the pressure from the execution stage to the agreement stage, eventually up to the point where request ordering becomes the decisive factor influencing performance. In such scenarios, their reduced cell size is expected to give CRASH and SPARE an advantage over APPBFT due to less messages being sent by the agreement stage.
However, there might also be scenarios in which SPARE performs worse than CRASH and APPBFT: Of the three system configurations, SPARE is most vulnerable to (nondeterministic) fluctuations in processing times of active execution replicas. The origin of this property lies in SPARE's voting circumstances during normal-case operation: With only $f + 1$ active execution replicas providing replies, the slowest reply dictates the point in time at which the result becomes stable. In contrast, result verification in APPBFT is complete as soon as the first $f + 1$ matching replies, out of a total of $2f + 1$ replies, are available; that is, the performance of APPBFT is not directly affected by a non-faulty but slow active execution replica. Performing no reply verification at all, the same applies to CRASH. Note that in case of fluctuations in processing times leading to a noticeable deterioration in performance, in order to mitigate the problem, SPARE can be configured to activate additional passive execution replicas as soon as it suspects or detects one or more active execution replicas to be slow. In consequence, more replies become available, decreasing the influence of slow execution replicas; this comes at the cost of an increased resource footprint.

Workload Characteristics In Section 4.11.3.3, we concluded that one of the decisive factors for a passive execution replica using 99% less processor time than an active execution replica is the fact that read-only requests constitute the majority of the workload

in RUBiS: Due to such requests leaving the application state unmodified when being processed by an active execution replica, passive execution replicas remain completely unaffected by them. Analyzing the workloads of network-based services often considered critical in data centers, we found that there is a large number of use cases in which read operations outnumber write operations. In the following, we discuss different examples from the fields of distributed file systems, Internet-commerce applications, and coordination services.

In a study on the workload characteristics of distributed file systems, in which they investigated NFS [145] traces in a university environment, Ellard et al. [63] conclude that, for a central email and service system, read requests outnumber write requests by a factor of three; about the same ratio applies to the I/O operations performed on behalf of those requests. In a different study, Leung et al. [107] focused on the distributed file system running in a large enterprise data center and identified a read-to-write ratio for I/O operations of more than 2:1. For TPC-W [70], a multi-tier benchmark application designed to simulate the characteristics of real-world Internet-commerce services, we found an even larger ratio of read-only requests to state-modifying requests: for the shopping-mix scenario reads outnumber writes by a factor of four. Finally, significant differences can also be observed in other application domains: The Fetching Service, which is part of Yahoo's search engine, relies on an external service to coordinate its processes; according to Hunt et al. [84], interaction between both services shows a read-to-write ratio between 10:1 and 100:1 during periods of high loads.

Given these workload characteristics, it is reasonable to assume that for all those network-based services an integration with SPARE would be of advantage: First, because it would allow them to tolerate Byzantine faults in execution replicas. Second, because SPARE's use of passive replication is expected to lead to significant resource savings compared to a traditional APPBFT-like approach solely based on active replication.

Overhead for Passive Replication Our comparison of overall resource footprints presented in Section 4.11.4.2 has shown that the overhead for passive replication in SPARE, with the exception of persistent storage, is modest: Distributing and verifying state updates, for example, only requires 1% more processor time and 2% more network resources in RUBiS. Apart from the effects of the read-mostly workload discussed above, the low overhead is founded in the fact that state updates are significantly smaller than their corresponding state-modifying requests, as evaluated in Section 4.11.3.3. Note that this characteristic of RUBiS does not necessarily apply to all other network-based services: In a distributed file system (e.g., NFS [145]), for example, large parts of a write request consist of data that represents the state modification (i.e., the contents of the chunk to be written to file). As such, this data must also be included in the corresponding state update, causing updates to be of similar size than client requests. The same is true for other applications including, for example, coordination services [22, 30, 84] and key-value stores [55, 68].

The effects of larger update sizes on the overhead for passive replication in SPARE are dependent on whether the system is configured to make use of update hashes or not: If hashing is disabled, more data needs to be sent over the network in order to distribute

an update. In contrast, if hashing is enabled, the calculation of update hashes becomes more expensive for larger state updates, resulting in a (presumably small) increase in CPU usage; however, network usage in this case is not affected at all, as the size of an update hash is independent of the size of the corresponding state update.

4.13 Chapter Summary

In this chapter, we examined whether the conventional wisdom that Byzantine fault tolerance inherently requires much more resources than crash fault tolerance actually holds true. Based on our results, we can conclude that this does not necessarily have to be the case. In contrast, our work on SPARE shows that by taking advantage of special properties of the virtualized environment available in today's data centers, it is possible to build an infrastructure service providing resilience against Byzantine faults at the service-application level that consumes nearly the same amount of CPU, network, and power as a comparable crash-tolerant service. For this purpose, passive replication in particular has proven to be an effective means enabling a fault and intrusion-tolerant system to minimize its resource footprint.
Having successfully introduced passive replication at the execution stage in the context of SPARE, in the next chapter, we focus on potential resource savings made possible by extending the concept to the agreement stage of a Byzantine fault-tolerant system.

5
Passive Byzantine Fault-tolerant Replication

While in SPARE we have focused on introducing passive replication at the execution stage of a Byzantine fault-tolerant system, in this chapter, we extend the concept to the agreement stage. As our main contribution, we present an approach which allows a subset of replicas in a system to be kept passive during normal-case operation in order to save resources. In contrast to active replicas, passive replicas neither participate in the agreement protocol nor execute client requests; instead, they are brought up to speed by verified state updates provided by the active replicas in the system. In case of suspected or detected faults, the system initiates a reconfiguration protocol that activates passive replicas in a consistent manner. To underline the flexibility of our approach, we present two different instances of our architecture: One that does not rely on trusted components and requires a total of $3f + 1$ replicas to tolerate up to f faults, and one that makes use of a trusted service for authenticating messages and therefore only comprises $2f + 1$ replicas. For both variants, we show that applying passive replication at the agreement stage does not require new agreement protocols to be developed from scratch: Both the resource-saving normal-case operation mode as well as the configuration switch can be implemented as additional protocols alongside existing agreement protocols.

5.1 Resource-efficient Agreement and Execution

With all replicas participating in system operations at all times, existing Byzantine fault-tolerant systems have large resource footprints. To address this problem, we present *resource-efficient Byzantine fault tolerance* (REBFT), an approach that introduces passive replicas at both the agreement stage as well as the execution stage of a fault and intrusion-tolerant system. In this section, we present the main goals behind REBFT and outline the key mechanisms and techniques used to achieve them.

5.1.1 Resource-efficient Agreement

As discussed in Section 2.2.1, when it comes to minimizing the resource overhead of Byzantine fault-tolerant agreement, the focus of research in recent years has been on reducing the minimum number of replicas required in a system [41, 131, 152, 154]. Note that, besides having to run fewer servers, such an approach has another benefit with regard to resource usage: With Byzantine fault-tolerant agreement protocols [34, 41, 152, 153, 154] relying on an all-to-all communication pattern (see Section 3.1.1), reducing the number of participants in the agreement protocol leads to fewer messages being sent over the network; this effect can even be increased by decreasing the number of protocol phases, as proposed by Veronese et al. in MinBFT [154]. Besides, Van Renesse et al. presented a protocol that draws on the concept of chain replication [132] to minimize network usage: In Shuttle [131], messages are sent along a chain of replicas, limiting the interactions of each replica to message exchanges with at most two other replicas.
In contrast to the static approaches mentioned above, REBFT exploits benign conditions to dynamically reduce the resource footprint of a Byzantine fault-tolerant system. To achieve this, REBFT builds on the idea of using different agreement protocols for different purposes [77]: During normal-case operation, a REBFT system runs a protocol in which only a subset of replicas participate actively; the subset is chosen to comprise the minimum number of replicas required to make progress in the absence of faults. In case of suspected or detected replica faults, REBFT initiates a transition protocol that performs a switch to a more robust (but less resource-efficient) agreement protocol, which is able to tolerate faults.

5.1.2 Resource-efficient Execution

Yin et al. [161] have shown that $2f + 1$ replicas are sufficient to be able to tolerate up to f Byzantine faults at the execution stage; this constitutes a significant reduction compared with the $3f + 1$ execution replicas used in other fault and intrusion-tolerant systems [34, 35, 153]. However, with regard to resource consumption, most Byzantine fault-tolerant systems that rely on $2f + 1$ execution replicas [41, 152, 154, 161] only have a single mode of operation: At all times, they consume the amount of resources required to handle the worst case of f replica faults by processing all client requests on all execution replicas available in the system.

In contrast, at the execution stage, REBFT applies a similar approach as ZZ [159] and SPARE (see Chapter 4): During normal-case operation, each request is only processed on the minimum number of replicas that allows a client to prove a result correct in the absence of faults; to actually tolerate faults, a request is processed on additional replicas. However, unlike ZZ and SPARE, REBFT does not assume replicas to run in a virtualized environment, thereby omitting the need to trust a virtualization layer.
Similar to SPARE, REBFT makes use of passive replication to save resources at the execution stage during normal-case operation. Furthermore, in the absence of faults, REBFT minimizes the overhead for creating checkpoints as, in contrast to existing Byzantine fault-tolerant systems [34, 35, 96, 152, 154, 161], it does not require the creation of periodic service-application snapshots.

5.2 The REBFT Architecture

This section presents an overview of the basic architecture of REBFT and of the requirements a service application needs to fulfill in order to be integrated. Note that in subsequent sections, we present two different instances of the REBFT architecture: REPBFT (see Section 5.3), which relies on the PBFT [34] agreement protocol and requires $3f + 1$ replicas to tolerate f faults, and REMINBFT (see Section 5.4), which comprises $2f + 1$ replicas and makes use of the MinBFT [154] protocol for fault tolerance.

5.2.1 Replicas

As illustrated by REPBFT and REMINBFT in subsequent sections, the total number of replicas in an instance of REBFT, in the following also referred to as a REBFT *cell*, may vary between different architecture variants. However, they all have in common that during normal-case operation replicas assume different roles in order to minimize the resource footprint of the system: Of all the replicas in the cell, f are *passive* which means that they do not participate in the ordering of client requests at the agreement stage and that they also do not process any client requests at the execution stage. All remaining replicas in the cell are kept *active* and fully participate in both stages; in particular, this includes providing passive replicas with state updates in order to bring them up to speed. Note that we do not make any assumptions on how replica roles are assigned as long as all nodes know which replicas are active and which replicas are passive; one possibility to solve this problem is to use totally-ordered replica ids and to select the replicas with the f highest ids to be passive.

5.2.2 Service Application

With a subset of execution replicas actively processing client requests, REBFT requires service-application instances to implement the same deterministic state machine [138]. In addition, similar to SPARE (see Section 4.2.2) and as shown in Figure 5.1, active execution replicas must put out state updates reflecting the effects of client requests

```
1 /* Execution of service requests */
2 [REPLY, UPDATE] processRequest(REQUEST request);

4 /* Application of state updates */
5 void applyUpdate(UPDATE update);
```

Figure 5.1: Overview of the functionality required from a service application to use the resource-saving mechanisms of REBFT (pseudocode): In order to support passive replication, a service instance must provide means to retrieve and apply the state updates triggered by client requests.

processed (L. 2). Having been verified, such state updates are used to bring the state of passive execution replicas up to speed (L. 5). Note that, in contrast to SPARE and other fault and intrusion-tolerant systems [34, 35, 152, 154, 161], REBFT does not necessarily need service applications to provide means for retrieving and setting their state in its entirety: Whether or not such a functionality is actually required depends on the fall-back protocol in use; during normal-case operation, REBFT relies on lightweight checkpoints that are created without involvement of the service application (see Section 5.1.2).

5.3 Resource-efficient Agreement and Execution based on PBFT

This section presents REPBFT, an instance of REBFT that relies on the PBFT [34] agreement protocol to ensure progress in the presence of faults and consequently requires $3f + 1$ replicas per cell. However, to save resources, under benign conditions only $2f + 1$ of the replicas actively participate in providing the service while f replicas remain passive (see Section 5.2.1). In the following, we provide details on the resource-saving protocol run in REPBFT during normal-case operation and present a mechanism that allows the system to switch to PBFT in case of suspected or detected faults.

5.3.1 Normal-case Operation

In the absence of faults, active replicas in REPBFT run the protocol shown in Figure 5.2. Similar to PBFT (see Section 3.1.1) actively participating replicas assume different roles; in REPBFT, the replica with the lowest id serves as *leader*, while all other active replicas are *followers*. In order to agree on a client request, active replicas execute three protocol phases (i.e., PREPREPARE, PREPARE, and COMMIT), as in PBFT. However, in contrast to PBFT, some replicas in REPBFT remain passive and, with the exception of state updates, do not exchange messages with other replicas during normal-case operation. In order to prove the identity of the sender, all agreement-protocol messages sent are authenticated.

5.3.1.1 Agreement Stage

Having received a request o from a client, the leader id_L is responsible for initiating the agreement process by assigning a sequence number s to the request and then sending a $\langle \text{PREPREPARE}, id_L, o, s, p \rangle$ message to all followers; p is the id of the current protocol

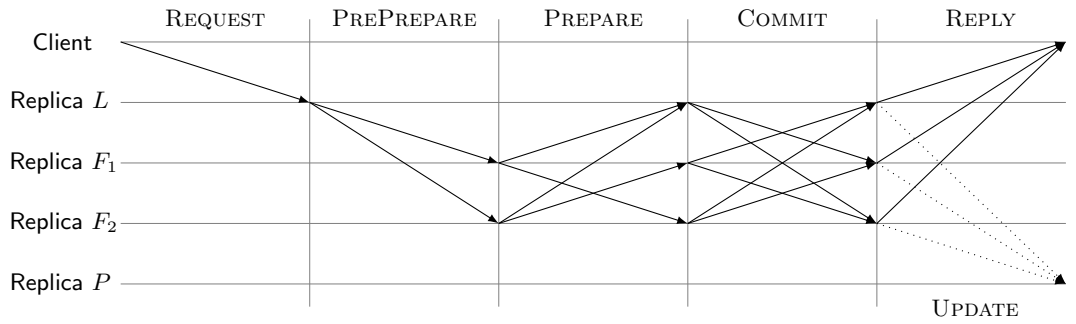


Figure 5.2: Message flow of requests, agreement messages, and replies (—) as well as state updates (····) in REPBFT for a cell that is able to tolerate one fault: Only active replicas (i.e., a leader replica L and two follower replicas F_1 and F_2) participate in the agreement and execution of client requests. In contrast, the passive replica P is brought up to speed by state updates.

generation and used by replicas to identify (and consequently drop) old messages, similar to the view information in PBFT. A follower accepts a PREPREPARE message from the leader if it has not yet accepted another PREPREPARE message binding a different request o' to the same sequence number s. Having accepted a PREPREPARE message, a follower id_F multicasts a $\langle \text{PREPARE}, id_F, o, s, p \rangle$ message to all active replicas informing them about the event. In order to complete the PREPARE phase, replicas participating in the agreement protocol must obtain PREPARE messages from all followers and those messages must match the leader's PREPREPARE message. In case of success, an active replica id_A creates a $\langle \text{COMMIT}, id_A, o, s, p \rangle$ message and sends it to all other active replicas in the cell. Once a replica has received matching COMMIT messages from all active replicas, agreement for request o is complete.

Note that, in contrast to PBFT, in order to successfully complete an agreement-protocol instance in REPBFT during normal-case operation, all active replicas must have provided a COMMIT message for the corresponding request. This property has two important consequences: First, if a request is committed on one active replica, it must (at least) have been prepared on all active replicas; as further discussed in Section 5.3.2, this property is crucial for ensuring safety during a protocol switch. Second, a protocol instance only makes progress as long as all active replicas behave according to specification and the messages sent reach their intended recipients. If this is not given, REPBFT switches to PBFT to ensure liveness (see Section 5.3.2).

The agreement stage of a REPBFT cell may increase throughput by running protocol instances for different client requests in parallel. As in PBFT, the number of concurrent instances W a replica participates in is limited in REPBFT to prevent a faulty leader from exhausting the space of sequence numbers: In particular, an active replica only sends own agreement messages for sequence numbers between a low water mark s_{low} and a high water mark $s_{high} = s_{low} + W$. In Section 5.3.1.3, we discuss how to advance the window defined by the water marks based on checkpoints.

5.3.1.2 Execution Stage

As a replica in PBFT, an active REPBFT replica processes client requests for which agreement has completed successfully in the order determined by their sequence numbers. In contrast to PBFT, however, an active replica id_A in REPBFT does not only send a reply v to the client after having processed a request (see Figure 5.2), but also multicasts an $\langle \text{UPDATE}, id_A, s, u, v \rangle$ message to all passive replicas in the cell; s is the agreement sequence number of the corresponding request and u is a state update reflecting the request's modifications to the service state. Having obtained at least $f + 1$ matching UPDATE messages from different active replicas, a passive replica has proof of the update's correctness. In such case, a passive replica adds the reply included in the update to its local reply cache (see Section 2.1.2.2); this step enables the replica to provide the client with a correct reply during fault handling (see Section 5.3.2.1). Furthermore, a passive replica brings its state up to speed by applying the verified state update to its local service-application instance, respecting the order of sequence numbers.

5.3.1.3 Checkpoints and Garbage Collection

In the protocol presented so far, active replicas can never be sure that their state updates have actually brought the passive replicas up to speed: For example, if the active replicas in a cell were separated from the passive replicas due to a network partition, state updates would not reach their intended recipients, leading passive replicas to fall behind without active replicas noticing. As the system must be prepared to activate passive replicas in the course of a protocol switch (see Section 5.3.2), dealing with such a scenario would require an infinite amount of memory: Active replicas would have to store all client requests as well as agreement messages in order to be able to prove to passive replicas that the requests have been agreed on. As other Byzantine fault-tolerant systems [34, 35, 41, 96, 152, 154], REPBFT addresses this problem by making use of periodic checkpoints, allowing active replicas to limit the information that needs to be kept available to messages not reflected in the latest stable checkpoint.

A checkpoint in REPBFT is reached each time a replica has processed a client request (active replicas) or applied a state update (passive replicas) whose agreement sequence number s is divisible by a system-wide constant K (e.g., 100). Having reached a checkpoint, a replica id_R multicasts a signed $\langle \text{CHECKPOINT}, id_R, s \rangle$ message to all other replicas in the cell. Note that, in contrast to existing Byzantine fault-tolerant systems [34, 35, 41, 96, 152, 154], a checkpoint in REPBFT serves primarily as a notification indicating the execution-stage progress of a replica and therefore does not require the creation of a service-application snapshot.

Checkpoints in REPBFT become stable as soon as a replica manages to assemble a checkpoint certificate that contains matching CHECKPOINT messages from all replicas in the cell; at such point, a replica stores the stable checkpoint certificate and discards older ones. In addition, an active replica advances the window for agreement-protocol instances the replica participates in (see Section 5.3.1.1) by setting the start of the window to the sequence number s of the latest stable checkpoint. Furthermore, an active replica discards all client requests and agreement messages stored that correspond to sequence

numbers of up to s. This is safe as a stable checkpoint is a proof that all replicas in the cell have at least advanced to sequence number $s+1$ and consequently will never require information about prior protocol instances.

5.3.2 Protocol Switch

The protocol REPBFT runs during normal-case operation is designed to make progress in the absence of faults (see Section 5.3.1). In the following, we present the mechanism that allows REPBFT to activate passive replicas and to perform a protocol switch to PBFT in order to ensure liveness in the presence of faults. During such a switch, replicas provide an *abort history* containing information about the current status of pending agreement-protocol instances. Based on the local abort histories of different replicas, one of the replicas, the *transition coordinator*, creates and distributes a global abort history which, once accepted and processed, ensures that replicas start PBFT in a consistent manner.

5.3.2.1 Initiating a Protocol Switch

Similar to other Byzantine fault-tolerant systems [34, 35, 41, 77, 96, 97, 152, 154], REPBFT relies on the help of clients to inform replicas about suspected or detected faults. If a client id_C is not able to obtain a verified result within a certain period of time after having issued a request o, the client multicasts a $\langle \text{PANIC}, id_C, o \rangle$ message to all replicas in the cell. Note that there could be different reasons for a result not becoming stable at a client: Amongst other things, this includes scenarios in which one or more active replicas are faulty, do not properly participate in the agreement of the request, and fail to return a correct reply, leaving the client with too few matching replies to successfully verify the result. Note that the same effect may be caused by network problems leading correct replies to be delayed or dropped.

As shown in Figure 5.3, having received a PANIC message, non-faulty REPBFT replicas prevent the system from unnecessarily abandoning normal-case operation: For example, a replica ignores a PANIC message for an old request if the same client has already issued a subsequent request (L. 3–5); ignoring old requests is possible due to the assumption that each client has at most a single outstanding request (see Section 2.1.2.1). Furthermore, replicas do not trigger a protocol switch if the request indicated by a PANIC message is already covered by the latest stable checkpoint (L. 11); instead, a replica only retransmits the corresponding reply stored in its local rely cache (L. 8). Note that in such case, a protocol switch is not necessary as the stability of the checkpoint is a prove that all replicas in the cell have obtained the correct reply to the request (see Section 5.3.1.3), either by processing the request themselves (active replicas) or by learning it in an update (passive replicas). In consequence, all non-faulty replicas will send the correct reply as a reaction to the PANIC message, eventually allowing the client to make progress.

If none of the conditions discussed above applies to the PANIC message received, a replica considers a protocol switch to be necessary. In such case, a replica forwards the PANIC message to all other replicas in the cell to ensure that they also receive the message (L. 14). Furthermore, the replica executes the transition protocol presented in the following sections, which is responsible for performing a safe switch to PBFT (L. 15).

Global data structures		
REPLYSTORE	`replies`	Reply store containing the replies to the latest requests of each client
CHECKPOINT	`checkpoint`	Latest stable checkpoint obtained by the local replica

```
1  void handlePanic(PANIC panic) {
2    /* Ignore PANIC messages for old requests. */
3    REQUESTID rid := panic.o.rid;
4    REPLY reply := replies.get(panic.id_C);
5    if(rid < reply.rid) return;

7    /* Retransmit cached reply. */
8    if(rid == reply.rid) send reply to client panic.id_C;

10   /* No protocol switch necessary if the request is covered by the latest stable checkpoint. */
11   if((rid == reply.rid) && (reply.s <= checkpoint.s)) return;

13   /* Trigger protocol switch. */
14   Forward panic to all replicas in the cell;
15   Execute transition protocol;
16 }
```

Figure 5.3: REPBFT mechanism for handling PANIC messages (pseudocode): In order to prevent the system from performing unnecessary protocol switches, non-faulty REPBFT replicas only accept PANIC messages for new requests that are not covered by the latest stable checkpoint.

5.3.2.2 Creating a Local Abort History

While running the transition protocol, a non-faulty active replica stops to participate in the agreement of requests. As a result, the agreement stage of the system does no longer make progress (see Section 5.3.1.1), which in turn allows replicas to reach a consistent state. For this purpose, at the beginning of the transition protocol, each non-faulty active replica creates a local abort history which, in a subsequent step presented in Section 5.3.2.3, will then be used to assemble a global abort history.

Similar to a view-change message in PBFT, the local abort history of a replica contains information about all client requests not covered by the latest stable checkpoint that either have or might have been processed on at least one non-faulty replica in the cell. This includes all requests (with higher agreement sequence numbers than the latest stable checkpoint) for which the local active replica has sent a COMMIT message, as such requests might have been committed and executed on other active replicas (see Section 5.3.1.1). Having identified these requests, for each of them, an active replica appends the corresponding PREPREPARE message as well as $2f$ matching PREPARE messages to its local abort history. In addition, an active replica adds the certificate (i.e., a set of matching CHECKPOINT messages, see Section 5.3.1.3) proving the validity of the latest stable checkpoint.

Once its local abort history h for a protocol p (see Section 5.3.1.1) is complete, an active replica id_A sends a $\langle \text{HISTORY}, id_A, h, p \rangle$ message to the transition coordinator. The role of the transition coordinator is assigned to one of the active replicas, for example, based on replica ids; we assume that all replicas in the cell are aware of the transition-coordinator

selection algorithm. Note that, as discussed in Section 5.3.2.4, a HISTORY message must be authenticated using a signature guaranteeing that, if one non-faulty replica accepts the message, all other non-faulty replicas also accept the message.

5.3.2.3 Creating a Global Abort History

When the transition coordinator receives the local abort history of an active replica, it only takes the history into account if the signature of the corresponding HISTORY message is valid. Based on its own local abort history as well as f valid local histories submitted by other active replicas, the transition coordinator creates a global abort history, which serves one important purpose: It allows non-faulty replicas to reach a consistent view on the progress the overall system has made prior to the start of the transition protocol. This is crucial to ensure that a key requirement for safety (see Section 2.1.1.3) is fulfilled: that each client request that has been executed on at least one non-faulty replica is also executed on all other non-faulty replicas in the cell.

Properties of a Global Abort History For REPBFT, this means that a global abort history must contain information that not only is of help to active replicas, which up this point participated in the agreement of requests, but also to passive replicas, which so far exclusively took part in the execution stage by being brought up to speed based on state updates. To this end, a global abort history provides two properties, one affecting the agreement stage (P_A) and the other being related to the execution stage (P_E):

P_A *If a request has been committed on at least one non-faulty active replica, but has not been committed on at least one other non-faulty active replica, then the global abort history contains a proof that the request has been prepared.*

P_E *If a request has been executed on at least one non-faulty active replica, but the corresponding state update has not become stable on at least one non-faulty passive replica, then the global abort history contains a proof that the request has been prepared.*

As discussed in the context of PBFT (see Section 3.1.1), the existence of a proof that a request has been prepared guarantees that no other request could have been agreed on in the corresponding protocol instance. Note that in Section 5.3.4.1 we discuss in detail why any valid global abort history in REPBFT provides the two properties. For now, we focus on why these properties are important: Property P_A targets requests for which the agreement process has been completed on some but not all non-faulty active replicas; it ensures that all non-faulty (active and passive) replicas learn about the affected request in order to prevent them from ever accepting a different request for the same agreement sequence number. Property P_E basically serves the same purpose for requests that have successfully completed the agreement stage of all active replicas but whose corresponding state updates have not been applied to all passive replicas; the property enables the non-faulty passive replicas that have not applied a state update to learn the corresponding request and thereby ensures that those replicas will not assign the particular agreement sequence number to a different request in the future.

Computing a Global Abort History Having obtained $f+1$ valid local abort histories from different active replicas, the transition coordinator starts to create the global abort history. As shown in Figure 5.4, the global abort history contains different *slots*, one for each agreement sequence number between the latest stable checkpoint (i.e., `#200`) and the newest agreement-protocol instance (i.e., `#205`) included in the local abort histories. Note that, as all active replicas in REPBFT have to participate in the agreement of requests in order for the system to make progress, in the worst case, the number of slots in the global abort history matches the size of the agreement-protocol window (see Section 5.3.1.1).
After the transition coordinator has determined which slots to add to the global abort history, it chooses the value for each slot independently according to the following rules:

H1 A request o is chosen as the slot value if one or more local abort histories contain a valid proof (in the form of PREPREPARE and PREPARE messages, see Section 5.3.2.2) that request o has been successfully prepared (e.g., slot `#201`).

H2 If rule H1 does not apply, the slot value is set to a special *null request* $\perp$ that corresponds to a no-op at execution time. This rule either takes effect if none of the local abort histories contains a proof for the slot (as in slot `#202`) or if all proofs available for the slot are invalid (as in slot `#204`). In both cases, no request could have been committed in the corresponding agreement-protocol instance: As at least one of the $f+1$ local abort histories must have been provided by a non-faulty active replica, this replica would have included a valid proof if such a request existed.

When the global abort history h_{global} for a protocol p is complete, the transition coordinator id_{TC} multicasts a $\langle \text{SWITCH}, id_{TC}, h_{global}, \mathcal{H}_{local}, p \rangle$ message to all replicas in the cell; $\mathcal{H}_{local}$ is the set of $f+1$ local abort histories that served as basis for the global history.

5.3.2.4 Processing a Global Abort History

Having obtained a global abort history, both active and passive replicas verify its correctness by reproducing the steps performed by the transition coordinator to create the history (see Section 5.3.2.3). In order to be able to do this properly, the local abort histories included must have been authenticated using a signature (see Section 5.3.2.2) as otherwise non-faulty replicas might reject the global history due to being based on too few valid local histories, even if the global history has been created by a non-faulty transition coordinator.
If the global abort history has been successfully verified, a replica (this includes the transition coordinator) uses the history to complete the switch to PBFT, which is initialized with a view in which the transition coordinator serves as leader. In particular, a replica creates a new PBFT instance for each slot contained in the global abort history: If the slot value is a regular request, the replica is bound to ensure that this request will be the result of the PBFT instance; that is, if the replica is the PBFT leader, it must send a PREPREPARE message for this request, and if the replica is a PBFT follower, it is only allowed to send a matching PREPARE message in case the leader has proposed this request.

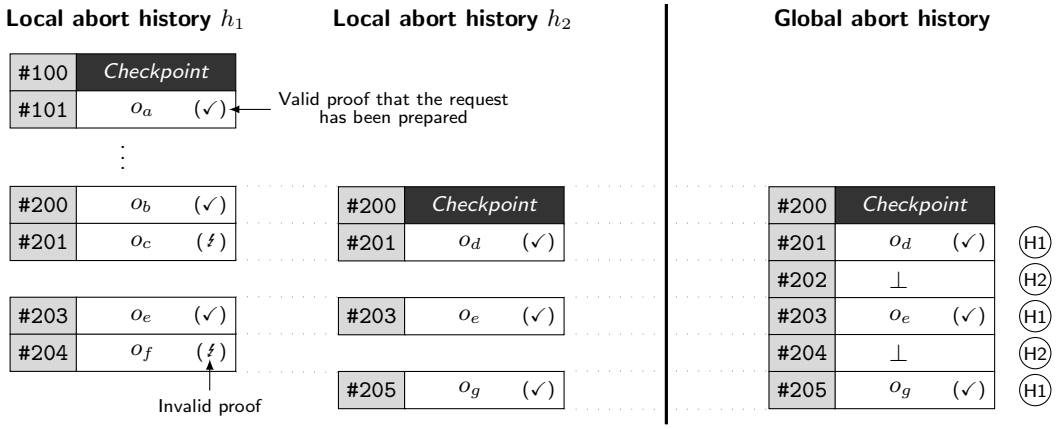


Figure 5.4: Example of how a global abort history is created in a REPBFT cell that is able to tolerate one fault: Using the rules H1 and H2, the value of each slot is determined based on the values of the respective slots in the local abort histories provided by different active replicas; once complete, each slot of the global history either contains a regular request o or a null request $\perp$.

In contrast, a slot in the global abort history containing a null request does not impose such restrictions on the corresponding PBFT instance, as a null request indicates that no regular request with this particular agreement sequence number could have previously been committed.
Having processed the global abort history, a replica starts to participate in the new PBFT instances created, thereby completing the protocol switch. Note that, at this point, replicas that have been passive during REPBFT's normal-case protocol can be considered activated as, from then on, they are no longer brought up to speed by state updates, but instead execute client requests themselves.

5.3.2.5 Handling Faults during the Protocol Switch

In case the default transition coordinator is faulty, it might fail to deliver a valid global abort history. To address such and related problems, replicas protect the protocol switch with a timeout that expires if a replica is not able to obtain a valid global abort history within a certain period of time; similar to the timeout protecting a leader change in PBFT, the timeout should be long enough to make it unlikely that a non-faulty transition coordinator is wrongfully accused of being faulty.
When the timeout expires, an active replica id_A changes its current protocol id from p to p' and sends its local abort history h in a $\langle \text{HISTORY}, id_A, h, p' \rangle$ message to the transition coordinator of p', which is different from the transition coordinator of p, but appointed based on the same deterministic selection algorithm (see Section 5.3.2.2). In addition, the replica sets up a new timeout to twice the length of the previous timeout. As in PBFT's leader-change protocol, if this timeout also expires, the procedure is retried (possibly multiple times) until the switch completed successfully thanks to one of the at least $f + 1$ non-faulty active replicas serving as transition coordinator.

5.3.3 Running PBFT

After the protocol switch to PBFT has been completed successfully, a REPBFT cell is able to tolerate up to f faulty replicas. As discussed in Section 5.3.2.4, as a result of the protocol switch, the agreement of client requests that have not been covered by the latest stable checkpoint is repeated in PBFT. Note that this process is limited to the agreement stage and does not lead to client requests being executed more than once: Based on the agreement sequence numbers assigned to requests, which remain the same for requests that have been committed on at least one replica (see Section 5.3.2.3), the execution stage is able to identify and consequently ignore such requests.
Having started to run the PBFT protocol, the next steps depend on the fault assumptions made for a particular REPBFT setting: If faults are expected to be of permanent nature, a REPBFT cell may be configured to stick with PBFT once having switched to it. In contrast, in order to tolerate temporary replica faults and/or network problems, the system could be configured to execute only a certain number of PBFT instances [77] before switching back to REPBFT's normal-case protocol: During such a transition, the designated active replicas start to provide state updates to the designated passive replicas while those are still participating in the agreement of client requests. As soon as the first REPBFT checkpoint is reached, the system fully switches back to its normal-case protocol with replicas resuming their particular roles.

5.3.4 Safety and Liveness

In the following, we discuss the decisive properties of a global abort history enabling REPBFT to safely substitute its protocol while switching from normal-case operation mode to fault-handling mode. Furthermore, we explain why such a switch will eventually be completed successfully, even in the presence of faulty replicas.

5.3.4.1 Properties of a Global Abort History

The rules applied by the transition coordinator to create the global abort history in REPBFT (see Section 5.3.2.3) are the same rules used in PBFT to determine the contents of a new-view message during leader change. The main difference, however, is that a transition coordinator in REPBFT only requires $f + 1$ local abort histories to create a global abort history while in PBFT a new-view message is based on $2f + 1$ view-change messages. The reduction in REPBFT is possible as a transition coordinator exclusively accepts local abort histories from active replicas: With the normal-case protocol only making progress if all active replicas participate accordingly, it is guaranteed that each subset of $f + 1$ local abort histories contains at least one history provided by a non-faulty active replica that has seen the latest agreement state. In contrast, up to f non-faulty replicas in PBFT might put out a view-change message without having participated in the agreement-protocol instances in question, forcing PBFT to increase the threshold by this number of view-change messages.

Note that the fact that a global abort history in REPBFT is created based on at least one local history of a non-faulty active replica id_A is crucial: Applying rule H1 to such a local history guarantees property P_A of a global abort history (see Section 5.3.2.3): If a request o has been committed on one or more non-faulty active replicas, the request must at least have been prepared on replica id_A; in consequence, replica id_A inserts a valid proof for request o into its local abort history, which will eventually be also included in the global abort history due to rule H1. The only scenario in which replica id_A may not supply a proof for request o is if the request is covered by a stable checkpoint. However, in such case all replicas in the cell must have either confirmed to have executed the request (active replicas) or to have applied the corresponding state update (passive replicas), otherwise the checkpoint would not have become stable. For the same reason, a global abort history is able to provide property P_E (see Section 5.3.2.3): With a non-faulty active replica only processing a client request after it has been committed, a request executed on at least one non-faulty active replica is guaranteed to appear in the global abort history as long as there are one or more passive replicas that have not confirmed the stability of the corresponding state update by sending a checkpoint.
In summary, REPBFT ensures that a global abort history contains a valid proof for all client requests that had an effect (either through direct execution or through application of the corresponding state update) on some, but not all, non-faulty replicas in the cell. In consequence, a global abort history allows non-faulty replicas to reach a consistent state.

5.3.4.2 Ensuring System Progress

If an active replica fails to participate in the agreement of client requests (e.g., due to having crashed) while REPBFT runs in normal-case operation mode, the agreement process stops immediately (see Section 5.3.1.1). In contrast, faulty passive replicas only indirectly prevent the system from making progress: If at least one passive replica does not confirm to have reached a checkpoint, the checkpoint will not become stable. As a result, the agreement of client requests will eventually stop as active replicas are no longer able to advance the window limiting the number of concurrent protocol instances (see Section 5.3.1.3). Either way, a stopped agreement process prevents the system from executing additional client requests, which consequently forces the corresponding clients to demand a protocol switch due to lack of replies (see Section 5.3.2.1); thanks to non-faulty replicas forwarding the clients' PANIC messages, eventually all non-faulty replicas will initiate the transition protocol.
Having triggered the protocol switch locally, a transition coordinator requires $f + 1$ valid local abort histories from different active replicas to create a global abort history. As at most f of the $2f + 1$ active replicas in the cell are assumed to fail, it is guaranteed that eventually a transition coordinator has $f + 1$ or more of such local abort histories available (including its own). Furthermore, by relying on the same mechanism as PBFT to adjust timeouts [34], REPBFT ensures that the role of transition coordinator can be assigned to different replicas in case acting transition coordinators fail to provide a valid global history (see Section 5.3.2.5).

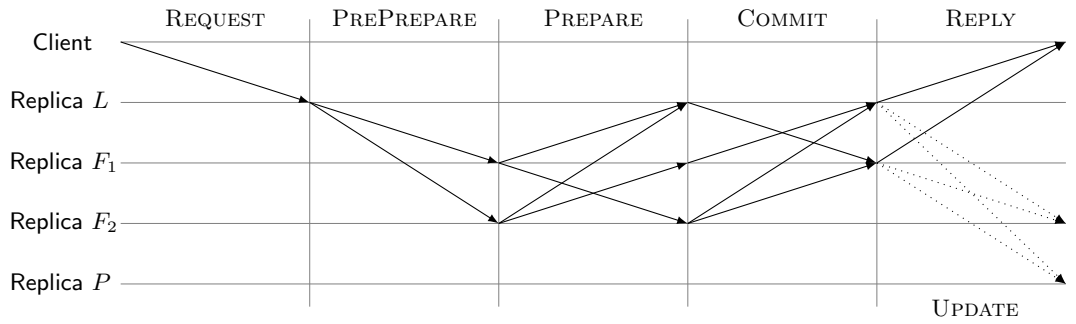


Figure 5.5: Message flow of requests, agreement messages, and replies (—) as well as state updates (····) in REPBFT* for a cell that is able to tolerate one fault: In contrast to REPBFT (see Figure 5.2), REPBFT* relies on only $f+1$ active replicas at the execution stage, instead of $2f+1$.

5.3.5 Optimizations

In the following, we outline a number of optimizations for REPBFT's normal-case protocol. Furthermore, we present a variant of REPBFT that allows additional resource savings under benign conditions.

Protocol Optimizations As a result of the close relationship between both protocols, a number of optimizations proposed for PBFT can also be applied to REPBFT's normal-case protocol. In particular, this includes the use of batches which allow active replicas to agree on multiple client requests within a single protocol instance (see Section 2.1.2.2). Furthermore, in order to reduce the amount of data to be sent over the network, PREPARE and COMMIT messages may comprise a hash of the client request instead of the full request. Besides, verification of replies and state updates can also be based on message hashes (see Section 2.1.1.4) by requiring one replica to send a full version while other replicas only provide a hash computed over the reply/update.

REPBFT* In the basic REPBFT approach presented in Section 5.3.1, all of the $2f+1$ replicas that are active at the agreement stage are also active at the execution stage. As discussed in the context of PBFT (see Section 3.1.2), $f+1$ execution replicas are sufficient to make progress under benign conditions. Therefore, to further reduce resource usage, it is possible to decrease the number of active replicas at the execution stage (see Figure 5.5), a variant which we refer to as REPBFT*: With each request being processed on $f+1$ active replicas, during normal-case operation, REPBFT* clients are able to verify the results to their requests; the same is true for passive replicas with regard to state updates. As a result of the fact that some replicas take part in the agreement but not the execution of requests (i.e., replica F_2 in Figure 5.5), there is no need to provide them with COMMIT messages, thereby saving additional network resources. Due to COMMIT messages not being relevant during a protocol switch (see Section 5.3.2), such an optimization does not prevent the affected replicas from participating in the transition protocol.

5.4 Resource-efficient Agreement and Execution based on MinBFT

In this section, we present REMINBFT, a REBFT variant that uses the MinBFT [154] agreement protocol to make progress in the presence of suspected or detected faults. As MinBFT, REMINBFT relies on a trusted service to authenticate agreement-protocol messages and consequently requires one protocol phase less than REPBFT and a total of $2f+1$ replicas to tolerate up to f faults (see Section 2.2.1). However, during normal-case operation, only $f+1$ of the replicas remain active. Below, we discuss assumptions on the trusted message certification service as well as REMINBFT's protocols for saving resources and for performing the switch to MinBFT. Note that, as REMINBFT and REPBFT share many similarities, in this section, we focus on aspects that are specific to REMINBFT and have not already been presented in Section 5.3 in the context of REPBFT.

5.4.1 Message Certification Service

As discussed in Section 2.2.1, in order to be able to reduce the minimum number of replicas required in a cell to $2f+1$, a faulty replica must be prevented from successfully performing equivocation [41]; that is, a replica must not be able to send messages with the same identity but different contents without being detected. Similar to MinBFT, REMINBFT addresses this problem by relying on a trusted service for certifying messages. We assume that each replica has a local instance of the service at its disposal that only fails by crashing. However, we do not impose any restrictions on how the message certification service is implemented as long as it provides the interface presented in Figure 5.6 as well as the functionality discussed below. Our prototype implementation of REMINBFT (see Section 5.5.1), for example, uses a special-purpose hardware component; note that this component is not a contribution of this thesis.

Message Certificates The main tasks of the trusted service are the creation and verification of message certificates $\langle \text{CERTIFICATE}, id_{MCS}, c, proof \rangle$; id_{MCS} is the id of the service instance that created the certificate, c is a counter value (see below), and $proof$ is a cryptographically protected proof linking the certificate to the corresponding message m. Note that one possibility to create $proof$ is to compute a hash-based message authentication code [98] over m, id_{MCS}, c, and a secret key that is only known to instances of the message certification service but not to any other components in the system [86, 154]. Having received a certified message, a replica only accepts the message if it matches its certificate; otherwise, the message is discarded and will not be processed.
In order to be used to prevent equivocation in REMINBFT, the message certificates created by an instance of the message certification service must provide the following property: If c_1 is the counter value included in a certificate created by the instance and c_2 is the counter value included in the *next* certificate created by the same instance, then $c_2 = c_1 + 1$. This means that an instance must guarantee to never use the same counter value twice, to always assign monotonically increasing counter values, and to not leave any gaps between the counter values of two subsequent certificates [108]. Note that such

```
1 /* Certificate creation */
2 MESSAGECERTIFICATE createCertificate(MESSAGE message);

4 /* Certificate verification */
5 boolean checkCertificate(MESSAGE message, MESSAGECERTIFICATE certificate);
```

Figure 5.6: Interface of REMINBFT's trusted message certification service (pseudocode): The `createCertificate` method allows a replica to obtain a certificate for a message to be sent. Having received a message, a replica can verify it by invoking the `checkCertificate` method.

a requirement makes it necessary for the system component implementing a service instance to be tamperproof, thereby ensuring that no external entity, not even the local REMINBFT replica, is able to reset the counter while the system is running.

Preventing Equivocation Requiring replicas to provide message certificates forces a faulty replica trying to perform equivocation to create multiple certificates if it wants to send messages with the same identity but different contents. Non-faulty replicas are able to protect themselves against such an attempt by following a simple rule: A non-faulty replica must process the messages received from another replica in the order determined by the counter values in their respective certificates; if the sequence of messages contains a gap, the replica must wait until the corresponding message becomes available. This way, non-faulty replicas either process all messages put out by a faulty replica in the same order (and consequently make consistent decisions) or, in case the faulty replica refuses to sent all messages to all replicas, they stop to process the faulty replica's messages due to detecting gaps. Either way, a faulty replica is not able to lead non-faulty replicas into performing inconsistent actions.
Note that, as further discussed in Section 5.4.2, some types of messages in REMINBFT are not authenticated using the message certification service and therefore do not carry a certified counter value. For such messages, there are no restrictions on the order in which they are allowed to be processed.

5.4.2 Normal-case Operation

During normal-case operation, REMINBFT executes the protocol shown in Figure 5.7: Of the $2f + 1$ replicas in the cell, $f + 1$ actively participate in system operations (i.e., one leader replica and f follower replicas) while the other f remain passive.

Agreement Stage Having received a request o, the leader id_L proposes it by sending a $\langle \text{PREPARE}, id_L, o, s, p \rangle_{cert_L}$ message to all followers; s is the agreement sequence number id_L has assigned to the request, p is the current protocol id, and $cert_L$ is a certificate created by the message certification service (see Section 5.4.1) covering the entire PREPARE message. When a follower id_F accepts the proposal of the leader, it notifies all active

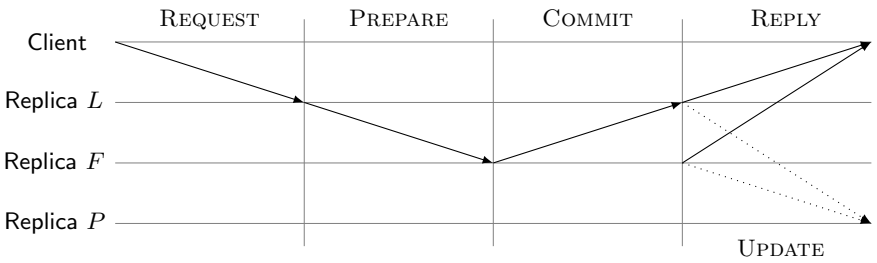


Figure 5.7: Message flow of requests, agreement messages, and replies (—) as well as state updates (····) in REMINBFT for a cell that is able to tolerate one fault: Only the leader replica L and the follower replica F participate in the agreement and execution of client requests. In contrast, the passive replica P is brought up to speed by state updates.

replicas in the cell by multicasting a $\langle \text{COMMIT}, id_F, o, s, p, cert_L \rangle_{cert_F}$ message, which is also authenticated by the message certification service. Having obtained a PREPARE message and f matching COMMIT messages, the agreement process of a request is complete; in such case, an active replica forwards the request to the execution stage.

As REPBFT (see Section 5.3.1.1), REMINBFT uses a window-based mechanism to limit the number of concurrent agreement-protocol instances. However, REMINBFT poses an additional requirement: A replica may only send a PREPARE/COMMIT message for an instance, if it has already sent a PREPARE/COMMIT message for all preceding instances. Note that this rule still allows different protocol instances to overlap. However, it forces a replica to process them in the order of their sequence numbers, which enables the replica to account for a continuous sequence of certified messages during a protocol switch (see Section 5.4.3). If a faulty replica fails to meet this requirement, all non-faulty replicas stop to process the faulty replica's messages as soon as they detect the gap. As in REPBFT, a protocol switch will be performed in such case, allowing the system to make progress.

Execution Stage Similar to REPBFT (see Section 5.3.1.2), active replicas in REMINBFT bring passive replicas up to speed via state updates. A passive replica only applies a state update after having received matching versions from all active replicas. State updates in REMINBFT are authenticated as in REPBFT; they do not contain certificates created by the message certification service, as sending a wrong update has the same effect as sending no update at all: the update will not become stable at the affected passive replica.

Checkpoints and Garbage Collection REMINBFT relies on periodic checkpoints to perform garbage collection: As in REPBFT (see Section 5.3.1.3), a checkpoint in REMINBFT becomes stable as soon as all replicas in the cell, both active and passive ones, have distributed CHECKPOINT messages indicating that they have made the same progress at the execution stage. For the same reason as state updates (see above), CHECKPOINT messages are not authenticated using the message certification service.

In contrast to REPBFT, CHECKPOINT messages of active replicas in REMINBFT also contain a set of counter values: For each active replica, this set contains the counter value assigned to the agreement message (i.e., the PREPARE message in case of the leader or the COMMIT message in case of a follower) the replica has sent in the protocol instance of the request that triggered the checkpoint. In Section 5.4.3, we discuss how these counter values allow passive replicas to be activated in the course of a protocol switch. Note that a checkpoint only becomes stable if the counter values included in the CHECKPOINT messages of active replicas also match.

5.4.3 Protocol Switch

Similar to REPBFT, the protocol REMINBFT executes during normal-case operation (see Section 5.4.2) exclusively makes progress under benign conditions, requiring a transition protocol that safely switches to a more resilient protocol, in this case MinBFT. Note that, with regard to such a procedure, there is a key difference between REPBFT and REMINBFT: While at least $f + 1$ of the $2f + 1$ active replicas in REPBFT are assumed to be non-faulty, there might only be a single such replica in REMINBFT if the other f active replicas fail. As a consequence, the mechanism for performing a protocol switch cannot be based on information provided by different replicas. In the following, we discuss how this problem is addressed by REMINBFT's transition protocol.

Initiating a Protocol Switch REMINBFT relies on the same mechanism as REPBFT to initiate a protocol switch: If an active replica, after having been notified by a client, decides that a protocol switch is necessary (see Section 5.3.2.1), the replica stops participating in the agreement of requests and informs the other replicas in the cell. In addition, an active replica also forwards the latest stable checkpoint certificate to all passive replicas, followed by all certified messages it has sent since the generation of the checkpoint; as discussed below, this step later allows passive replicas to complete the switch to MinBFT.

Creating an Abort History Having initiated a protocol switch, only a single active replica in REMINBFT creates and distributes an abort history: the transition coordinator (see Section 5.3.2.2). In contrast to REPBFT (see Section 5.3.2.4), the (global) abort history of the transition coordinator in REMINBFT is only based on local knowledge and does not include any information provided by other active replicas in the form of local abort histories; note that this approach is a consequence of the fact that, as discussed above, in the worst case of f active replicas failing, the transition coordinator is the only remaining non-faulty replica that has participated in the agreement of client requests.
In order to create an abort history, the transition coordinator in REMINBFT performs the following steps: First, it adds the latest stable checkpoint certificate to the history. Next, the transition coordinator includes all certified messages it has sent since the point in time reflected by latest stable checkpoint. When the abort history h for a protocol p is complete, the transition coordinator id_{TC} multicasts a $\langle \text{SWITCH}, id_{TC}, h, p \rangle_{cert_{TC}}$ message to all replicas in the cell; $cert_{TC}$ is a certificate created by the message certification service for the SWITCH message.

Note that a valid abort history provides a continuous sequence with regard to the counter values of certified messages: The sequence starts with the transition coordinator's counter value in the stable checkpoint certificate (see Section 5.4.2) and ends with the counter value of the SWITCH message's certificate. As a non-faulty transition coordinator includes all certified messages in between, the sequence of counter values does not contain any gaps; an abort history that contains gaps is invalid and will not be processed by non-faulty replicas. As a result, a valid abort history necessarily contains information about all the agreement-protocol instances the transition coordinator has participated in, which allows other replicas to reach a consistent state by processing the abort history.

Processing an Abort History When a non-faulty (active or passive) replica receives a valid abort history from the transition coordinator, the replica initializes MinBFT with a view in which the transition coordinator serves as leader. For each valid agreement message in the abort history, the replica creates a new MinBFT instance and only accepts the client request included in the agreement message as the outcome of this particular instance, similar to the procedure in REPBFT (see Section 5.3.2.4).
Note that in order for passive replicas in REMINBFT to complete a protocol switch additional measures need to be taken: While the system runs the normal-case protocol, passive replicas do not receive any certified messages from active replicas. As a result, without intervention, passive replicas would not process the first certified message they receive, which is the abort history, as from their point of view the abort history does not contain the next counter value expected from the transition coordinator. To address this problem, passive replicas rely on the verified counter-value information included in stable checkpoints (see Section 5.4.2) to update their expectations on counter values. This way, when active replicas provide the latest stable checkpoints as well as subsequent certified messages at the start of the protocol switch (see above), passive replicas are able to join the agreement-stage communication.

Handling Faults during the Protocol Switch & Running MinBFT In case the default transition coordinator fails to deliver a valid abort history, a similar timeout-based mechanism as in REPBFT (see Section 5.3.2.5) is used to reassign the role of transition coordinator to another replica. Having eventually switched to MinBFT, the system may return to executing the normal-case protocol at a later point in time (see Section 5.3.3).

5.4.4 Safety and Liveness

In the following. we address safety and liveness aspects in REMINBFT. As in previous sections, we focus our discussion on the main differences to REPBFT (see Section 5.3.4.2).

Properties of an Abort History Although being created based on local information of a single active replica (i.e., the transition coordinator), an abort history in REMINBFT provides similar properties as the global abort history in REPBFT (see Section 5.3.4.1): If a client request has been committed on at least one non-faulty active replica in the cell, but not on others, the request is guaranteed to be included in a valid abort history of the

transition coordinator. This is due to the fact that, in order for a request to be committed, the transition coordinator must have either provided a valid PREPARE message (if it has been the leader) or a valid COMMIT message (if it has been a follower) for the request. In consequence, the only way for the transition coordinator to create a valid abort history is to add this agreement-protocol message to its history (see Section 5.4.3).
Besides, a valid abort history in REMINBFT is also guaranteed to contain all requests that have been executed on at least one non-faulty active replica but whose corresponding state updates have not become stable at one or more non-faulty passive replicas: With confirmation from all replicas in the cell being required for a checkpoint to become stable (see Section 5.4.2), such requests can not be covered by the latest stable checkpoint. Therefore, as the request in question must have been committed on at least one non-faulty replica, it is guaranteed to be included in a valid abort history (see above).

Ensuring System Progress Relying on the same mechanism as REPBFT to initiate a protocol switch (see Section 5.4.3), all non-faulty replicas in REMINBFT will eventually start the transition to MinBFT if the normal-case protocol does no longer make progress (see Section 5.3.4.2). During the switch, a faulty transition coordinator may require the coordinator role to be assigned to a different active replica (see Section 5.4.3). However, as at most f of the $f + 1$ active replicas are assumed to be faulty, there is at least one non-faulty active replica in the cell that provides a valid abort history while serving as transition coordinator, allowing all non-faulty replicas to complete the switch to MinBFT.

5.5 Evaluation

In this section, we discuss measurement results for the normal-case operation protocols of REPBFT and REMINBFT. Furthermore, for both systems, we present experiments that evaluate the protocol switch to PBFT and MinBFT, respectively. Having already evaluated the impact of passive replication on the execution stage in the context of SPARE (see Section 4.11), in the following we focus on the agreement stage.

5.5.1 Environment and Experiments

The experiments shown in this section are conducted using a replica cluster of 8-core servers (2.3 GHz, 8 GB RAM) and a client cluster of 12-core machines (2.4 GHz, 24 GB RAM); all servers are connected with switched Gigabit Ethernet. For comparison, we not only evaluate the performance and resource (i.e., CPU and network) usage of REPBFT and REMINBFT but also repeat the experiments for PBFT and MinBFT. In order to be able to focus on the differences between the four agreement protocols, all prototypes share as much code as possible; that is, we do not use the original implementations of these systems [34, 154]. Requiring a trusted message certification service, the prototypes of MinBFT and REMINBFT rely on the FPGA-based CASH subsystem [86] for this purpose. In all cases, the systems evaluated are dimensioned to be resilient against one Byzantine fault. As a consequence, the cells of PBFT and REPBFT comprise four replicas whereas the cells of MinBFT and REMINBFT comprise three replicas.

In our experiments, we run two benchmarks that are commonly used [34, 41, 77, 96, 109, 152, 153, 154, 161] to evaluate Byzantine fault-tolerant agreement protocols: a *0/4 benchmark*, in which clients repeatedly send requests with empty payloads to the service and receive replies with four-kilobyte payloads, and a *4/0 benchmark*, in which request payloads are of size four kilobytes and the payloads of replies are empty. While the 0/4 benchmark is designed to represent use cases in which clients read data from the service, the 4/0 benchmark models a workload that consists of write requests. Note that, as in both cases processing a request equals a no-op, the benchmarks require minimal involvement of the execution stage and are consequently effective means to evaluate differences in agreement protocols.

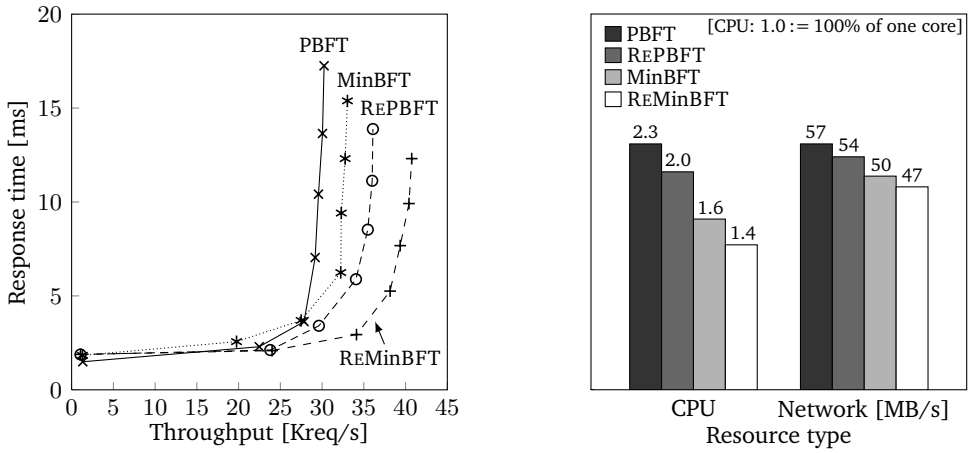
5.5.2 Normal-case Operation

In the following, we present results of the 0/4 and 4/0 benchmarks for PBFT, REPBFT, MinBFT, and REMINBFT for system operations in the absence of faults. Under such conditions, REPBFT and REMINBFT execute their protocols for the normal case presented in Sections 5.3.1 and 5.4.2, respectively. Note that, in the experiments in Sections 5.5.2.1 and 5.5.2.2, the execution of a request in REPBFT and REMINBFT leads to an empty state update being generated and sent to the passive replica. We evaluate the impact of different state-update sizes in Section 5.5.2.3.

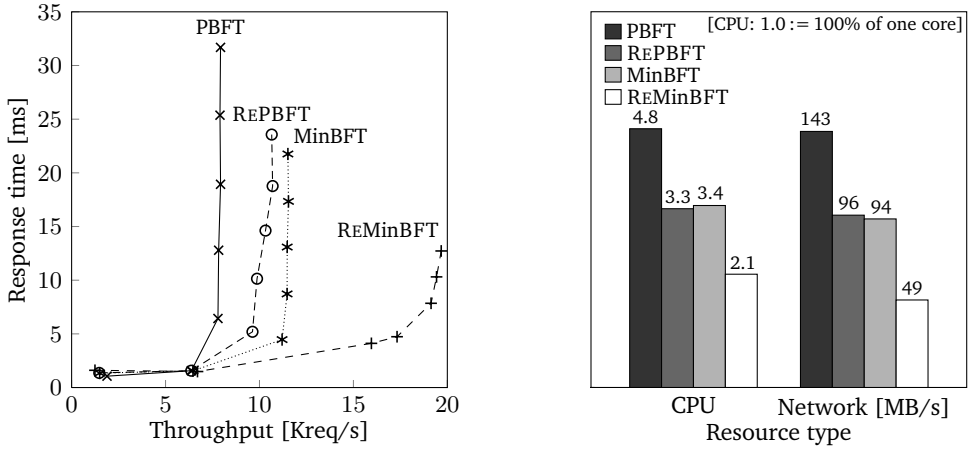
5.5.2.1 0/4 Benchmark

Figure 5.8a shows the performance and resource-usage results for the 0/4 benchmark gained by stepwise increasing the number of clients that concurrently use the service to 500. The numbers for CPU and network usage represent an aggregation of the resource consumption of all replicas in the cell at maximum throughput. For better comparison, the numbers are normalized to a throughput of 10,000 requests per second.

Performance With clients issuing requests with empty payloads, a major factor influencing performance in this benchmark is the replicas' task to send replies. As a consequence, the fact that MinBFT and REMINBFT comprise one protocol phase less than PBFT and REPBFT (compare Sections 5.3.1 and 5.4.2) only results in minor differences in maximum throughput (i.e., 9% for MinBFT over PBFT and 12% for REMINBFT over REPBFT). However, comparing the systems that are directly related to each other, our experiments show that by requiring less agreement-protocol messages than their counterparts, REPBFT and REMINBFT allow throughput increases of 19% and 23% over PBFT and MinBFT, respectively.
Note that the benefits of needing to authenticate less agreement-protocol messages can be illustrated focusing on a comparison between MinBFT and REMINBFT: For this experiment, the limiting factor in MinBFT is the access to the FPGA providing the trusted message certification service; as a result, MinBFT's maximum throughput is even lower



(a) Results for the 0/4 benchmark with empty client requests and four-kilobyte replies



(b) Results for the 4/0 benchmark with four-kilobyte client requests and empty replies

Figure 5.8: Measurement results of the 0/4 and 4/0 benchmarks for PBFT, RePBFT, MinBFT, and ReMinBFT: Executing normal-case operation protocols that are designed for benign conditions, RePBFT and ReMinBFT are not only able to save resources in the absence of faults but also achieve better performance compared with their respective counterparts PBFT and MinBFT.

than the maximum throughput of RePBFT. In contrast, due to ReMinBFT only creating/verifying certificates for two (i.e., the Prepare and one Commit) messages per agreement-protocol instance instead of three as MinBFT (i.e., the Prepare and two Commits), ReMinBFT achieves a higher maximum throughput than MinBFT.

Resource Usage Applying the optimization discussed in Section 5.3.5, all four systems evaluated reduce the amount of data to be sent over the network by making only one (active) replica send a full reply while all others respond with hashes; different replicas send the full replies for different requests. Nevertheless, the need to send full replies,

combined with the fact that for this benchmark replies are much larger than agreement-protocol messages, results in RePBFT (ReMinBFT) replicas transmitting moderate 5% less data over the network than PBFT (MinBFT) replicas. With regard to CPU usage, the savings achieved by RePBFT and ReMinBFT are higher: Relying on a passive replica, which neither participates in the agreement protocol nor sends replies, allows a reduction in overall CPU usage by 11% and 15% compared with PBFT and MinBFT, respectively.

5.5.2.2 4/0 Benchmark

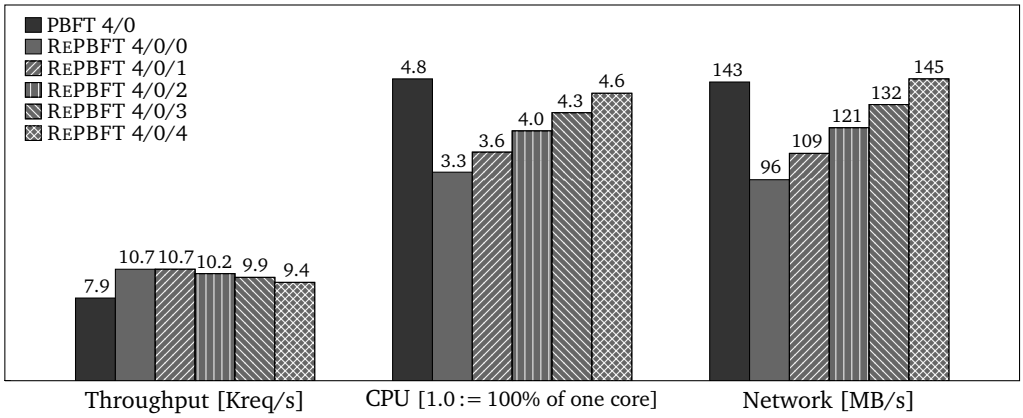
In contrast to the 0/4 benchmark, in which replies represent the decisive factor, the 4/0 benchmark is dominated by client requests. Figure 5.8b shows the measurement results for performance (using up to 250 clients) and resource usage for this use case.

Performance For the 4/0 benchmark, the maximum throughput achievable by a system depends on the number of follower replicas participating in the agreement protocol: Having received the request from the client, the leader in PBFT proposes the request to three followers, thereby saturating its network connection at an overall throughput of less than 8,000 client requests per second. In contrast, the leader in RePBFT (see Section 5.3.1) and MinBFT distributes each request to only two followers, allowing these systems to achieve a higher throughput of about 10,700 and 11,500 requests per second, respectively; the difference in maximum throughput between both systems illustrates the overhead of RePBFT's more complex agreement protocol. Finally, with the ReMinBFT leader forwarding client requests to a single active follower (see Section 5.4.2), ReMinBFT is able to realize a maximum throughput of about 19,700 requests per second, an increase of 71% compared with MinBFT.
Note that a possible way to increase throughput for the 4/0 benchmark would be to make clients send their requests to all active replicas instead of only the leader [34, 154]. This way, it would be sufficient for the leader to propose a request hash to followers instead of the full request. However, applying such an approach introduces a new problem: By sending requests with the same id but different contents to different replicas, a faulty client may cause non-faulty replicas to disagree on what the actual request is. As pointed out by Clement et al. [44] for a similar issue, such a scenario must be prevented by all means; otherwise malicious clients could cause major service disruptions.

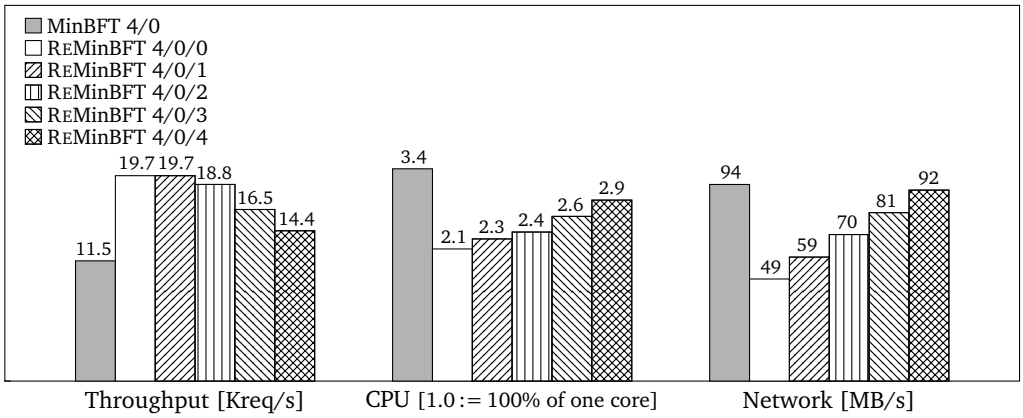
Resource Usage The CPU and network usage results for the 4/0 benchmark show that the introduction of a passive replica, which allows the leader to propose each request to one less replica, also has a significant impact on resource consumption: Compared with PBFT, RePBFT uses 31% less CPU and sends 33% less data over the network. For ReMinBFT, the savings over MinBFT are 38% (CPU) and 48% (network), respectively.

5.5.2.3 Impact of State Updates

In this section, we investigate the impact of state-update sizes on the performance and resource usage of RePBFT and ReMinBFT by extending the 4/0 benchmark of Section 5.5.2.2 to a *4/0/z benchmark*, with z indicating the payload size of state updates in



(a) Impact of the state-update size in REPBFT



(b) Impact of the state-update size in REMINBFT

Figure 5.9: Throughput and resource-usage results for different state-update sizes in REPBFT and REMINBFT: With the size of state updates increasing, CPU and network usage also increases, resulting in a decrease in system throughput. However, even for state updates of four kilobytes, REPBFT and REMINBFT achieve a higher throughput than PBFT and MinBFT, respectively.

kilobytes. Note that, modeling a write workload (see Section 5.5.1), the 4/0 benchmark is a suitable candidate for this purpose; in contrast, a read-only workload, as represented by the 0/4 benchmark, does not lead to state modifications that would have to be applied to passive replicas. In all the experiments, only one active replica sends the full state update to the passive replica, while the other active replicas in the cell provide hashes (see Section 5.3.5). Drawing from the insight gained in Section 5.5.2.2 that the network connection of the leader replica is a bottleneck for the 4/0 benchmark, we configure the leader in REPBFT and REMINBFT to always send state-update hashes.

Figure 5.9 presents the maximum throughput achieved for different state-update sizes between zero and four kilobytes as well as the impact on CPU and network usage. The results show that increasing the size of state updates from zero to one kilobyte has no

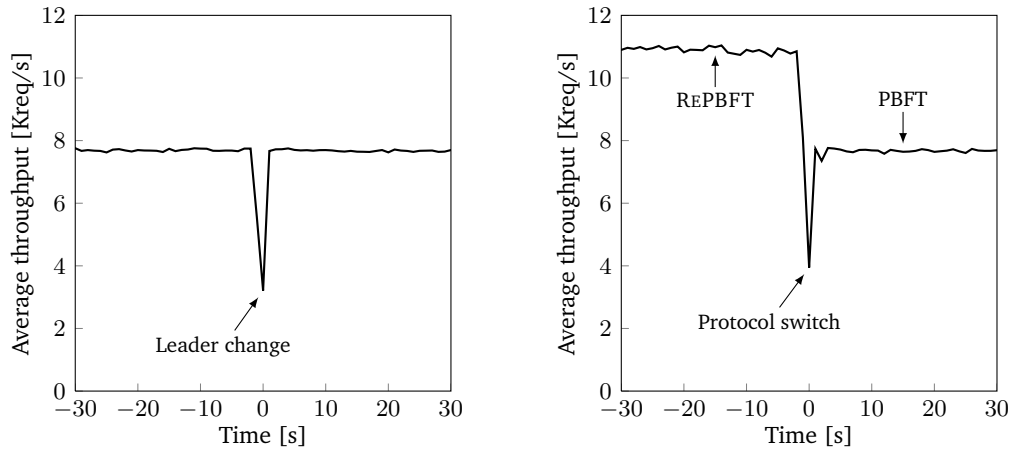
observable effect on the overall throughput of RePBFT and ReMinBFT. Note that this observation is important with regard to possible use-case scenarios that have similar characteristics as the RUBiS service evaluated in Section 4.11.3.3 in the context of Spare: For RUBiS, we found requests and replies on average to be 13 times larger than the corresponding state updates. Translated to the 4/0/z benchmark, such applications fall in the (sub) 4/0/1 category for which RePBFT and ReMinBFT show similar performance results as for the 4/0/0 benchmarks with empty state updates; nevertheless, small non-empty state updates come with additional overhead in terms of CPU and network usage. With the size of state updates increasing, more resources are consumed in RePBFT and ReMinBFT, eventually reaching similar (network) or slightly lower (CPU) levels than in PBFT and MinBFT. However, even for state updates of four kilobytes, RePBFT and ReMinBFT achieve a 19% and 25% higher throughput compared with PBFT and MinBFT, respectively. This is a direct result of the optimization discussed above, which allows the leader replica to only provide hashes for state updates, thereby avoiding additional pressure on the leader's network connection.

5.5.3 Fault Handling

Having been designed to save resources under benign conditions, the normal-case protocols of RePBFT and ReMinBFT do not ensure progress in the presence of faults, requiring the systems to switch to the resilient PBFT and MinBFT protocol, respectively. In the following, we evaluate the performance impact of such a protocol switch and compare it to the performance impact of a leader change in PBFT and MinBFT. For this purpose, we conduct a 4/0 benchmark experiment (see Section 5.5.2.2) in which we manipulate the leader replica to stop proposing new client requests one protocol instance short of a new checkpoint. In consequence, for a checkpoint interval of 100 (see Section 5.3.1.3), the abort histories of RePBFT and ReMinBFT comprise 99 slots (see Section 5.3.2.3).

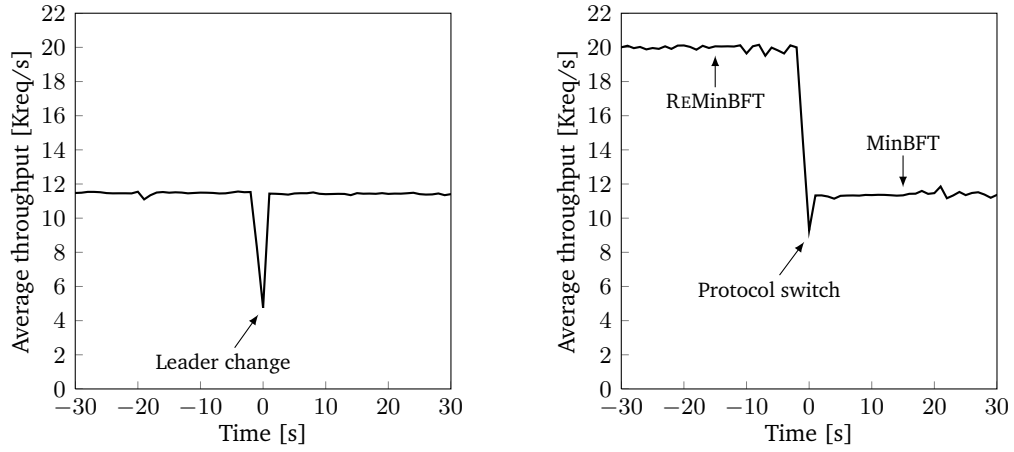
Figure 5.10b shows the overall system throughput prior, during, and after a protocol switch in RePBFT for an experiment in which 100 clients concurrently issue requests to the service. As a result of the transition protocol being executed, the throughput briefly drops to about 4,000 requests per second before stabilizing at the normal-case level for PBFT. Comparing a protocol switch in RePBFT to a leader change in PBFT (see Figure 5.10a), we can conclude that both cause a similar performance overhead: In both cases, the maximum latency experienced by a client in the experiments was less than 850 milliseconds.

In contrast, a protocol switch in ReMinBFT is more efficient than a leader change in MinBFT, as illustrated by Figures 5.10c and 5.10d for an experiment with 200 concurrent clients: While changing the leader in MinBFT (similar to the protocol switch in RePBFT, see Section 5.3.2) requires two rounds of replica communication after having been initiated, in ReMinBFT only the transition coordinator's abort history has to be distributed for the protocol switch (see Section 5.4.3). In consequence, ReMinBFT clients whose requests were affected by the reconfiguration procedure had to wait less than 700 milliseconds for their replies to become stable in our experiments.



(a) Leader change in PBFT

(b) Protocol switch in REPBFT



(c) Leader change in MinBFT

(d) Protocol switch in REMINBFT

Figure 5.10: Impact of a faulty leader replica on the throughput of PBFT, REPBFT, MinBFT, and REMINBFT for the 4/0 benchmark: As a result of the protocol switch, the throughput of REPBFT and REMINBFT drops to the normal-case levels achieved by PBFT and MinBFT, respectively.

5.5.4 Summary

The evaluation of SPARE in the previous chapter (see Section 4.11) has already shown that passive replication is an effective means to save resources at the execution stage of a Byzantine fault-tolerant system. The results of the REBFT experiments presented in this section support the conclusion that the same applies if the concept is extended to the agreement stage. In addition, the measurements revealed that by minimizing communication overhead in REBFT it is possible to not only save resources but also to improve performance without reducing fault-handling efficiency.

5.6 Discussion

In the following, we summarize the impact faulty clients and replicas can have on a REBFT system that is executing the normal-case protocol. Furthermore, we discuss different approaches to define which replicas in a cell are active and which replicas are passive. Finally, we elaborate on how the results of our evaluation can be transfered to other use-case scenarios.

5.6.1 Impact of Faulty Clients and Replicas

As discussed in Section 5.1, both the agreement stage as well as the execution stage of REBFT have been designed to save resources under benign conditions by relying on a protocol that is only able to make progress in the absence of faults. In consequence, a single faulty system component may cause the system to switch to fault-handling mode: For example, an active replica that fails to participate in the agreement of client requests stalls progress while REBFT is executing the normal-case protocol. The same is eventually true for a faulty passive replica that omits to distribute a checkpoint notification, consequently preventing the checkpoint from becoming stable (see Section 5.3.1.3); such a scenario leads to the leader replica at some point not being able to start new agreement-protocol instances due to reaching the upper end of its sequence-number window (see Section 5.3.1.1). Finally, a faulty client can trigger an unnecessary protocol switch, for example, by sending a PANIC message for a request for which it has already obtained a stable result. In comparison, apart from network problems, in PBFT and MinBFT in general only a faulty leader replica can cause the agreement process for client requests to temporarily stall and consequently force the leader role to be assigned to a different replica. In contrast, faulty follower replicas and faulty clients are not able to trigger system reconfigurations.
Note that faulty behavior of clients in REBFT however does not always lead to the system abandoning resource-saving mode: As discussed in Section 5.3.2.1, when receiving a PANIC message from a client, a replica first checks whether it has evidence (e.g., in the form of a stable checkpoint) that a protocol switch can be omitted. Only if a replica cannot guarantee that this is the case, the replica initiates the transition protocol. This way, REBFT minimizes the probability of a protocol switch being performed unnecessarily.

5.6.2 Assignment of Replica Roles

In general, REBFT makes no assumptions on how the roles of active and passive replicas are assigned as long as it is guaranteed that all non-faulty nodes in the cell consider the same subset of replicas active and the same subset of replicas passive (see Section 5.2.1). In the most simple case, replica roles are assigned statically and never change over the lifetime of the system. A possible application scenario for such an assignment could be to place passive replicas on less powerful servers whose hardware resources are not able to provide the maximum throughput of REPBFT and REMINBFT but are sufficient to run

PBFT and MinBFT during fault handling. In contrast, the roles of replicas may also be assigned dynamically, possibly selecting a different set of replicas to remain active after the system switches back to normal-case operation after fault handling (see Section 5.3.3). Such an approach might be used to allow a geo-replicated system to adapt to changing client-access characteristics, as briefly outlined in Section 7.3.
Besides dividing replicas into active and passive ones, another import role assignment in REBFT is the selection of the transition coordinator. In principle there are two different options: First, to appoint one of the active follower replicas of the normal-case protocol as transition coordinator or, second, to select the active replica that up to this point served as leader. Note that, while other Byzantine fault-tolerant protocols like PBFT and MinBFT in similar situations implement the former, the latter offers a key benefit in REBFT: If the leader role is stable across a transition from normal-case operation to fault handling, malicious clients and replicas cannot exploit the protocol-switch mechanism to force the system into taking away the leader role from a non-faulty replica. Apart from that, as any faulty component can be the cause for a transition to fault-handling mode in REBFT (see Section 5.6.1), there is no inherent reason to change the leader during this procedure.

5.6.3 Transferability of Results

Having already evaluated passive replication at the execution stage in the context of SPARE (see Section 4.11), our experiments for REPBFT and REMINBFT in Section 5.5 focused on assessing the impact of the REBFT approach on the agreement stage of a Byzantine fault-tolerant system. Note that this decision has mainly two implications with respect to the transferability of our results to other use-case scenarios: First, with the execution stage in our experiments only invoking a no-op during request processing (see Section 5.5.1), its influence on performance and resource usage has been minimal. As a consequence, we expect to see overall performance benefits and resource savings similar to those in REBFT for use cases in which the agreement stage is responsible for a major part of total service latency as, for example, in key-value stores [55, 68]. Second, due to the fact that the agreement stage of a fault and intrusion-tolerant system is agnostic to the particular service application running at the execution stage, our results for REPBFT and REMINBFT are not specific to a certain use case. Instead, a similar reduction of CPU and network usage at the agreement stage is presumably possible for all applications with asymmetric sizes of requests and replies, ranging from key-value stores over distributed file systems (e.g., NFS [145]) to coordination services [22, 30, 84].

5.7 Chapter Summary

In this chapter, we presented an approach to minimize the resource consumption of a fault and intrusion-tolerant system by relying on a normal-case operation mode in which only a subset of replicas actively participate in both the agreement stage as well as the execution stage. Introducing such an operation mode does not require a system to be redesigned from scratch. Instead, a resource-efficient protocol for the normal case can be

derived from existing Byzantine fault-tolerant agreement protocols, as illustrated in this chapter by the examples of PBFT and MinBFT. In this context, we presented REMINBFT, the first Byzantine fault-tolerant system that is able to tolerate scenarios in which all but one of the replicas active during normal-case operation become faulty.
Having so far focused on reducing the resource footprint of a fault and intrusion-tolerant system, in the next chapter, we investigate how different modes of operation can be used to improve resource efficiency if increasing performance is the primary goal. For this purpose, we draw on the insight gained in this and the previous chapter that under benign conditions it is sufficient to actively process a client request on a subset of replicas as long as additional replicas are prepared to assist in case of suspected or detected faults.

6
On-demand Replica Consistency

Byzantine fault tolerance for stateful services is usually associated with a performance penalty, meaning that a service that is resilient against arbitrary faults performs worse than its unreplicated non-fault-tolerant equivalent [34]. The rationale behind this assessment is that the need for an agreement stage, however complex or simple the protocol in use might be, inherently results in higher latency [34, 42, 96]. Furthermore, due to all execution replicas processing all client requests [34, 35, 41, 154, 161], just as the server side of the unreplicated service, the execution stage, at best, causes no additional performance overhead; at worst, the measures required to ensure determinism and consistency (e.g., sequential execution) further degrade performance.

As our main contribution in this chapter, we show that decreased performance is not an inherent property of Byzantine fault tolerance: Relying on the concept of *on-demand replica consistency* presented in this chapter, Byzantine fault-tolerant systems can be built that achieve higher throughput and lower latencies during normal-case operation than unreplicated systems providing the same services. With on-demand replica consistency, we approach the problem of resource efficiency from a different direction as in previous chapters: While SPARE and REBFT make use of trusted components to minimize the amount of resources required in the overall system, in this chapter, we investigate how to improve resource efficiency by minimizing the number of redundant request executions per replica. Applying such a technique to a stateful service results in parts of an execution replica's state becoming outdated, which is why we also present a mechanism that addresses the emerging challenge of safely and efficiently bringing execution replicas up to speed in the course of fault handling. We conclude the chapter by evaluating our approach with two case studies: a distributed file system and a coordination service.

6.1 Increasing Performance of Byzantine Fault-tolerant Services

The overhead for state-machine replication in general, and the need to keep the state of execution replicas consistent in particular, usually degrades throughput and increases latency in existing Byzantine fault-tolerant systems. In other words: Making an unreplicated service resilient against Byzantine faults not only requires more resources but also results in less performance. To address this problem, we present ODRC, a system named after one of its core concepts: on-demand replica consistency. Below, we give an overview of the key ideas behind ODRC and outline how they are implemented in the system.

6.1.1 High Performance through Resource Efficiency

Increasing the performance of a Byzantine fault-tolerant system is a problem that has been widely addressed in recent years. Most solutions developed for this purpose are aimed at increasing the throughput and/or minimizing the latency of the agreement stage: PBFT [34], for example, provides an optimization for read-only operations that reduces the number of protocol phases a client request not modifying the service state has to go through. Other systems [41, 49, 152, 154] achieve a similar effect for all requests by relying on trusted components (see Section 2.2.1) that also allow an agreement stage to reduce the number of messages that have to be sent per protocol instance. Zyzzyva [96] and Abstract [77] minimize the agreement-stage latency by running a protocol optimized for normal-case operation that requires the execution stage to process requests in an order that is speculated on.

In contrast to the systems mentioned above, ODRC focuses on the execution stage in order to increase the performance of the overall system. The rationale behind this decision is that, thanks to the previous and still ongoing intensive research on agreement protocols, for more and more services, the main factor influencing performance in a Byzantine fault-tolerant system shifts from the agreement stage to the execution stage: With agreement-stage latencies decreasing thanks to more efficient protocols, for an increasing number of use cases, request processing takes more time than request ordering. Note that ours is not the first work targeting the execution stage: CBASE [97], for example, incorporates a mechanism allowing client requests that access different parts of the service state, and therefore do not interfere with each other, to be processed concurrently. Zzyzx [79] proposes a lock-based approach for use cases in which data sharing is uncommon that temporarily gives individual clients exclusive access to certain service-state parts, running a separate protocol that bypasses the agreement stage.

Sharing one of its key ideas with SPARE and REBFT, ODRC is designed to reduce the number of executions per client request to a minimum. For this purpose, ODRC extends the concept of preferred quorums, which so far has been used in both agreement-based and quorum-based Byzantine fault-tolerant systems [1, 42, 51, 79]: Instead of relying on a preferred quorum of $2f + 1$ execution replicas, ODRC executes a request on a subset of only $f + 1$ execution replicas during normal-case operation [159]; the subset is selected based on a state-partition scheme [3, 133]. With different requests being executed on

different subsets of execution replicas, the individual load per execution replica decreases and resources become available, which in turn can be used to process more requests, resulting in an increase in overall system throughput. In contrast to ZZ [159] and SPARE, which also rely on subsets of $f + 1$ execution replicas during normal-case operation, ODRC neither requires a virtualized environment nor comprises trusted components. Furthermore, unlike Zzyzx [79], ODRC is able to also increase performance in cases where data sharing is common as it does not depend on clients locking state parts.

6.1.2 Strong Consistency

Using different subsets of execution replicas to process client requests in ODRC inevitably leads to a situation in which the overall service states of different execution replicas in the system diverge. A similar problem may arise in systems whose execution replicas do not execute requests based on a stable total order that is identical across replicas [77, 90, 96]. However, in contrast to such systems, ODRC does not require execution replicas to provide a rollback mechanism and consequently does not suffer from problems related with such a procedure including, for example, the fact that some operations cannot be undone due to involving invocations of external services. Another approach to deal with divergent execution-replica states is to relax consistency guarantees which is not only used for crash-tolerant services [55, 75, 84] but has also been proposed in the field of Byzantine fault-tolerant systems [141]. Using such a strategy, the client must be aware of the lack of strong consistency and, if necessary, be able to deal with the effects.
Acknowledging the fact that practitioners often back away from relaxed consistency semantics [32, 46], ODRC provides strong consistency by approaching the problem of divergent execution-replica states differently. A key insight in this context is that the existence of non-identical overall service states does not necessarily have to be a problem during normal-case operation: As long as client requests that read or write the same part of the service state are handled by the same subset of execution replicas, requests always access state parts that are up to date. Only in the presence of faults, when additional execution replicas are required to step in, further actions have to be taken. In such case, ODRC ensures consistency on demand, updating only the state parts needed in order for an execution replica to participate in fault handling for a particular request. A crucial property allowing ODRC to bring specific parts of an execution replica's service state up to speed is the fact that during normal-case operation state parts may become outdated but never inconsistent. As a result, ODRC is able to guarantee strong consistency without performing rollbacks [77, 90, 96] or requiring assistance from clients [141].

6.1.3 Efficient Fault Handling

As discussed in the context of SPARE and REBFT (see Sections 4.1.3 and 5.5.3, respectively), a fast reaction to faults is crucial for a Byzantine fault-tolerant system, independent of the fact that faults are in general assumed to occur rarely [78, 96, 158, 159]. Pursuing an optimistic approach that under benign conditions processes requests on only

a subset of execution replicas, ODRC faces the same challenge at the beginning of fault-handling procedures as SPARE: As discussed in Section 6.1.2, in order to be able to assist in fault handling, the service state of an execution replica first has to be brought up to date. However, note that there is an important difference between ODRC and SPARE: While SPARE relies on passive backup replicas that are kept in resource-saving mode during normal-case operation, ODRC for this purpose uses active execution replicas that are already running. In consequence, ODRC is able to save the latency and resource overhead associated with activating an execution replica.

6.2 The ODRC Architecture

In this section, we present assumptions on the environment in which ODRC is used as well as on the service applications to be integrated with the system. Besides essential requirements, we also discuss desirable characteristics that increase the extent to which a service application can benefit from ODRC.

6.2.1 Environment

Following its design goal to increase resource efficiency without requiring any trusted components (see Section 6.1.1), an instance of ODRC, which in the following is also referred to as an ODRC *cell*, regularly comprises a minimum of $3f + 1$ servers. In addition, as discussed below, an ODRC cell may be extended with additional servers in order to allow a system to scale up performance. Independent of the particular system configuration, result verification in ODRC is performed by voters at the client side.

Regular Cell In the regular configuration, as depicted in Figure 6.1, ODRC relies on $3f+1$ agreement replicas, each hosted on a different physical server, as well as $3f + 1$ execution replicas; due to the fact that ODRC only requires the agreement stage and the execution stage to be logically separate, not necessarily physically [161], an execution replica may either be integrated with an agreement replica (and consequently running in the same process), co-located on the same server (in a different process), or placed on an entirely different server. Note that, as further discussed in Section 6.6.3, the regular ODRC configuration is not minimal with regard to the number of execution replicas. However, using equal set sizes for both agreement replicas and execution replicas, as done in most Byzantine fault-tolerant systems [34, 35, 96, 97, 153], allows us to better illustrate the impact of ODRC on performance. Furthermore, in contrast to SPARE (presented in Chapter 4), ODRC's goal is efficient usage of resources, not a reduction of the amount of resources allocated.
As shown in Figure 6.1, ODRC introduces an additional stage, the *selection stage*, between agreement stage and execution stage. The selection stage consists of a set of *selector* components, one for each execution replica in the cell, and is mainly responsible for deciding which client request to process on which execution replica, as described in detail in Section 6.3. Note that selectors use the sequence of totally-ordered client requests

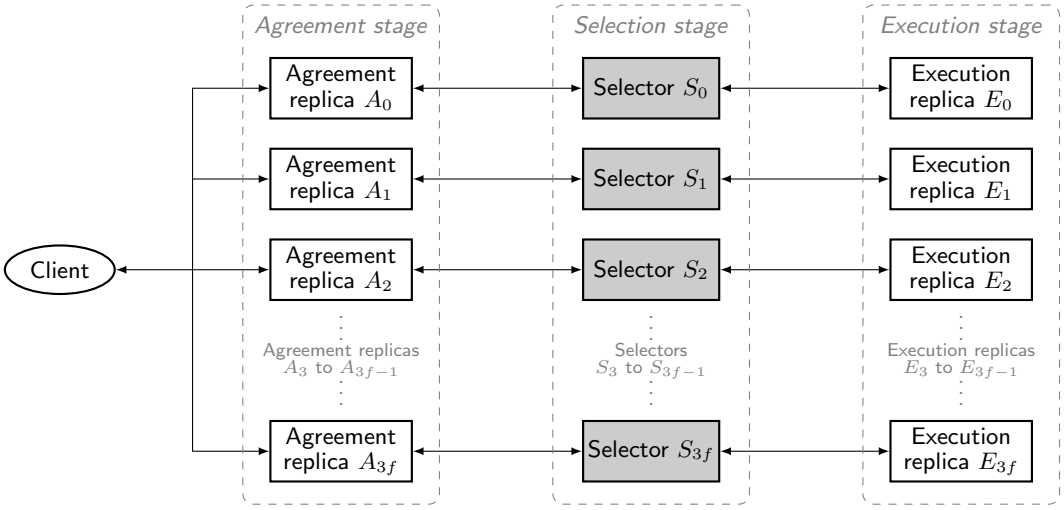


Figure 6.1: Overview of the ODRC architecture: To increase performance by improving resource efficiency, ODRC introduces a new stage, the *selection stage*, between the agreement stage and the execution stage of a traditional Byzantine fault-tolerant system.

provided by the agreement stage as input, but do not make any assumptions on how the order has been established. As a result, ODRC is not restricted to a specific agreement protocol. Furthermore, with selectors emitting a sequence of client requests as output, the efforts of integrating execution replicas with ODRC are similar to the overhead in other Byzantine fault-tolerant systems.

Extended Cell Besides its regular configuration, ODRC offers the possibility to extend a cell by adding more execution replicas, each preferably running on its own physical server. Increasing the number of execution replicas (and consequently selectors) further extends the processing capacities of the system, allowing it to execute more requests at a time; we discuss such scalability issues in more detail in Section 6.6.2. Note that adding more agreement replicas to a cell could also lead to improvements for agreement stages that are able to take advantage of the additional replicas [89]; however, for most Byzantine fault-tolerant agreement protocols [34, 41, 96, 153, 154] such a measure would result in a performance decrease, due to an increase in communication overhead.

6.2.2 Service Application

In the following, we present assumptions about the service state as well as the functionality of an execution replica that allow a service application to be integrated with ODRC.

6.2.2.1 Service State

ODRC assumes that execution replicas implement a deterministic state machine (see Section 2.1.2.3) that consists of a set of variables X encoding its service state (see Section 2.1.1.5) and a set of operations working on them. The execution of an operation

in the course of processing a client request leads to an arbitrary number of state variables (i.e., none, one, more than one, or all) being read and/or modified and a reply being created. ODRC requires the state variables X of the service application to be grouped into *(state) objects* [35, 79, 97, 159]: Each state object has a unique identifier x and comprises a set of up to $|X|$ state variables (i.e., $0 < |x| \leq |X|$); the sizes of different objects may vary. Altogether, objects cover the entire state (i.e., $\bigcup x_i = X$). For simplicity and without loss of generality, in this chapter, we assume state objects to be disjoint (i.e., $x_i \cap x_j = \emptyset$, if $i \neq j$).
Note that ODRC only requires the existence of state objects but makes no assumptions on how they are assigned. As a result, different service applications can and should provide this abstraction in different ways. In general, there are two particular rules that should be followed: First, state variables that are often accessed by the same operation(s) should belong to the same state object. Second, state variables that are rarely accessed together should be assigned to different state objects.
In practice, the abstractions used by a service application are usually a good pointer for finding ODRC state objects: For file systems, for example, each file should be its own object (see Section 6.7) as different files are rarely, if ever, accessed by the same operation; the same applies to meta-data services of most distributed file systems [14, 79, 107]. For coordination services [22, 30, 84] (see Section 6.8) and key-value stores [55, 68], a state object may comprise a single data entry, whereas for more powerful databases it could make more sense to map an entire bucket of keys to an ODRC object.

6.2.2.2 Request Analysis

ODRC makes the decision whether or not to process a client request on a particular execution replica based on the state objects the request accesses. To retrieve this information, we assume that an application-specific *request analysis function* can be specified for every service integrated with ODRC:

```
SET<OBJECT> analyzeRequest(REQUEST request);
```

When `analyzeRequest` is invoked for a request, the function returns the maximum set of state objects that might be accessed during the processing of the request in the execution replica. Note that, in the context of ODRC, we state that a request *accesses a (state) object* when the object is included in the set of objects returned by the request analysis function, regardless of whether the request actually reads or modifies the object during execution. Due to focusing on stateful services, in the following, we assume each request to access at least one state object.
If a detailed analysis of a client request is not feasible, for example, due to being too expensive, the request analysis function can be implemented conservatively; that is, the set of state objects returned may contain more objects than the request is actually going to access. For some services, however, it might not be possible at all to specify a request analysis function as client requests do not carry enough information to determine the state objects they access. Nevertheless, many existing replicated Byzantine fault-tolerant systems use similar functions to efficiently implement state-machine replication: For ex-

```
1 /* Execution of service requests */
2 REPLY processRequest(REQUEST request);

4 /* Checkpointing of service-state objects */
5 OBJECTSTATE getObjectState(OBJECT object);
6 void setObjectState(OBJECT object, OBJECTSTATE state);
```

Figure 6.2: Overview of the functionality required from a service application in order to be integrated with ODRC (pseudocode): Besides comprising a function to process client requests, an application must provide means to retrieve and replace the contents of individual state objects.

ample, systems derived from BASE [35, 97, 161] utilize information about state access of requests to determine state changes; in addition, CBASE [97] executes client requests in parallel based, among other criteria, on the state objects they access in the course of being processed. In Sections 6.7 and 6.8 we discuss examples of request analysis functions for a distributed file system and a coordination service, respectively.

6.2.2.3 Execution-replica Interface

In addition to the properties discussed above, an execution replica of a service application at least needs to provide the methods included in the interface presented in Figure 6.2 in order to be integrated with ODRC: As in SPARE (see Section 4.2.2) and REBFT (see Section 5.2.2), an execution replica must provide a method to process client requests (L. 2). However, this method is only required to return a reply containing the result of the operation executed; in contrast to SPARE and REBFT, ODRC does not rely on state updates.
As most existing Byzantine fault-tolerant systems [34, 96, 97, 152, 154, 161], including SPARE and REBFT, ODRC makes use of checkpoints to retrieve and set the service state of execution replicas. However there are two main differences between ODRC and most other Byzantine fault-tolerant systems: First, as proposed by Yin et al. [161], checkpoints in ODRC are only used within the execution stage and have no effect on the agreement stage; treating the agreement stage as a black box (see Section 6.2.1), ODRC makes no assumption on how the agreement protocol performs garbage collection. Second, ODRC relies on *object checkpoints* only covering individual state objects, not full checkpoints comprising the contents of the entire service state.
Note that the two methods required to get and set the state of objects in ODRC (L. 5–6) may be implemented based on already existing mechanisms: For example, systems derived from BASE [35, 97, 161] implement a copy-on-write approach that only snapshots state objects modified since the last checkpoint; in addition, these systems also provide a method to selectively update state objects. In contrast, implementing such methods from scratch offers the potential to create object checkpoints more efficiently by taking advantage of a crucial difference between full checkpoints and object checkpoints: While the creation of a full checkpoint usually requires exclusive access to the entire service state in order to be consistent (“stop the world”), object checkpoints can be safely taken in parallel to the execution of requests accessing different state objects.

6.3 Selective Request Execution

In this section, we introduce the concept of *selective request execution* and describe how the selection stage of ODRC (see Section 6.2.1) applies it to increase performance by improving resource efficiency: Instead of processing all requests on all execution replicas available in the cell, as in traditional approaches [34, 41, 96, 97, 152, 153, 154, 161], a request in ODRC is executed on only a subset of execution replicas that is selected based on the state objects accessed by the request. As a result, selective request execution reduces the individual load on execution replicas, consequently freeing resources, which can then be used to process additional requests. Note that in this section, we assume the absence of faults; this assumption is dropped in Section 6.4 in which ODRC's fault-handling procedures are discussed in detail.

6.3.1 State Distribution

As in traditional Byzantine fault-tolerant systems [34, 41, 96, 97, 152, 153, 154, 161], all execution replicas in ODRC are required to host all objects belonging to the state of a service application. However, in contrast to existing approaches, each execution replica in ODRC is only responsible for keeping a particular subset of state objects up to date at all times. The assignment of state objects to execution replicas is performed statically satisfying one important requirement:

> *In order to ensure safety (see Section 2.1.1.3), state objects must be distributed across the cell in a way that each object is assigned to (at least) $f + 1$ execution replicas.*

Apart from this requirement, ODRC poses no restrictions on how the decision which state object to assign to which execution replica is reached. Throughout this chapter, we assume each state object to be assigned to exactly $f + 1$ execution replicas.
In the context of ODRC, we refer to a state object that is part of an execution replica's subset of objects as a *maintained (state) object* of the particular execution replica. As a result of the requirement above, each state object is maintained on $f + 1$ execution replicas and *unmaintained* on all others. We assume that each execution replica in the cell has access to the information on which subset of execution replicas a state object is maintained, for example, by means of a global assignment relation.

6.3.2 Selector

ODRC implements the concept of selective request execution by introducing a selection stage between the agreement stage and the execution stage of a Byzantine fault-tolerant system (see Section 6.2.1). The selection stage comprises a set of selector components, one for each execution replica in the cell; in the remainder of this chapter, we refer to an execution replica that corresponds to a particular selector as the selector's *local* execution replica. Note that a selector only performs selective request execution on behalf of its local execution replica. Selectors of different execution replicas do not interact with each

```
1 /* Methods for normal-case operation */
2 void insertRequest(REQUEST request);
3 REQUEST nextRequest();

5 /* Method for fault handling */
6 void forceRequest(REQUEST request);
```

Figure 6.3: Overview of the ODRC selector interface (pseudocode): During normal-case operation, an agreement replica and its corresponding execution replica use a selector as a producer/consumer queue calling `insertRequest` and `nextRequest`, respectively, in order to exchange client requests. In contrast, the `forceRequest` method is only called by the agreement replica in the course of fault-handling procedures, notifying the selector that result verification for a particular request has stalled and consequently additional replies are needed to make progress.

other in order to decide where to process a request, but rely on the same deterministic state machine (see Section 6.3.3) and operate on the same input: the totally-ordered sequence of client requests provided by the agreement stage (see Section 2.1.2.2). As a result, all non-faulty selectors of the same cell behave in a consistent manner.
Figure 6.3 presents the interface of a selector in ODRC: Relying on the two methods `insertRequest` (L. 2) and `nextRequest` (L. 3), an agreement replica and an execution replica are able to use a selector like a producer/consumer queue: When the next client request in line has become stable, the agreement replica submits the request to the selector by calling `insertRequest`. Independently, the execution replica fetches the next client request to be executed by calling `nextRequest`, which blocks while there are no requests ready for processing. For simplicity, we assume that an execution replica calls the `nextRequest` method of its local selector as soon as the invocation of `processRequest` (see Section 6.2.2.3) for the previous request has completed. In contrast to the other methods, `forceRequest` (L. 6) is only used in the course of fault handling: As discussed in detail in Section 6.4.3.1, invoking this method allows an agreement replica to force a selector to process a client request that so far has not been selected for execution on the local execution replica.

6.3.3 Basic Algorithm

A selector's main task is to determine whether or not to process a client request on the local execution replica. To make this decision, a selector executes a classification algorithm for each invocation of the `insertRequest` method by an agreement replica.

Overview The main goal of the selector algorithm is to divide client requests into two categories (see Figure 6.4): The first category includes requests that are selected for execution on the local execution replica due to accessing *at least one maintained state object*. Such requests are added to a dedicated queue, respecting the order defined by the agreement stage. When the local execution replica fetches the next request to process, the selector removes the first element from the queue of selected requests and returns

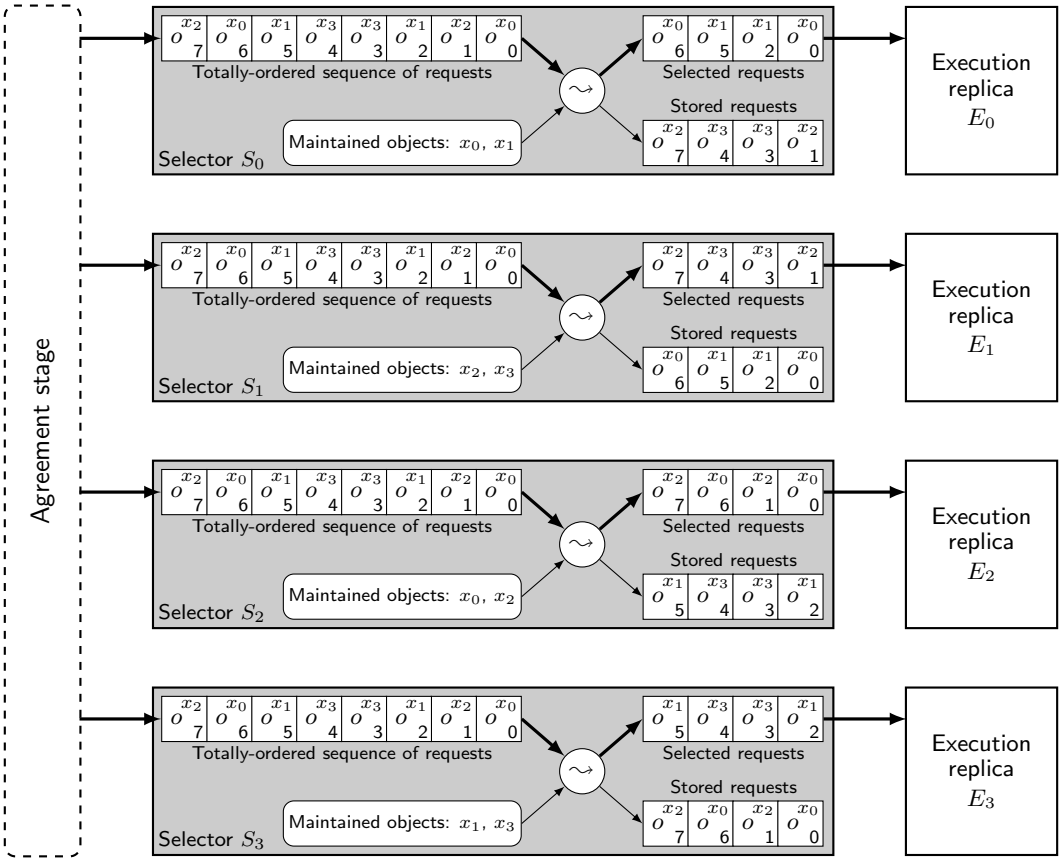


Figure 6.4: Example of ODRC selectors in action for a service state comprising four objects $x_0, ..., x_3$ and a subsequence of eight totally-ordered client requests $o^{\text{Object accessed}}_{\text{Sequence number}}$: All requests accessing at least one locally maintained object are selected for execution on the local execution replica assigned to a selector; requests that are not selected are buffered in a queue.

it to the replica. In contrast, the second category includes all remaining requests, which have not been selected for local execution. Such requests are inserted into a storage queue, thereby preserving their relative order to each other. Note that this approach allows the selection stage of ODRC to maintain the following invariant:

> *At all times, a selector is able to bring each state object on its local execution replica up to date, if both the selector and the execution replica are not faulty.*

With regard to maintained state objects, this is true because such objects are always up to date on the local execution replica due to the replica processing all client requests accessing at least one maintained object. In contrast, unmaintained state objects may become outdated. However, the selector, if necessary, is always able to update such objects by subsequently selecting all (state-modifying) client requests from the storage queue for execution that access the unmaintained objects in question.

Global data structures		
SET<OBJECT>	$X_{MAINTAINED}$	Static set of locally maintained state objects
QUEUE<REQUEST>	selected	Queue containing requests that have been selected for execution
QUEUE<REQUEST>	stored	Queue containing requests that have not been selected for execution

```
1  void insertRequest(REQUEST request) {
2    /* Analyze request in order to get the set of state objects accessed during execution. */
3    SET<OBJECT> X_request := analyzeRequest(request);
5    /* Log access to unmaintained state objects (see Figure 6.8). */
6    logAccess(request);
8    /* Check whether the request will access maintained state objects. */
9    if(X_request ∩ X_MAINTAINED == ∅) {
10     /* Do not select request for execution as it will only access unmaintained state objects. */
11     stored.enqueue(request);
12   } else {
13     /* Ensure that all unmaintained objects the request will access are up to date (see Figure 6.6). */
14     updateObjects(X_request \ X_MAINTAINED, request);
16     /* Select request for execution on the local execution replica. */
17     selected.enqueue(request);
18   }
19 }
```

```
20 REQUEST nextRequest() {
21   /* Return the next selected request to be executed. */
22   return selected.dequeue();
23 }
```

Figure 6.5: Basic ODRC mechanism for selecting requests to be executed (pseudocode): If a client request will access at least one maintained state object, the request is selected for execution on the execution replica assigned to the selector; otherwise, the request is stored in a local queue.

In combination with ODRC's state-distribution scheme, in the absence of faults, the selection algorithm guarantees that at least $f + 1$ execution replicas provide replies to each client request: As discussed in Section 6.3.1, each state object is maintained by at least $f + 1$ execution replicas. Therefore, assuming that every client request accesses one or more state objects (see Section 6.2.2.2), each request is selected for execution by at least $f + 1$ selectors due to accessing a state object that is maintained by the respective local execution replica. In consequence, clients are provided with enough replies to verify the correctness of a result. In the presence of faults, replies from additional execution replicas are required to ensure progress, as further discussed in Section 6.4.3.1.

Detailed Algorithm As shown in Figure 6.5, when an agreement replica invokes the `insertRequest` method of a selector, the selector first executes the application-specific request analysis function (see Section 6.2.2.2) to determine the set of state objects $X_{accessed}$

the client request will access during execution (L. 3). Next, the selector logs the accesses to all unmaintained objects included in $X_{accessed}$ (L. 6); this step will be further discussed in Section 6.4.3.2. If $X_{accessed}$ only consists of unmaintained state objects, the selector adds the request to the storage queue (L. 11); in this case, the request will not be processed on the local execution replica. However, if $X_{accessed}$ contains at least one maintained state object, the request is selected for local execution. In consequence, the selector inserts the request into the output queue of selected requests (L. 17) in order for the request to be fetched by the local execution replica (L. 22). Prior to that, the selector ensures that the states of all objects accessed by the request are up to date (L. 14); please refer to Section 6.4.2 for a detailed description of the steps required for this task.

6.3.4 Checkpoints and Garbage Collection

As discussed in Section 6.3.3, storing client requests that have not been selected for execution on the local replica allows a selector to update each local copy of an unmaintained state object at any point in time. In order to limit the size of its request store, a selector uses a garbage-collection mechanism similar to the one proposed by Yin et al. [161]. The mechanism relies on periodic checkpoints that become stable as soon as $f + 1$ matching certificates from different execution replicas are available. When all state modifications caused by a client request are part of stable checkpoints, it is safe for a selector to discard the request from its local request store. In contrast to Yin et al. [161], garbage collection in ODRC is based on object checkpoints covering only a single state object (see Section 6.2.2.3) instead of full checkpoints comprising the entire service state.

Checkpoint Creation As in REBFT (see Section 5.3.1.3) and other Byzantine fault-tolerant systems [34, 96, 97, 152, 154, 161], the creation of a checkpoint in ODRC is not dependent on physical time but on the number of requests executed: For every Kth execution of a request accessing a state object x, a selector id_S generates an object checkpoint for x by invoking the `getObjectState` method of the local execution replica (see Section 6.2.2.3); K is a system-wide constant (e.g., 100). Using every Kth access to an object to trigger the creation of an object checkpoint guarantees consistency: Due to the fact that all requests accessing a maintained object are selected for local execution (see Section 6.3.3), all selectors assigned to the same object x create and store a checkpoint for object x after their local execution replicas have processed the same request.
In a second step, the selector id_S computes a hash x_{hash} of the object checkpoint and multicasts a $\langle \text{CHECKPOINT}, id_S, s, x_{hash}, \mathcal{V}_{hashes} \rangle$ certificate to all selectors in the cell; s is the agreement sequence number of the client request that triggered the creation of the object checkpoint and $\mathcal{V}_{hashes}$ is a set of hashes of the replies to the K latest requests that have accessed object x. Please refer to Section 6.4.3.1 for a discussion of the rationale behind including $\mathcal{V}_{hashes}$ in the checkpoint certificate.

Checkpoint Verification Similar to checkpoints in the system proposed by Yin et al. [161], an object checkpoint in ODRC becomes stable as soon as a selector manages to obtain $f + 1$ matching checkpoint certificates from different selectors for the same sequence

number and state object. Checkpoint stability guarantees that, in case of being required to fetch the full object state from another selector (see Section 6.4.2), a selector is able to verify the state's correctness based on the hash included in the stable checkpoint.
When a checkpoint becomes stable, a selector can not only be sure of having learned the correct hash of the object state but also of having obtained a sufficient number of matching versions to successfully verify the reply hashes contained in $\mathcal{V}_{hashes}$. Consequently, it is safe for a selector to hand over the verified reply hashes to its local agreement replica in order for them to be added to the reply cache. This way, as further discussed in Section 6.4.3.1, ODRC ensures that a client is provided with enough replies (and reply hashes) to prove a result correct, even if up to f execution replicas that actually processed the corresponding request become faulty before their replies reach the client.
In the absence of faults, ODRC's state-distribution scheme presented in Section 6.3.1 ensures that at least $f + 1$ matching checkpoint certificates provided by different execution replicas become available, eventually leading to all selectors considering an object checkpoint to be stable. In the presence faults, checkpoint certificates from additional execution replicas are required to make progress, as discussed in Section 6.4.3.2.

Garbage Collection When an object checkpoint becomes stable, a selector checks whether it is safe to remove client requests from its local request store: A stored request with sequence number s may not be deleted before the selector has obtained stable checkpoint certificates indicating sequence numbers of s or higher for all state objects accessed by the request. In other words, a request can be garbage-collected when all its effects are reflected in stable object checkpoints. As long as this is not the case, a client request must remain in the local request store because it might be necessary to process the request in the course of fault handling (see Section 6.4.3.1). For the same reason, an old checkpoint must be kept until the request store contains no more requests depending on it.

6.4 On-demand Replica Consistency

In this section, we introduce the concept of on-demand replica consistency, which, besides giving the system its name, allows ODRC to efficiently handle client requests that not only access maintained but also unmaintained state objects; in the following, such requests are referred to as *cross-border requests*. Furthermore, we present ODRC's fault-handling mechanism, which relies on on-demand replica consistency to provide additional replies to client requests in the presence of faults.

6.4.1 Concept Overview

Using the algorithm presented in Section 6.3.3, a selector ensures that maintained state objects are always kept up to date on its local execution replica by processing all client requests accessing such objects. However, unmaintained state objects may become outdated, if requests modifying them are not selected for execution but instead added to the selector's local request store. For service applications in which client requests never "cross borders" and always access disjunct sets of state objects, for example, due to every

request reading and/or writing only a single object, no additional measures have to be taken in order to guarantee consistency. In all other cases, the basic selector algorithm alone is not sufficient as cross-border requests must not access outdated state objects.
One way to guarantee consistency of affected objects would be to perform a full state update, including the transfer of state contents from other execution replicas. However, besides being expensive in terms of communication and processing overhead, such an approach would also consume more resources than actually necessary due to also updating unmaintained state objects that will not be accessed on the local execution replica. Therefore, to minimize both performance and resource overhead for the handling of cross-border requests, ODRC relies on the concept of on-demand replica consistency. In particular "on demand" refers to two dimensions:

- **Time**: Consistency of the service state of an execution replica is only ensured when a client request to be executed actually demands it.
- **Space**: Consistency of the service state of an execution replica is only ensured for objects actually accessed by a client request to be executed.

As discussed in more detail in Section 6.4.2, a selector implements on-demand replica consistency by relying on a combination of object checkpoints and request execution: First, the selector applies the latest stable checkpoints for the affected unmaintained state objects to its local execution replica. Then, the selector instructs the execution replica to process all stored client requests accessing the particular state objects. Note that, besides being used in the context of cross-border requests, the concept of on-demand replica consistency also plays an important role in ODRC's fault handling (see Section 6.4.3).

6.4.2 Handling Cross-border Requests

Prior to selecting a cross-border request for local execution, a selector is required to pay special attention to execution-replica consistency: With a cross-border request not only accessing maintained but also unmaintained state objects, a selector must ensure that all objects read and/or modified by such a request are up to date on the selector's local execution replica. To perform this task, a selector in ODRC invokes the `updateObjects` method before forwarding the request (see Figure 6.5, L. 14). The method executes the two-step algorithm presented in Figure 6.6: In the first step, the algorithm determines which of the requests contained in the local request store contributes to bringing the unmaintained state objects in question up to speed. In the second step, the affected objects are actually updated by processing the requests selected in the first step on the local execution replica.

Selection of Requests Instead of processing all client requests contained in the local request store, which would result in a significant overhead, a selector aims at identifying the minimum set of requests that have dependencies to a cross-border request and therefore have to be executed in order to prepare the service state of the local execution

Global data structures		
QUEUE<REQUEST>	`selected`	Queue containing requests that have been selected for execution
QUEUE<REQUEST>	`stored`	Queue containing requests that have not been selected for execution
REPLICA	`replica`	Local execution replica

```
1  void updateObjects(SET<OBJECT> X_update, REQUEST request) {
2    /* Get sequence number of the latest stored request prior to request. */
3    int seqNr := highest seq. nr. of a request in stored that is smaller than the seq. nr. of request;
5    /* Starting with the latest, determine stored requests that have dependencies to request. */
6    QUEUE<REQUEST> dependencies := ∅;
7    for(int i := seqNr; i ≥ 0; --i) {
8      REQUEST req := stored.get(i);
9      if(req == null) continue;
11     /* Check for dependencies between request r and the requests in dependencies. */
12     SET<OBJECT> X_req := analyzeRequest(req);
13     if(X_req ∩ X_update ≠ ∅) {
14       /* Request req has a direct or indirect dependency to request. */
15       dependencies.enqueue(r);
16       X_update := X_update ∪ X_req;
17     }
18   }
20   /* Update objects accessed by requests in dependencies using the respective object checkpoints. */
21   for each OBJECT object in X_update {
22     OBJECTSTATE state := fetch and verify the state of object based on stable checkpoint;
23     if(state has not already been applied) {
24       /* Update object on the local execution replica using state. */
25       replica.setObjectState(object, state);
26     }
27   }
29   /* Select requests in dependencies for execution on the local execution replica. */
30   for(int i := (dependencies.size() − 1); i ≥ 0; −−i) {
31     Request req := dependencies.get(i);
32     stored.delete(req);
33     selected.enqueue(req);
34   }
35 }
```

Figure 6.6: Algorithm of an ODRC selector for updating unmaintained state objects (pseudocode): Starting with the latest stored request that has not been selected for execution, the algorithm goes back in time in order to identify and subsequently select requests that contribute to bringing the unmaintained state objects in question up to date. If these requests require access to additional state objects, the selector takes care of updating them as well.

replica. Starting with the latest stored client request whose sequence number is smaller than the sequence number of the cross-border request (L. 3 in Figure 6.6), the following operations are repeated by the selector for each stored request `req` (L. 7–8).
First, the set of state objects X_{req} accessed by request `req` is composed using the request analysis function (L. 12, see Section 6.2.2.2). Second, if any object in X_{req} is a member in the set of state objects to update X_{update}, request `req` contributes to bringing them up to date and is therefore selected for execution (L. 13–15); furthermore, X_{update} is updated by adding all objects contained in X_{req} (L. 16) as these objects also have to be consistent when request `req` will be processed on the local execution replica. Note that additional objects in X_{req} only have to be updated to the extent required by request `req`; they do not have to reflect the current state of the object.
In summary, this algorithm step goes back in time selecting all stored client requests accessing objects to update. As these requests may require additional objects to be consistent, those objects are also updated to resemble their state at this point in time. In the worst case, for each object affected, this procedure involves processing all corresponding client requests since the latest stable object checkpoint.

Update of State Objects Having selected the client requests to execute, a selector is able to perform the actual update procedures for the unmaintained state objects accessed by the cross-border request: First, for each object in X_{update}, the selector applies the latest stable object checkpoint (L. 21–25); with checkpoint certificates in ODRC only comprising state-object hashes (see Section 6.3.4), this step requires fetching the corresponding object state from another selector in the cell (L. 22). Next, the selector forwards all requests selected in the first algorithm step to its local execution replica, thereby preserving their relative order as defined by the agreement stage (L. 30–33). As a result, at the time the local execution replica will process the cross-border request, all maintained and unmaintained state objects accessed by the request will be up to date, allowing the replica to provide a consistent reply without needing to update its entire service state.

6.4.3 Fault Handling

As discussed in Section 6.3.3, the key idea behind the concept of selective request execution is to process each client request on the minimum number of execution replicas necessary to make progress. In consequence, non-cross-border requests are executed on $f + 1$ different replicas during normal-case operation as in such case $f + 1$ replies are sufficient for a client to prove a result correct; in contrast, depending on their state-access characteristics, cross-border requests are processed on additional execution replicas (see Section 6.4.2). While this approach is able to ensure liveness in the absence of faults, a client might not be provided with enough matching replies to successfully complete result verification in the presence of faults; a related problem arises in the context of stalled object-checkpoint verifications. As described in the following, to address these issues, ODRC's fault-handling procedures lead to additional execution replicas processing the requests affected, eventually enabling clients to make progress.

6.4.3.1 Stalled Result Verification

Following the standard procedure in Byzantine fault-tolerant systems described in Section 2.1.2.1, a client in ODRC reacts to a stalled result verification by sending a notification to all agreement replicas in the cell. The subsequent fault handling is performed by both the agreement stage and the selection stage.

Role of the Agreement Stage Upon receiving a stalled-verification notification from a client, an agreement replica in ODRC performs the standard fault-handling procedures. Amongst other things, this includes the retransmission of the corresponding reply in cases where the particular request has already been processed by the local execution replica.
In contrast to other approaches, the reply cache of an agreement replica in ODRC may also contain replies to requests that have not been executed locally: As discussed in Section 6.3.4, through stable object-checkpoint certificates, an agreement replica in ODRC obtains verified hashes of replies originating from non-local execution replicas. This way, an agreement replica is even able to assist in fault handling if the service state of its local execution replica has advanced to a point where the execution replica is no longer able to process the request; such a situation occurs when an execution replica applies a stable object checkpoint in which the effects of the request in question are already reflected (i.e., an object checkpoint whose sequence number is higher than the sequence number of the request). Note that in such cases it is sufficient for the affected agreement replicas to only provide a reply hash to the client: As the object-checkpoint certificate, and therefore the reply hash, has become stable, there must be at least one non-faulty agreement replica in the cell whose local execution replica has actually processed the request, enabling this agreement replica to retransmit the full reply on the reception of a stalled-verification notification.
When fault handling for a request is triggered it is not guaranteed that the local reply cache of an agreement replica in ODRC contains the corresponding reply, for example, due to the execution of the request still being in progress. Also, the local selector might not have selected the request for execution in the first place. With an ODRC agreement replica not being able to distinguish between both cases, it cannot decide whether a local reply will eventually arrive or not, and therefore has to rely on the selector to participate in fault handling: For this purpose, each time an agreement replica receives a stalled-verification notification for a request whose reply is not available in the cache, the agreement replica invokes the selector's `forceRequest` method (see Section 6.3.2) for the request in question.

Role of the Selection Stage An invocation of `forceRequest` can be seen as an order of the agreement replica to the selector to ensure that the local execution replica, provided that it is non-faulty, processes a request and generates a corresponding reply. A non-faulty selector complies with this order by performing the steps shown in Figure 6.7: If the request has already been selected for local execution, there are no additional actions to be done (L. 3 and 8); as soon as processing is complete, the reply will become avail-

Global data structures		
QUEUE<REQUEST>	selected	Queue containing requests that have been selected for execution
QUEUE<REQUEST>	stored	Queue containing requests that have not been selected for execution

```
 1 void forceRequest(REQUEST request) {
 2   /* There is nothing to be done if the request has already been selected for execution. */
 3   if(request ∉ stored) return;

 5   /* Select request for execution on the local execution replica. */
 6   SET<OBJECT> X_request := analyzeRequest(request);
 7   updateObjects(X_request \ X_MAINTAINED, request);
 8   stored.delete(request);
 9   selected.enqueue(request);
10 }
```

Figure 6.7: Basic fault-handling mechanism of an ODRC selector (pseudocode): On the reception of a stalled-verification notification for a request whose reply is not yet available, an agreement replica calls the selector's `forceRequest` method indicating the request in question. If the request so far has not been selected for execution, it will now be processed on the local execution replica. With all non-faulty selectors in the cell behaving accordingly, a sufficient number of additional replies are provided to allow the client to successfully complete result verification.

able. Otherwise, the selector retroactively selects the request for execution on the local execution replica, using similar steps as in the regular procedure (see Figure 6.5): First, the selector updates the state objects accessed by the request (L. 6–7), which are all unmaintained. Then, the selector forwards the request to its local execution replica (L. 9), leading to an additional reply being provided to the client handling the stalled result verification. With all selectors that originally have not selected the request for execution learning about the stalled-verification notification, the client will eventually receive enough matching replies to prove the result correct.

6.4.3.2 Stalled Object-checkpoint Verification

With ODRC assigning the responsibility for keeping a state object up to date to only a minimum number of selectors, faulty or slow replicas may temporarily prevent object checkpoints from becoming stable, for example, by providing a faulty object-checkpoint certificate. In order to ensure progress in such case, ODRC selectors rely on the mechanism sketched in Figure 6.8 which leads to additional certificates being created if an object checkpoint cannot be successfully verified within a certain period of time after its creation: By recording the access to unmaintained state objects, a selector is able to determine when a checkpoint for such an object is due. If one or more selectors maintaining a state object at this point fail to provide a correct checkpoint certificate, a selector instructs its local execution replica to step in.

Global data structures		
SET<OBJECT>	$X_{MAINTAINED}$	Static set of locally maintained state objects
TABLE<OBJECT, INTEGER>	`accessed`	Table containing access counters for unmaintained state objects

```
 1 void logAccess(REQUEST request) {
 2   /* Determine unmaintained state objects accessed by the request. */
 3   SET<OBJECT> X_request := analyzeRequest(request);
 4   SET<OBJECT> X_request_unmaintained := X_request \ X_MAINTAINED;

 6   /* Start timers if object checkpoints are due. */
 7   for each OBJECT object in X_request_unmaintained {
 8     if((++accessed[object] mod K) == 0) {
 9       Attach request to a timer timer for object;
10       Start timer: On expiration invoke alarm;
11     }
12   }
13 }
```

```
14 void alarm(REQUEST request) {
15   /* Force object-checkpoint creation by processing the request on the local execution replica. */
16   forceRequest(request);
17 }
```

Figure 6.8: Monitoring mechanism for object checkpoints in ODRC (pseudocode): By recording the access to unmaintained state objects, a selector knows when checkpoints for these objects are due. In case such an object checkpoint does not become stable within a certain period of time, a selector forces the request that triggers the checkpoint in question to be processed on the local execution replica; this eventually leads to an additional checkpoint certificate being created.

Monitoring Access to Unmaintained State Objects As discussed in Section 6.3.2, for every request forwarded by the agreement stage, a selector invokes a method called `logAccess` (see Figure 6.5, L. 6). As shown in Figure 6.8, this method increments a counter for each unmaintained state object accessed by the request (L. 7–8). This way, a selector is able to determine the point in time at which the next checkpoint for an unmaintained state object is scheduled: A resulting access-counter value divisible by K (i.e., the checkpoint interval, see Section 6.3.4) indicates that all non-faulty execution replicas maintaining the particular state object will checkpoint this object after having executed the request. In consequence, the corresponding object checkpoint should become stable within a certain (application-dependent) period of time. To monitor this, a selector attaches the request to an object-specific timer (L. 9) and starts the timer (L. 10). The timer is stopped when the selector is able to obtain a stable checkpoint for the object (omitted in Figure 6.8), which is the regular scenario in the absence of faults and network delays.

Forcing Object-checkpoint Creation In case a selector is not able to successfully verify an object checkpoint in time, the object-specific timer expires and invokes its handler method `alarm` (L. 10, 14). At this point, a selector triggers the local fault handling for

the request attached to the timer (L. 16): As discussed in Section 6.4.3.1, this leads to all unmaintained state objects accessed by the request being updated and the request being processed on the local execution replica. Furthermore, with the request performing the Kth access to the object in question since the latest checkpoint, its execution triggers the creation and distribution of the next checkpoint. With all non-faulty selectors that do not maintain the object behaving in the same manner, all selectors in the cell are eventually provided with a sufficient number of correct certificates to be able to successfully verify the object checkpoint.

6.5 Safety and Liveness

Introducing a set of selector components between agreement stage and execution stage creates additional potential points of failure. At the same time, with ODRC treating the agreement stage as a black box (see Section 6.2.1), most mechanisms of the surrounding architecture ensuring safety (e.g., committing requests, see Section 2.1.2.2) and liveness (e.g., replacing a faulty leader, see Section 3.1.1) remain unaffected. In the following, we therefore limit our discussion of safety and liveness in ODRC to aspects directly related to the selection stage.

6.5.1 Containment of Faults

Being located between two stages, most of the interaction of a selector in ODRC is with its local agreement and execution replicas. If a fault occurs in one of these components, it may propagate to the selector, and vice versa, especially in cases where they are all are running on the same physical machine. Similar to the approach taken in traditional Byzantine fault-tolerant systems [34, 41, 49, 96, 97, 152, 153, 154], when considering an ODRC component to be faulty, one must therefore assume the entire replica (including agreement replica, selector, and execution replica) to be faulty. With a regular ODRC cell comprising a total of $3f + 1$ replicas (see Section 6.2.1), the system is able to tolerate up to f faulty replicas.

To ensure safety, selectors in ODRC have to be fault independent; that is, a failure of one selector must not result in the failure of another selector in the same cell. An important step towards achieving this property in practice is ODRC's design decision to greatly restrict the communication of a selector with external components not belonging to the selector's local replica. This characteristic not just results in faults being contained but also protects selectors from becoming compromised by other replicas. Note that there is only one occasion at which a selector interacts with other replicas: the exchange of object checkpoints between selectors (see Section 6.3.4). However, due to the fact that selectors only apply a checkpoint obtained from another selector after having successfully verified the checkpoint's correctness, this mechanism is not at risk of propagating faults and/or being exploited to compromise a selector.

6.5.2 Protection Against Malicious Clients

As discussed in Section 6.2.2.2, in order to apply selective request execution, selectors must analyze incoming client requests with regard to the state objects they access. With this step requiring selectors to interpret the message, a practical ODRC implementation must prevent malicious clients from being able to this way trigger correlated faults in different selectors. One approach to address this problem is the introduction of heterogeneous selector implementations (see Section 2.1.1.2).
Exploiting the fault-handling mechanism presented in Section 6.4.3.1, a malicious client in ODRC might try to trick a selector into processing arbitrary operations by sending a stalled-verification notification for a request that has not been committed in the agreement stage. Upon receiving such a notification, an agreement replica would invoke the `forceRequest` method of its local selector due to not having cached a corresponding reply for the request. However, as a selector in ODRC only considers selecting requests for execution that are included in the totally-ordered sequence provided by the agreement stage, the selector would ignore the request at this point. Furthermore, the selection algorithm used in ODRC prevents a non-faulty selector from processing the same client request more than once on its local execution replica.

6.5.3 Consistency of Execution-replica States

The safety properties of ODRC are primarily based on the safety properties of the protocol executed in the agreement stage. As ODRC does not modify the agreement protocol its correctness is preserved. In particular, it is guaranteed that in the presence of at most f faults all non-faulty agreement replicas provide their respective local selectors with the identical sequence of totally-ordered client requests (see Section 2.1.2.2). In traditional Byzantine fault-tolerant systems [34, 41, 49, 152, 154, 161], this agreement-stage output sequence $\mathcal{O}_A$ dictates the order in which all non-faulty execution replicas process requests. In contrast, in ODRC, a non-faulty execution replica processes requests based on a sequence $\mathcal{O}_S$ that is provided by its local selector; as a result, the sequence $\mathcal{O}_S$ differs across replicas. A correct selector transforms $\mathcal{O}_A$ into $\mathcal{O}_S$ ensuring the following two properties of the selector output sequence:

- **Dependent Requests**: Client requests whose sets of accessed state objects overlap in at least one object appear in $\mathcal{O}_S$ in the same relative order as in $\mathcal{O}_A$.
- **Independent Requests**: Client requests whose sets of accessed state objects are disjoint may appear in $\mathcal{O}_S$ in a different relative order as in $\mathcal{O}_A$.

Note that in this context ODRC relies on a similar notion of request dependency as CBASE [97], using state access as the main criteria. However, in contrast to CBASE, ODRC does not necessarily require knowledge about whether a request reads or modifies the state. As a result, read-only requests are treated as dependent in ODRC if their sets of accessed state objects overlap; in CBASE, such requests are not considered to be dependent. Nevertheless, as discussed in Section 6.6.1.2, providing a selector with knowledge about read-only access of requests is also beneficial in ODRC.

Reordering independent client requests in an ODRC selector output sequence is safe as, for a given initial execution-replica state, the result of processing those requests in any order leaves deterministic execution replicas in the same final state. A selector may therefore delay the execution of a request exclusively accessing unmaintained state objects until it is required in the course of handling cross-border requests (see Section 6.4.2) or faults (see Section 6.4.3). Furthermore, it is safe for a selector to not include such a request in $\mathcal{O}_S$ at all if, at some point, the selector manages to obtain a set of stable object checkpoints reflecting all effects of the request on the service state: For a successful verification of an object checkpoint, at least $f + 1$ selectors in the cell, and therefore at least one non-faulty selector, must have provided correct information about both the state modifications performed by the request as well as about the request's reply (see Section 6.3.4). In consequence, a selector in such case is able to safely update the state of its local execution replica while not losing the ability to assist in fault handling for the request (see Section 6.4.3.1), all without never having to process the request locally.

6.5.4 Ensuring System Progress

During normal-case operation, applying the concept of selective request execution, selectors in ODRC ensure that each request is processed on at least $f + 1$ execution replicas (see Section 6.3.3), enabling the corresponding client to obtain enough matching replies to successfully verify the result (see Section 2.1.1.4). In contrast, in the presence of network problems and/or faulty or slow system components, a client at first might not be able to complete result verification, for example, due to a malicious selector deliberately refraining from selecting a client request for execution. A client reacts to such a situation by sending a stalled-verification notification for the request in question to the agreement stage (see Section 6.4.3.1). With this notification eventually arriving at its designated destinations, possibly requiring multiple retransmissions, at some point in time, all non-faulty agreement replicas in the cell will be aware of the stalled result verification. As a consequence, non-faulty agreement replicas whose local execution replicas have not yet processed the affected client request will force their selectors to do so, resulting in a sufficient number of additional replies being provided to the client in order to successfully complete the stalled result verification.

6.6 Optimizations, Extensions, and Variants

Having presented the general ODRC mechanisms in previous sections, in the following, we discuss a number of implementation details and optimizations that are of significance for the system in practice. Furthermore, due to ODRC's fundamental concepts of selective request execution and on-demand replica consistency not being limited for use in a particular system configuration, we discuss extensions and variants of ODRC that illustrate the applicability of our approach to a wide spectrum of fault-tolerant systems.

6.6.1 Optimizations

Below, we present optimizations that either directly increase performance of the ODRC mechanisms presented in Sections 6.3 and 6.4 or contribute to the efficiency of the overall system. Amongst other things, we discuss how knowledge about the particular application use case can be utilized to improve efficiency and/or to reduce replication overhead.

6.6.1.1 Dynamic Adaptation of State-object Distribution

As discussed in Section 6.3.1, each state object of a service running in an ODRC cell must be assigned to at least $f + 1$ execution replicas in order to ensure safety. With execution replicas processing all client requests accessing maintained state objects (see Section 6.3.3), an optimal scheme distributes state objects in a way that load is equally balanced across all execution replicas in the cell. To find such an optimal scheme, application-specific knowledge about state-access patterns of operations as well as information on client workloads is key.

Adaptation to Changing Workloads Note that the requirement above does not necessarily demand the distribution of state objects to be static. For some services, a static distribution might be sufficient to achieve good load balancing over the lifetime of a service. However, this might not apply to all use-case scenarios as for other services state-access patterns may change over time. In a file system, for example, different subsets of files could become "popular" at different points in time. As a result, such workload variation could create a load imbalance when using a static distribution scheme. To address this problem, ODRC allows the distribution of state objects to be dynamically adapted, enabling the system to restore a load balance across execution replicas in the face of changing client workloads.

Implementation Hints One way to implement a dynamic state-object distribution in ODRC is to allow an administrator to manually alter the assignment of state objects to execution replicas. In such case, the corresponding command would be treated as a regular client request that is totally-ordered by the agreement stage and then handed over to the selection stage. With all selectors on non-faulty replicas processing the command at the same logical point in time, the transition between old and new object-to-execution-replica mapping is performed consistently across different execution replicas in the cell. However, adaptation to changing workloads does not necessarily require human intervention in ODRC: Due to selectors recording access to unmaintained state objects (see Section 6.4.3.2), their knowledge about the workload is not limited to the client requests they select for execution. In combination with information on which state object is assigned to which execution replica, each selector is therefore able to locally estimate the current load distribution in the cell. Based on such a mechanism, a selector whose local execution replica experiences high load may delegate the responsibility for maintaining a state object to another selector, consequently reducing the number of client requests its

local execution replica must actively process. In case the other selector fails to comply, for example, due to being malicious and not maintaining the state object from then on, fault-handling procedures ensure that eventually other selectors (including the selector the state object has initially been assigned to) will step in to tolerate the fault.

6.6.1.2 Application-specific Optimizations

As a minimum requirement, a selector in ODRC must have knowledge about which state objects a client request possibly accesses during execution (see Section 6.2.2.2). If, in addition, a selector has application-specific information to distinguish requests that modify the service state from requests only performing read operations [34, 35, 96, 97], the following optimizations are possible: First, in order to bring an unmaintained state object up to speed, a selector may only include requests actually modifying the object, ignoring all read-only requests that have no effect on the object's state. Second, instead of creating a checkpoint at every Kth *access* to a state object (see Section 6.3.4), checkpoint creation may only be triggered at every Kth *write access*. This way, for services comprising a high read-to-write ratio (see Section 4.12.3), the overhead for generating and verifying object checkpoints can be significantly reduced. Third, if a selector knows that a client request entirely replacing the content of an unmaintained state object will not read the object before modifying it, the selector does not have to update the unmaintained state object prior to forwarding the request to its local execution replica. Instead, the client request in such case can be directly selected for execution, resulting in a lower response time. In the context of coordination services [22, 30, 84] and key-value stores [55, 68], overwriting the value assigned to an existing key is an example for an operation where such an optimization would be possible.

6.6.1.3 Optimized Updating of Unmaintained State Objects

As discussed in Section 6.4.2, a standard ODRC selector updates the unmaintained state objects accessed by a cross-border client request when the agreement replica hands the affected request over to the selector; that is, after the request has been committed by the agreement stage. Taking into account that when a client request is introduced into the agreement stage it is very likely that the request will eventually be committed [96], additional information provided by the agreement replica may allow a selector to optimistically trigger the updating process for an unmaintained state object in advance: Relying on the PBFT protocol (see Section 3.1.1), for example, the reception of a PREPREPARE message after the first protocol phase is already a good indication that the corresponding request will soon be committed. If an agreement replica at this point forwards the PREPREPARE message (which contains the client request) to its local selector, the selector could already start to update the affected state objects. As a result, less updating has to be done when the request is actually committed and selected for execution. Note that performing such an optimistic update is safe even if agreement replicas will not agree on the request afterwards, for example, due to a malicious leader having proposed a faulty request; in the worst case, the selector has unnecessarily updated unmaintained state objects that will not be accessed.

Apart from initiating the update process based on an indication that certain state objects will be accessed in the near future, an ODRC selector may also be configured to proactively bring unmaintained state objects up to speed, for example, during periods of reduced workload. This way, an execution replica has to reproduce fewer state modifications in cases where requests actually demand consistency of unmaintained objects, leading to a minimized fault-handling latency, for example. In general, in order to limit the number of update procedures of unmaintained state objects, one might also define a maximum number of outstanding state-modifying client requests, forcing a selector to trigger proactive updates when a certain threshold is reached.

6.6.1.4 Efficient Fault Handling

In order to keep the number of executions to a minimum, an ODRC selector usually only processes client requests exclusively accessing unmaintained state objects when its agreement replica instructs the selector to do so by invoking the `forceRequest` method (see Section 6.4.3.1). Retaining this strategy in the presence of replica crashes or other fault scenarios whose effects are not limited to single requests, however, could result in undesirable system behavior: If one or more execution replicas continuously fail to process requests accessing a certain state object, verification processes for the corresponding results will stall, requiring clients in each case to send stalled-verification notifications before being able to successfully prove a result correct. To speed up fault-handling latencies in such scenarios, an optimized selector may collect statistics on `forceRequest` invocations: On an accumulation of stalled-verification notifications for client requests accessing state objects maintained by the same execution replica, such a selector suspects the replica of being faulty and temporarily adds the affected objects to its own set of maintained state objects. In consequence, clients are provided with additional replies without having to explicitly demand them first and are therefore able to much faster complete result verification. Note that such an optimization does not require any interaction between selectors as each selector is always free to select more requests for execution than it is actually required by the state-object distribution scheme.

6.6.2 Execution-stage Extensions

With all execution replicas processing all client requests, traditional agreement-based Byzantine fault-tolerant systems [34, 41, 49, 152, 154, 161] fail to provide throughput scalability; that is, increasing the total number of execution replicas, while keeping the maximum number of faults to tolerate constant, does not result in an increase in system throughput. To improve throughput scalability, systems have been proposed where only a quorum of execution replicas processes a request [1, 51]. However, in order to be safe, any two quorums are required to overlap in at least one non-faulty replica. Therefore, quorum size in such systems increases with the total number of execution replicas.
In contrast, the number of execution replicas a (non-cross-border) request is processed on in ODRC is only dependent on the maximum number of faults to tolerate f: Under benign conditions this number is $f + 1$; otherwise, at most $2f + 1$ execution replicas

have to execute a request [42, 159]. As a consequence, extending the execution stage with additional replicas (see Section 6.2.1) allows the system to achieve better throughput performance, as shown by the evaluation results from our case studies presented in Sections 6.7 and 6.8.
Note that, due to accessing state objects that are maintained on more than $f+1$ execution replicas, cross-border requests may become a limiting factor for the scalability of a service in ODRC. In the worst case, a cross-border request is dependent on the entire service state and therefore has to be processed on all execution replicas. However, we have no indication that such requests are more than an academic problem: In our case studies, a distributed file system and a coordination service, we found the maximum number of state objects read and/or modified by a single request to be four, with the great majority of operations accessing only a single state object. For comparison: Entire service states in such usage scenarios may comprise hundreds or even several thousands of state objects.

6.6.3 Variants

A regular ODRC cell comprises $3f + 1$ agreement replicas, $3f + 1$ selectors, and $3f + 1$ execution replicas and is able to tolerate f Byzantine faults (see Section 6.2.1). However, due to the concepts of selective request execution and on-demand replica consistency not being dependent on this particular system configuration, a variety of other use cases are possible. Below, we discuss three examples of alternative system configurations.

- **Minimal ODRC:** While a Byzantine fault-tolerant agreement stage that does not rely on trusted components requires at least $3f + 1$ replicas, the execution stage can be reduced to $2f + 1$ replicas (see Section 2.2.1). In consequence, a minimal ODRC system configuration comprises $3f + 1$ agreement replicas, $2f + 1$ selectors, and $2f + 1$ execution replicas.
- **ODRC with Trusted Components:** Selectors in ODRC treat the agreement stage as a black box, as discussed in Section 6.2.1. Therefore, in order to minimize a cell's resource footprint, the selection stage and the execution stage of the minimal ODRC system configuration presented above could be combined with an agreement stage relying on trusted components [41, 130, 152, 154], thereby reducing the number of agreement replicas to $2f + 1$.
- **Crash-tolerant ODRC:** Applying selective request execution in combination with on-demand replica consistency is not limited to Byzantine fault-tolerant systems. Instead, our approach can also be utilized to increase performance in crash-tolerant systems based on active state-machine replication: In such a scenario, state objects are distributed across the cell in a way that each object is maintained by exactly one execution replica. Furthermore, results and object checkpoints do not have to be verified by voting. Apart from that, mechanisms are similar to regular ODRC.

Note that minimizing the number of execution replicas in a cell is only reasonable if resource efficiency is the main concern: As such a step reduces the amount of resources available for request processing, it increases the individual load on replicas, and consequently limits the extent to which the selection stage is able to improve performance.

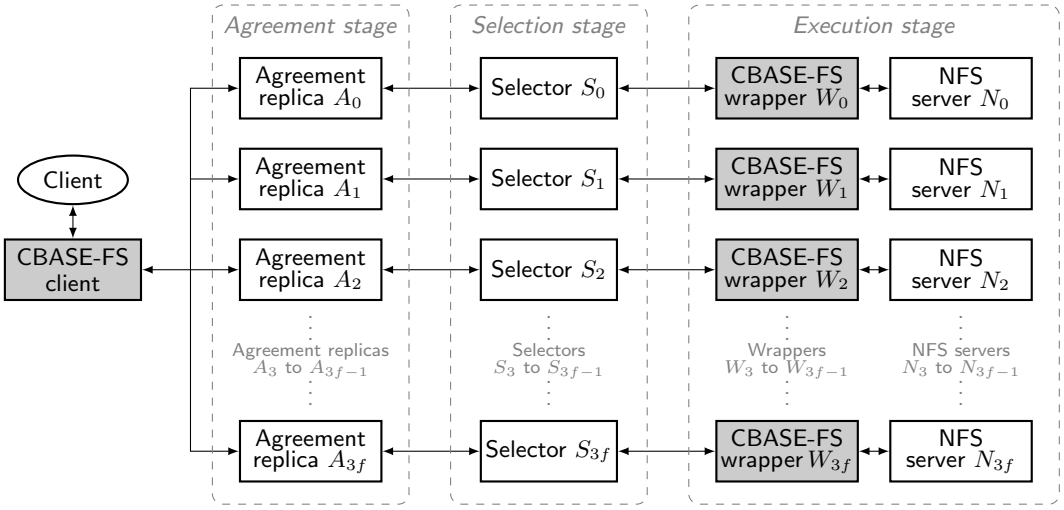


Figure 6.9: Architecture overview of the ODRC$_{NFS}$ prototype comprising components of CBASE-FS [97] (▭): In order to use the service, a client mounts ODRC$_{NFS}$ as a network file system. At the execution stage, a set of wrappers is responsible for ensuring deterministic multi-threaded execution of requests on an off-the-shelf NFS server.

6.7 ODRC$_{NFS}$: A Byzantine Fault-tolerant Network File System

In this section, we investigate the impact of selective request execution and on-demand replica consistency on non-trivial services. As a use-case example we have implemented ODRC$_{NFS}$, a Byzantine fault-tolerant network file system that is integrated with ODRC.

6.7.1 Service Integration

Instead of building ODRC$_{NFS}$ from scratch, we drew on the results of other researchers: Due to ODRC$_{NFS}$ not being the first Byzantine fault-tolerant network file systems based on state-machine replication [34, 35, 97, 161], in the following, we focus on ODRC-related aspects and omit a detailed discussion of solutions to general replication-related problems (e.g., non-determinism).

Prototype For the ODRC$_{NFS}$ prototype, we reused parts of CBASE-FS [97], which itself is an extension of a Byzantine fault-tolerant network file system written for BASE [35]. In contrast to the BASE implementation, CBASE-FS supports concurrent execution of independent client requests (see Section 6.5.3) and consequently provides better performance. However, by processing each client request on each execution replica, CBASE-FS has a high resource usage during normal-case operation. To address this issue, ODRC$_{NFS}$ extends CBASE-FS in order to combine the benefits of concurrent execution with selective request execution and on-demand replica consistency. Like BASE and CBASE-FS, ODRC$_{NFS}$ provides NFSv2 [145].

As shown in Figure 6.9, an application client uses $ODRC_{NFS}$ via a CBASE-FS client component that directly communicates with the ODRC cell. More precise, the main tasks of a CBASE-FS client is to issue file-system requests, to collect the corresponding replies provided by different execution replicas, and to verify the result by voting before returning it to the application client. At the server side, each $ODRC_{NFS}$ execution replica consists of two components: a CBASE-FS wrapper, which ensures deterministic concurrent execution of requests, and an off-the-shelf NFS server.

Selector Implementation As discussed in Section 6.2.2.1, ODRC requires the state of a service application to be divisible into objects in order to be able to perform selective request execution. NFS fulfills this requirement without the need for any modifications by already managing files and directories on the basis of *NFS objects*; consequently, $ODRC_{NFS}$ defines an ODRC state object to be a single NFS object.

Apart from assuming certain structural properties of the service state, an ODRC selector must also be able to retrieve information about the state objects accessed by a client request, as discussed in Section 6.2.2.2. Internally, NFS uses a *file handle* (i.e., a byte sequence of constant length) to uniquely identify each NFS object; there is only one type of file handle for both files and directories. In general, an NFS request carries the file handle(s) of the NFS object(s) the corresponding operation will read or modify, making it straight-forward to implement a request analysis function for ODRC. However, there is a subset of operations (including `CREATE`, `REMOVE`, and `RENAME`) that identify some of the objects accessed using the *name* of the NFS object (i.e., a string of variable length).

To deal with this non-uniform referencing scheme, an $ODRC_{NFS}$ selector maintains a name-to-file-handle mapping for each state object, allowing the selector to recognize the access to an object independent of whether it is performed via the object's file handle or its name. Note that the need for such a mapping could be eliminated by refactoring the NFS protocol to either use only file handles or only names to identify files and directories.

State-object Distribution Schemes Although ODRC requires each state object to be assigned to a minimum number of execution replicas in order to be safe (see Section 6.3.1), the problem of which state object to map to which execution replica is not relevant for correctness. With regard to performance, however, the state-object distribution scheme in use is of concern: By assigning state objects that are often accessed together to the same subset of execution replicas, the number of cross-border requests (see Section 6.4.2), and consequently the overhead for updating unmaintained objects, can be minimized.

Focusing on NFS, most operations read or modify only a single object (e.g., `SETATTR`, `GETATTR`, `READ`, and `WRITE`) and are therefore not capable of leading to cross-border requests in ODRC. However, some operations access two (e.g., `CREATE`, `REMOVE`, and `LOOKUP`) or even four (`RENAME`) NFS objects. As a consequence, such operations result in cross-border requests in case the state objects accessed are maintained by different execution replicas in an ODRC cell.

In order to investigate the impact of cross-border requests, we examine the following two state-object assignment strategies in our evaluation in Section 6.7.2:

- The *round-robin* strategy distributes state objects without taking dependencies into account; that is, the first file or directory to be created is assigned to the execution replicas E_0 to E_f, the second one is assigned to the execution replicas E_{f+1} to E_{2f+1}, and so on.
- The *locality* strategy is based on the insight that the two objects accessed by NFS operations like `CREATE`, `REMOVE`, and `LOOKUP` are a file and its parent directory. Therefore, this strategy assigns both state objects to the same subset of execution replicas; directories are assigned in a round-robin fashion.

Due to the fact that the round-robin strategy does not exploit any information on the characteristics of the NFS protocol, we consider it a worst-case approach when it comes to eliminating cross-border requests in practice; theoretically, one could imagine a strategy that deliberately assigns state objects that are accessed together to different execution replicas, but we consider such a strategy to be of no practical interest. In contrast, although relying on a relatively simple heuristic, we expect the locality strategy to significantly reduce the number of cross-border requests, resulting in an increase in performance. Nevertheless, as a directory and its parent directory may still be assigned to different execution replicas, even for this strategy cross-border requests might occur.

6.7.2 Evaluation

In this section, we present an evaluation of ODRC$_{NFS}$ under benign conditions as well as in the presence of faults. The main goal of our experiments is to study the impact of ODRC's two key concepts: selective request execution and on-demand replica consistency. To this end, we compare ODRC$_{NFS}$ to different traditional system configurations.

6.7.2.1 Environment

We evaluate ODRC$_{NFS}$ using a cluster of dual-core servers (2.4 GHz, 2 GB RAM) for the ODRC cell and a cluster of quad-core servers (2.4 GHz, 8 GB RAM) running the clients; all servers are connected with switched Gigabit Ethernet. Throughout the evaluation, execution replicas rely on 32 NFS server daemons each and a block size of 4 kilobytes for both reads and writes. In our experiments, we use the following two system configurations of ODRC:

- **ODRC$_4$** is a regular-cell system configuration comprising four physical servers, each hosting an agreement replica, a selector, and an execution replica. The agreement stage of ODRC$_4$ runs the PBFT protocol [34] (see Section 3.1.1) to order client requests. Furthermore, ODRC$_4$ supports concurrent execution of requests, as discussed in Section 6.7.1.

- **ODRC$_6$** is an extended-cell system configuration (see Section 6.6.2) adding two physical servers to ODRC$_4$. Each of the servers is running its own selector and execution replica; selectors on these servers learn the total order of client requests to be processed from agreement replicas hosted on other servers in the cell.

ODRC$_4$ and ODRC$_6$ allow us to evaluate the throughput scalability offered by selective request execution and on-demand replica consistency. In addition to the two ODRC variants, we make use of the following three system configurations in our experiments:

- **NFS** is an off-the-shelf implementation of the network file system. Due to comprising only a single server, this system configuration is not resilient against crashes and/or Byzantine faults.

- **BFT$_4$** serves as a baseline representing a traditional Byzantine fault-tolerant system that only consists of an agreement stage and an execution stage; that is, BFT$_4$ processes all client requests on all execution replicas after the requests have been agreed on. Like ODRC$_4$, BFT$_4$ supports multi-threaded execution of requests.

- **SPEC$_4$** is a Byzantine fault-tolerant system configuration that uses speculative execution: As proposed by Zyzzyva [96], all SPEC$_4$ execution replicas process a state-modifying client request after one agreement-protocol phase and send the reply back to the client; the client accepts the result to be correct as soon as all (for read-only requests: three) of the four execution replicas have provided matching replies. As discussed in Section 6.1.1, Kotla et al. [96] argue that speculative execution represents a good means to increase the performance of a Byzantine fault-tolerant system during normal-case operation. In our evaluation, we use SPEC$_4$ to compare this approach against the performance benefits provided by selective execution and on-demand replica consistency in benign situations; as a consequence, it is sufficient for SPEC$_4$ to only implement the fast path of the Zyzzyva protocol that is optimized for the absence of both faults and packet losses.

Comprising four agreement replicas and at least four execution replicas each, all fault-tolerant system configurations evaluated are resilient against one Byzantine fault. In order to minimize the impact of implementation-specific factors on evaluation results, the prototypes of ODRC$_4$, ODRC$_6$, BFT$_4$, and SPEC$_4$ share as much code as possible. As a consequence, differences in experiment results are not caused by heterogeneous code bases but by actual differences in system configurations.

6.7.2.2 Normal-case Operation

In this section, we present experiments measuring the impact of selective request execution in combination with on-demand replica consistency on the throughput and response time of ODRC$_{\text{NFS}}$ in the absence of faults. Please refer to Section 6.7.2.3 for an evaluation of ODRC$_{\text{NFS}}$'s fault-handling procedures.

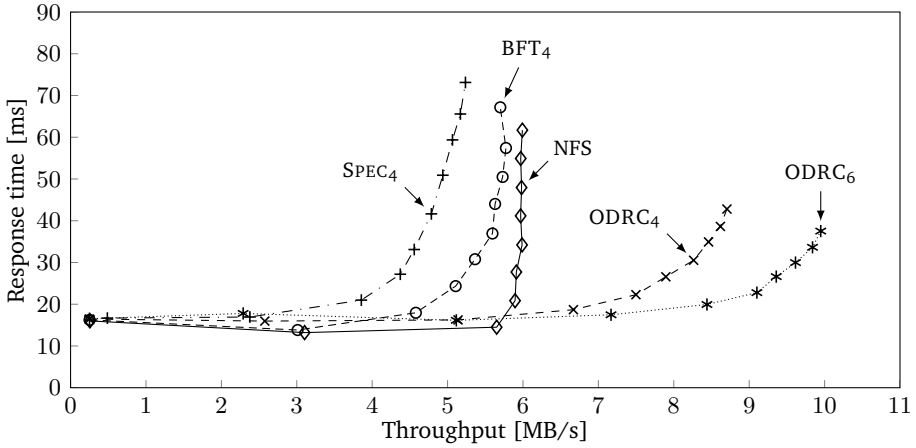


Figure 6.10: Throughput versus response time for different network file-system configurations for writes of 4 kilobytes: Applying selective request execution allows ODRC$_4$ to efficiently utilize the resources available, resulting in a performance increase that even outweighs the overhead for Byzantine fault-tolerant state machine replication. Extending a regular ODRC cell with additional execution replicas, as realized in the ODRC$_6$ system configuration, offers further improvements.

6.7.2.2.1 Microbenchmark

With each request having to be processed by all execution replicas in order to ensure consistency, write operations are usually a limiting factor in traditional Byzantine fault-tolerant network file systems. To evaluate the throughput and response time for write operations in ODRC$_{NFS}$, we conduct an experiment in which clients continuously write data in blocks of 4 kilobytes to separate files in a directory exported by the file system.

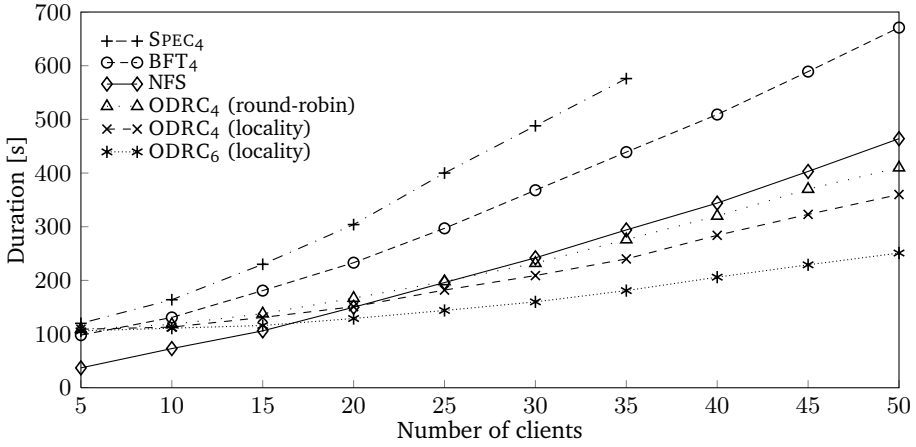
Preliminary Remarks Comprising exclusively write operations, which in ODRC$_{NFS}$ only access a single state object, the benchmark workload does not lead to any cross-border requests. In consequence, we omit evaluating ODRC$_{NFS}$ with different state-object distribution strategies designed to minimize the number of cross-border requests (see Section 6.7.1). Instead, in this experiment, ODRC$_{NFS}$ is configured to use the round-robin strategy that leads to each execution replica in ODRC$_4$ maintaining half (in ODRC$_6$: a third) of all files as maintained state objects.

Figure 6.10 shows the results of the experiment gained by varying the number of clients concurrently issuing write operations. Note that due to the fact that the file-system implementations used provide NFSv2 [145], and can therefore handle only small block sizes, the absolute throughputs realized are relatively small; later NFS versions allow larger block sizes and consequently lead to higher throughputs. However, as the primary goal of our evaluation is to compare selective request execution and on-demand replica consistency to traditional approaches, we are more interested in differences between the system configurations evaluated than in absolute values.

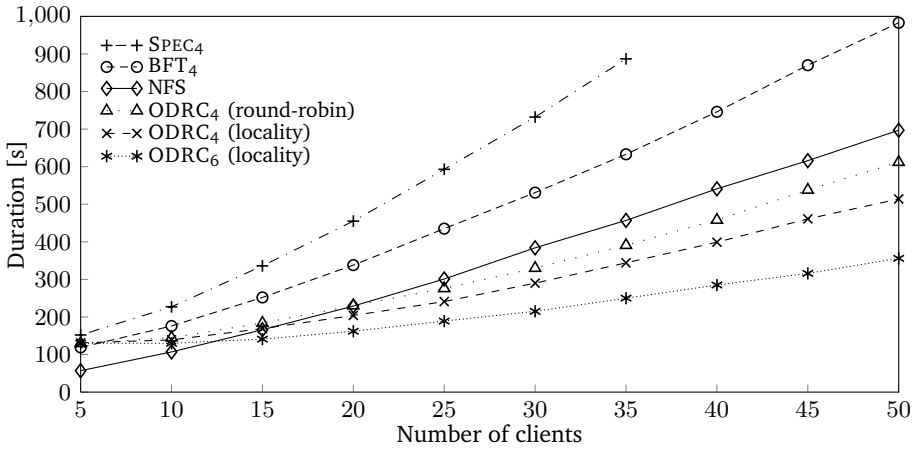
Comparison between ODRC and the Baseline The results of the benchmark show that for a small number of clients, the baseline system configuration BFT_4 achieves slightly better response times than both ODRC variants. The reason for the lower response times for small workloads in BFT_4 stems from differences in voting conditions at the client: An ODRC client completes result verification as soon as both of the execution replicas processing the client's request have provided the correct reply. In contrast, a BFT_4 client only needs to wait until it has obtained the correct replies of the two fastest of the four execution replicas in the cell. This way, BFT_4 is able to compensate individual delays introduced by the agreement stage and other replication overhead that may vary between different replicas. However, our measurement results show that this advantage of traditional Byzantine fault-tolerant systems is negligible for higher workloads.
The baseline system configuration BFT_4 reaches a maximum write throughput of about 5.7 MB/s. Applying selective request execution, $ODRC_4$ minimizes the individual load on execution replicas and is therefore able to utilize available resource more efficiently, resulting in improvements in throughput (about 53%) and response time (about 30%) compared to BFT_4 while comprising the same number of both agreement replicas and execution replicas. Relying on two additional servers, each hosting its own execution replica, $ODRC_6$ achieves a maximum throughput of about 9.9 MB/s (an increase of 74% over BFT_4) at further improved response times.

Comparison between ODRC and Unreplicated NFS Efficiently utilizing the resources available not only allows ODRC to increase performance compared with traditional Byzantine fault-tolerant systems: As our measurement results for the microbenchmark show, ODRC is also able to perform better than an unreplicated system providing the same service; that is, for the NFS use case, the benefits of applying selective request execution outweigh the combined overhead for state-machine replication and Byzantine fault-tolerant agreement. While the unreplicated NFS server is required to process all requests that are issued by clients, execution replicas in $ODRC_4$ statistically only need to handle every second (in $ODRC_6$: every third) client request. In consequence, an ODRC cell is still able to increase throughput at a point where the unreplicated NFS server has already reached saturation, eventually leading to a 45% ($ODRC_4$) and 65% ($ODRC_6$) higher maximum throughput, respectively.

Comparison between Selective Request Execution and Speculative Execution Analyzing the results for $SPEC_4$, we can conclude that speculative execution for the NFS use case in our setting does not offer any benefits over using traditional agreement-based state-machine replication: As the agreement stage only contributes about 1 to 4 milliseconds to the overall response time observed at the client, which is more than 14 milliseconds, most of the time is spent in the execution stage. Improving agreement latency by using speculative execution therefore has little effect on the overall response time. In contrast, as a client's voter in $SPEC_4$ is required to wait for matching replies of all four execution replicas, the slowest execution replica dictates performance. Selective request execution as applied by ODRC, on the other hand, targets at reducing load at the execution stage and therefore leads to significant response time and throughput improvements.



(a) Results of the read-mostly experiment



(b) Results of the write-mostly experiment

Figure 6.11: Results of the Postmark benchmark for different system configurations in dependence of the number of clients for two experiments where (a) reads and (b) writes are favored during the benchmark's transaction phase: The results confirm the findings of Section 6.7.2.2.1 that for medium and high workloads, ODRC performs better than an unreplicated NFS server and a traditional Byzantine fault-tolerant system (BFT$_4$). In addition, the numbers show that by eliminating almost all cross-border requests, the locality strategy leads to further improvements.

6.7.2.2.2 Postmark

The results gained from the microbenchmark experiments in Section 6.7.2.2.1 indicate that selective request execution is an effective means to increase the performance of ODRC$_{NFS}$. In this section, we use the Postmark [91] benchmark to evaluate whether this assessment is also true for non-trivial application use cases that involve cross-border requests. Postmark has been designed to model the file-server usage pattern of Internet

services such as email or web-based commerce. We use the same Postmark configuration as Kotla et al. [97] in their evaluation of CBASE-FS (see Section 6.7.1) and run a *read-mostly* experiment where the transaction phase of the benchmark favors reads over writes as well as a *write-mostly* experiment where reads are dominated by writes.

Analysis of State-object Distribution Strategies Figure 6.11 shows the durations of the Postmark experiments; smaller numbers indicate better performance. Focusing on the results of $ODRC_4$, we see that for both experiments benchmark durations are affected by the state-object distribution scheme (see Section 6.7.1) applied: Using the round-robin strategy, which does not exploit any knowledge about object dependencies, about 11% of all client requests become cross-border requests, requiring selectors to trigger update operations for unmaintained state objects. In contrast, by assigning a file and its parent directory to the same subset of execution replicas, the locality strategy is able to eliminate almost all cross-border requests, enabling $ODRC_4$ to run the read-mostly experiment in about 12% less time (write-mostly: 14%) compared with the round-robin strategy.

Comparison between ODRC and Other System Configurations Overall, the results of the Postmark experiments show a similar picture as the microbenchmark results in Section 6.7.2.2.1: The overhead for Byzantine fault-tolerant agreement as well as an execution stage that processes all client requests on all execution replicas cause the baseline system configuration BFT_4 to take (about 50%) longer to complete the experiments than an unreplicated NFS server that does not provide any resilience against Byzantine faults. Furthermore, relying on speculative execution does not provide any benefits in $SPEC_4$, but instead degrades performance. In contrast, applying selective request execution in combination with the locality strategy allows $ODRC_4$ to reduce benchmark durations for medium and high workloads by 19% (read-mostly) and 25% (write-mostly) compared with NFS, as well as 44% (read-mostly) and 47% (write-mostly) compared with BFT_4. Note that the latter numbers are close to the theoretical optimum for a cell comprising four execution replicas of a 50% reduction in durations, which translates to a 100% increase in throughput. Comprising an extended execution stage, $ODRC_6$ is able to make use of the additional resources, completing both benchmarks in about 60% less time than the baseline system configuration BFT_4; the optimum for $ODRC_6$ is a reduction by 67%.

6.7.2.3 Fault Handling

The experiments presented in Section 6.7.2.2 have shown that $ODRC_{NFS}$ achieves increased performance during normal-case operation compared with traditional systems thanks to applying selective request execution and on-demand replica consistency. In this section, we investigate the consequences of implementing this optimistic approach in the presence of faults. In this context, we distinguish between two categories of faults: First, a *state-object fault* leads to corrupted replies being returned for all client requests accessing the faulty object on an affected execution replica. Second, an *execution-replica fault* causes an affected execution replica to not provide correct replies at all. In all

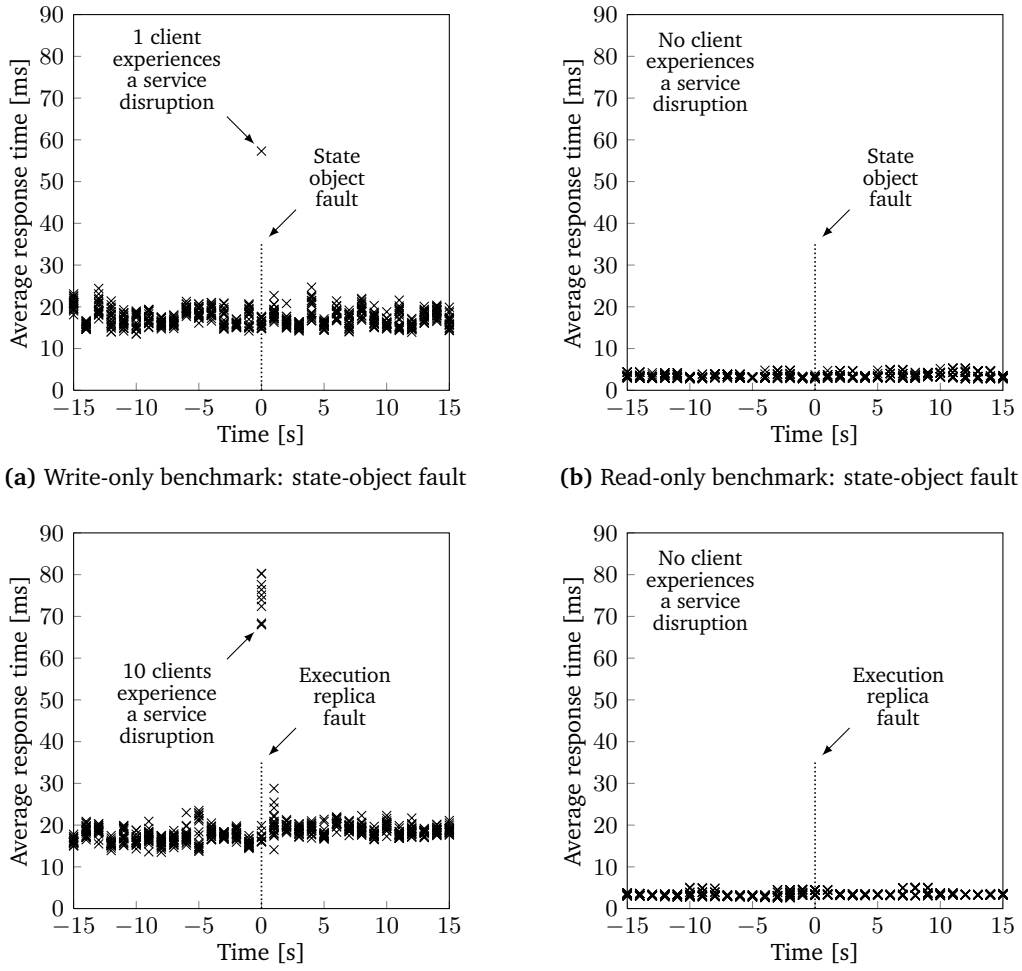
experiments presented in this section, we configure selectors to add the corresponding state objects to their local set of maintained objects after having learned about a stalled-verification notification in order to ensure a more efficient fault handling in the event of permanent faults (see Section 6.6.1.4).

6.7.2.3.1 Microbenchmarks

In this experiment, we repeat the write-only microbenchmark from Section 6.7.2.2.1 for ODRC$_4$ with 20 clients and artificially trigger faults during runtime. For comparison, we also conduct a read-only benchmark in which 20 clients continuously read data from separate files, using the same block size as for the write-only benchmark (i.e., 4 kilobytes).

Introducing Faults In both evaluation scenarios, we evaluate the worst-case scenario for an object checkpoint interval of $K = 100$ (see Section 6.3.4); that is, we trigger the fault at a point in time at which a checkpoint would be due. For the write-only benchmark, this means that a selector assisting in fault-handling procedures first has to process the latest 100 write operations for each of the files affected before its local execution replica is able to provide a reply to the client request in question.
For the read-only benchmark, we enable selectors to use an optimization presented in Section 6.6.1.2 that results in no additional overhead for bringing the unmaintained state objects involved up to speed: As stated by Kotla et al. [97], the `READ` operation in NFS updates the last-accessed-time attribute of a file; in consequence, `READ` is an operation that modifies the service state. However, as consecutive `READ` requests accessing the same file overwrite each others state modifications, during fault handling, only the latest `READ` operation actually has to be performed in order to bring the last-accessed-time attribute up to date. In our experiment, this operation is the one triggered by the client request for which result verification has stalled.

Impact of a State-object Fault Figure 6.12a shows that, after a state-object fault has occurred during the write-only benchmark, ODRC$_4$ is not able to provide the voter of an affected client with enough replies to make progress for about one second, resulting in a significant peak in average response time. During this time, non-faulty execution replicas not maintaining the file affected are updating their local copies of the file. As soon as this procedure is complete, the execution replicas are prepared to process the pending client request and to provide the deciding replies allowing the client to verify the result.
Note that in the write-only benchmark experiment only the client accessing the corrupted file copy observes a service disruption as a result of the state-object fault; all other clients performing read operations on different files do not experience any delays. Focusing on the results of the read-only benchmark (see Figure 6.12b), we see a similar picture. However, in this case, even the client directly affected by the state-object fault does not observe a notable increase in response times. This is due to the fact that the execution replicas assisting in fault handling are already prepared to step in without having to replay any state modifications first.



(a) Write-only benchmark: state-object fault

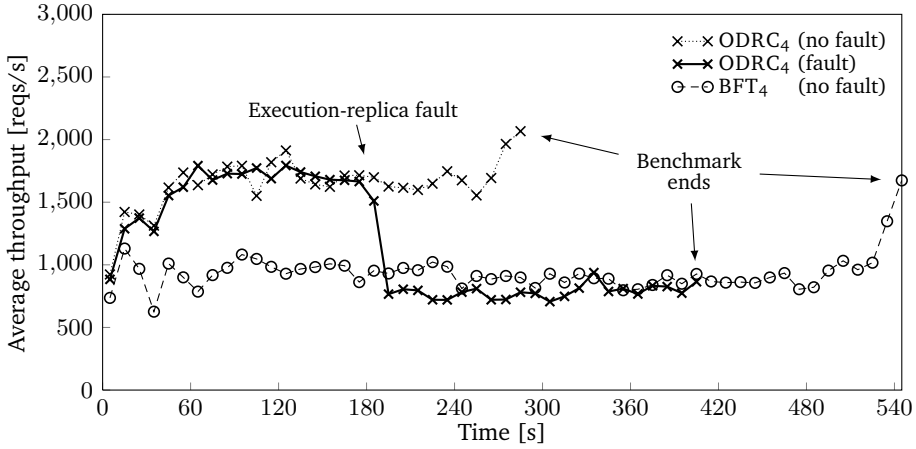
(b) Read-only benchmark: state-object fault

(c) Write-only benchmark: replica fault

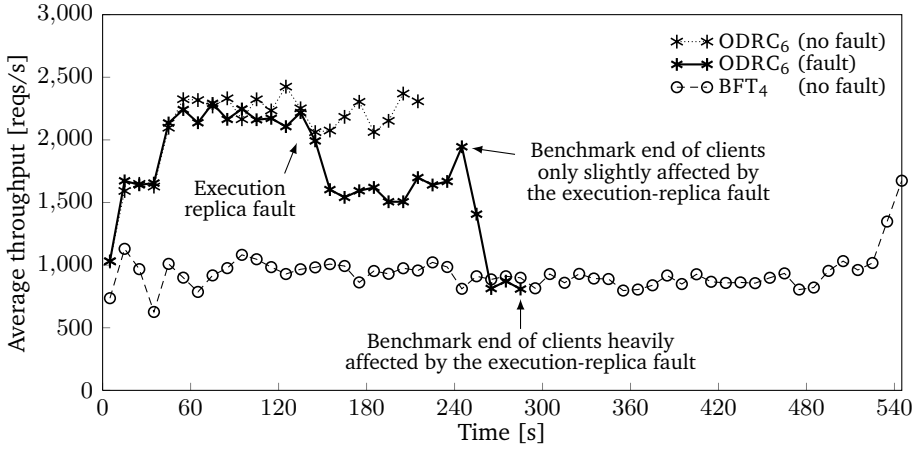
(d) Read-only benchmark: replica fault

Figure 6.12: Impact of faults on the average response time (1 sample/s) of $ODRC_4$ for different network file-system microbenchmarks with 20 clients: When a fault occurs, only clients accessing files affected observe a service disruption. After fault handling is complete and file copies have been updated on additional execution replicas, response times drop for subsequent requests.

Impact of an Execution-replica Fault Figures 6.12c and 6.12d indicate that the failure of an execution replica has a similar impact on $ODRC_4$'s system performance as multiple concurrent state-object faults: In the write-only benchmark, half of the clients experience a service disruption due to their files being maintained by the faulty execution replica. In the course of fault handling, selectors on the other execution replicas add the files affected to their local sets of maintained objects. In consequence, response times drop again for subsequent client requests. However, after an execution-replica fault, $ODRC_4$'s average response times do not reach the low levels provided during normal-case operation due to an increase in individual request load on the remaining execution replicas.



(a) Regular cell comprising four execution replicas



(b) Extended cell comprising four agreement replicas and six execution replicas

Figure 6.13: Impact of an execution-replica fault on the average throughput (1 sample/10 s) of ODRC$_4$ and ODRC$_6$ for the Postmark write-mostly benchmark with 30 clients: After the fault, the update procedure for a file affected is initiated on the first access to the file; with different files being accessed at different times, fault handling is distributed over a longer period of recovery.

6.7.2.3.2 Postmark

In order to study the impact of an execution-replica fault on ODRC$_4$ in a practical use-case scenario, we induce a fault during the transaction phase of a write-mostly Postmark experiment with 30 concurrent clients; we repeat the experiment with ODRC$_6$ to investigate the effects of having an extended execution stage.

Impact of an Execution-replica Fault in ODRC$_4$ Figure 6.13a shows the average combined throughput of all clients of ODRC$_4$ for this experiment in comparison to measurement results gained from normal-case operation experiments conducted with BFT$_4$ and

$ODRC_4$. About three minutes after the start, when the execution-replica becomes faulty, the throughput of $ODRC_4$ drops by about 50% due to the files affected by the fault being restored on non-faulty execution replicas. With the update of a file being initiated on the first access to the file after the execution-replica fault has occurred, the impact of fault handling is not concentrated to a single point in time. Instead fault-handling procedures for different files are distributed over a longer period of recovery, which in this case lasts about two minutes, allowing $ODRC_4$ to keep throughput performance close to the level achieved by the baseline system configuration BFT_4 in the absence of faults. With more and more files becoming up to date, throughput steadily increases during this phase, eventually matching BFT_4's normal-case performance.

Impact of an Execution-replica Fault in $ODRC_6$ The extended-cell system configuration $ODRC_6$ comprises more execution replicas than actually required to tolerate a single Byzantine fault. In consequence, not all execution replicas in the cell are required to participate in fault handling. In the $ODRC_6$ experiment we exploit this fact to improve resource efficiency by disabling fault-handling procedures for each file on two of six execution replicas. Figure 6.13b shows the benefits of this optimization: After the execution-replica fault, the throughput of $ODRC_6$ only decreases by about 30%, remaining well above the normal-case performance of BFT_4. Furthermore, some clients are able to complete the experiment in almost the same time as under benign conditions due to the fact that the files they access are maintained by execution replicas that play a minor role in fault handling. With the remaining clients in total issuing less requests, after about 255 seconds, system throughput drops to a lower level, staying there until all clients have completed the experiment.
In general, fault handling in $ODRC_6$ is not as expensive as in $ODRC_4$: An $ODRC_6$ execution replica maintains only a third of all state objects compared to an $ODRC_4$ execution replica, which is responsible for keeping half of all state objects up to date (see Section 6.7.2.2.1). Therefore, in case of an execution-replica fault, only a third of the service state in $ODRC_6$, compared to half in $ODRC_4$, has to be restored on other execution replicas in the cell, resulting in less fault-handling overhead.

6.8 $ODRC_{ZooKeeper}$: A Byzantine Fault-tolerant Coordination Service

In large-scale infrastructures, applications may be distributed across a large number of nodes. With coordination of processes becoming an inherently difficult tasks in such environments, many state-of-the-art systems (e.g., BigTable [36], Google File System [75], and MapReduce [116]) rely on external coordination services [22, 30, 84] for tasks like leader election, group membership, distributed synchronization, and configuration maintenance. As a result, a coordination service becomes an essential component whose resilience against faults is crucial for the well-functioning of an entire system. In this section, we present $ODRC_{ZooKeeper}$, a Byzantine fault-tolerant coordination service we have implemented by integrating the ZooKeeper [84] coordination middleware, which in its existing form is only resilient against crashes, with ODRC.

6.8.1 Service Integration

For the integration of ZooKeeper into ODRC, we used a similar approach as for the integration of CBASE-FS in the context of the NFS use case presented in Section 6.7.1. Therefore, in the following, we limit our discussion on ZooKeeper-specific aspects.

Prototype The architecture of ODRC$_{\text{ZooKeeper}}$ shares great similarities with the architecture of ODRC$_{\text{NFS}}$ (see Figure 6.9): In order to use the service, an application process invokes an operation at a client library; for ODRC$_{\text{ZooKeeper}}$, we modified the already existing ZooKeeper library to serve as the client of a Byzantine fault-tolerant state-machine replication protocol. At the server side, an ODRC$_{\text{ZooKeeper}}$ execution replica comprises the application logic of the original ZooKeeper server implementation as well as a wrapper component responsible for ensuring determinism by relying on consistent timestamps (see Section 4.9.2.2); please refer to Clement et al. [42] for details on how to enforce deterministic ZooKeeper replicas in the context of Byzantine fault tolerance.

Selector Implementation ZooKeeper manages information in *nodes* in a hierarchical tree. As in a file system, each node is uniquely identified by a path; however, in contrast to a file system, all nodes in ZooKeeper are able to store user data, even nodes that themselves have child nodes. For ODRC$_{\text{ZooKeeper}}$, we define an ODRC state object to be a single ZooKeeper node. Furthermore, in order for a selector to determine which state objects are accessed by a request, we implement a request analysis function (see Section 6.2.2.2) that extracts the path information present in all ZooKeeper client requests.

Fault-handling Optimization Data assigned to a node in ZooKeeper is always read and written atomically and in its entirety. As a result, a write operation completely replaces the data set by previous write requests; the only indication that other write operations to a particular node have ever occurred is a node-specific version counter that is incremented each time new data is assigned. Using a technique discussed in Section 6.6.1.2, by providing such application-specific knowledge about write operations in ZooKeeper to selectors, we are able to optimize fault-handling procedures in ODRC$_{\text{ZooKeeper}}$: Instead of processing multiple subsequent write requests that access the same node, in order to bring the node's state up to date, a selector may only process the latest write request and adjust the value of the version counter to reflect the correct version number. This way, the updating process is sped up due to fewer request having to be executed.

Consistency Guarantees As in traditional crash-tolerant systems, state modifications in ZooKeeper are applied in a consistent manner on all servers in a cell; in contrast, a read-only operation is exclusively executed by the server that is connected to the corresponding client. In order to increase performance, however, a ZooKeeper server processes a read-only request right after having received it, performing no additional coordination with state modifications triggered by other clients. As a result, ZooKeeper is only able to provide relaxed consistency guarantees. ODRC$_{\text{ZooKeeper}}$, on the other hand, offers strong consistency due to relying on ODRC's agreement stage that defines a total order on all client requests, independent of whether they read or write the state of a service.

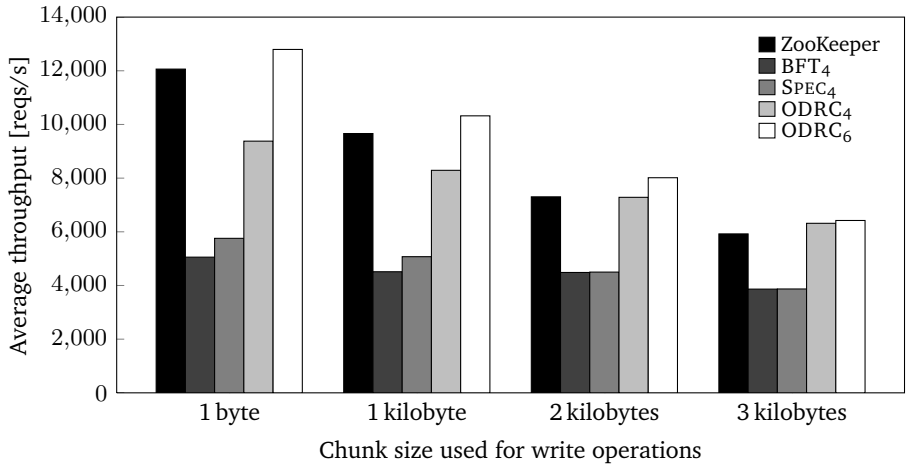


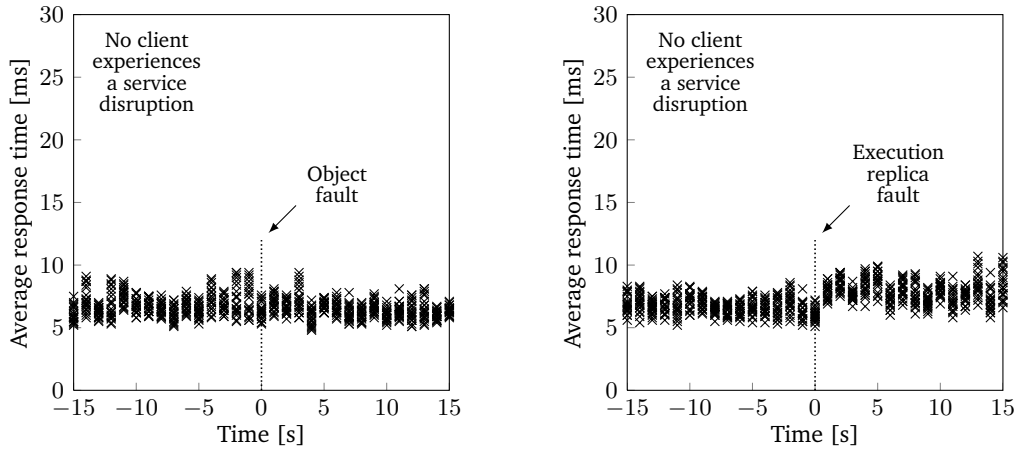
Figure 6.14: Throughput results for different system configurations for a write-only ZooKeeper microbenchmark in which clients repeatedly assign new data to nodes using different chunk sizes: With less data to be distributed across the cell, small chunk sizes put less load on the agreement stage, allowing ODRC to achieve higher improvements by performing selective request execution.

6.8.2 Evaluation

We conduct our evaluation of $\text{ODRC}_{\text{ZooKeeper}}$ on the same clusters of machines as the evaluation of ODRC_{NFS} in Section 6.7.2.1. All system configurations evaluated are dimensioned to tolerate a single (Byzantine) fault; in consequence, the unmodified crash-tolerant ZooKeeper setting is distributed over three servers. In order to minimize the effects caused by differences in the consistency guarantees provided, stemming from a different handling of read-only requests in ZooKeeper and $\text{ODRC}_{\text{ZooKeeper}}$ (see Section 6.8.1), our experiments focus on write-only workloads.

6.8.2.1 Normal-case Operation

We evaluate the write throughput of different system configurations during normal-case operation using clients that repeatedly assign new data to nodes. Due to the absence of cross-border requests during the runtime phase of the benchmark, we configure ODRC selectors to apply a round-robin strategy that equally assigns nodes to execution replicas without taking dependencies into account (compare Section 6.7.1).
Figure 6.14 shows the evaluation results for a variety of typical (i.e. relatively small [84]) chunk sizes of ZooKeeper write operations. For small writes of one byte, applying selective request execution enables ODRC_4 to achieve an 85% improvement over the baseline system configuration BFT_4. Increasing the chunk size of writes puts more load on the agreement stage as, besides establishing a global total order on requests, it is also responsible for distributing the contents of requests to all agreement replicas. As a result, the impact of the execution stage and, consequently, the benefits of selective request ex-



(a) Write-only benchmark: state-object fault

(b) Write-only benchmark: replica fault

Figure 6.15: Impact of a state-object fault and an execution-replica fault on the average response time (1 sample/s) of ODRC$_4$ for a write-only ZooKeeper workload from 20 clients: With object states in ZooKeeper being relatively small, in both cases, the overhead for fault-handling procedures in ODRC$_{\text{ZooKeeper}}$ is negligible, resulting in no observable service disruption at the client.

ecution decrease, resulting in ODRC$_4$ to provide a throughput increase of only 63% over BFT$_4$ for writes of 3 kilobytes. For the same reason, the improvements of using an extended cell comprising additional execution replicas in ODRC$_6$ are higher for workloads with small chunk sizes. Note that, as in the network file-system experiments presented in Section 6.7.2, relying on speculation does not lead to substantial performance gains in SPEC$_4$ due to the fact that clients can only make progress after having obtained matching replies from all four execution replicas.

6.8.2.2 Fault Handling

In the following, we investigate the worst-case impact of both a state-object fault, which is limited to a single data node, as well as an execution-replica fault, which affects the copies of all nodes maintained by the faulty replica. In each experiment, we artificially induce a fault during the runtime of a write-only benchmark with 20 clients using the ODRC$_4$ system configuration.

Figure 6.15a shows that in contrast to ODRC$_{\text{NFS}}$ (see Section 6.7.2.3.1), a state-object fault in ODRC$_{\text{ZooKeeper}}$ leads to no noticeable service disruption at the client. The reason for this property is the fault-handling optimization discussed in Section 6.8.1: In order to apply the state modifications of multiple write operations to an unmaintained node, it is sufficient for a selector to only process a single write request. As a result, the overhead for on-demand replica consistency in ODRC$_{\text{ZooKeeper}}$ is small, as also shown by the results for

the execution-replica fault scenario (see Figure 6.15b). Here, the only observable effect of the fault is an increased average response time caused by the fact that, with fewer non-faulty execution replicas remaining, each replica faces a higher individual load.

6.9 Discussion

In the following, we elaborate on the effects of cross-border requests on the overhead of on-demand replica consistency and discuss how to deal with a number of functional and non-functional limitations of ODRC. Furthermore, we present similarities and differences between ODRC and SPARE that have an influence on fault-handling latencies in both systems. Finally, based on the results obtained from the experiments conducted with ODRC_{NFS} and $\text{ODRC}_{\text{ZooKeeper}}$, we discuss implications for other application use cases.

6.9.1 Overhead of On-demand Replica Consistency

The results of our evaluations of ODRC_{NFS} in Section 6.7.2 and $\text{ODRC}_{\text{ZooKeeper}}$ in Section 6.8.2 have shown that selective request execution in combination with on-demand replica consistency can be an effective means to increase the performance of a Byzantine fault-tolerant system. Furthermore, our experiments have identified a decisive factor with regard to the overhead of on-demand replica consistency: cross-border requests. Due to interfering with the goal of processing each client request on only a minimum subset of execution replicas, cross-border requests can limit the extent to which selective request execution can be applied efficiently. In consequence, in order to benefit from this technique, the fraction of cross-border requests should be as small as possible.
Note that the fraction of cross-border requests is usually not an intrinsic property of a service. Instead, it depends to a great extent on both the state-access pattern of a service and on the state-object distribution scheme used in ODRC selectors. As shown in Section 6.7 by means of the locality strategy, the occurrence of cross-border requests can be significantly reduced by assigning groups of state objects with dependencies to the same subset of execution replicas. In general, we can conclude that the task of keeping the fraction of cross-border requests small becomes more simple the fewer dependencies between state objects exist in a particular service.

6.9.2 Limitations

As discussed in Section 6.2.2, ODRC makes certain assumptions on the composition of the service state as well as the information contained in client requests. Although most of these requirements are not very restrictive and therefore met by a large spectrum of existing applications, a service that does not provide the features required is disqualified from applying on-demand replica consistency if there is no possibility to modify its implementation accordingly. On the other hand, meeting all functional criteria does not necessarily mean that a service is able to benefit from being integrated with ODRC. As discussed in Section 6.9.1, a large number of state-object dependencies, and consequently a large

fraction of cross-border requests, is likely to limit the performance improvements achievable with our approach. Furthermore, we expect to see only small performance gains for application scenarios in which the response-time overhead for Byzantine fault-tolerant agreement dominates the processing time of client requests: Treating the agreement stage as a black box (see Section 6.2.1), ODRC only takes effect at the execution stage. Nevertheless, even in such use cases applying selective request has advantages, for example, due to minimizing the number of messages that are exchanged between clients and servers over the network.

6.9.3 Fault-handling Efficiency

With ODRC being designed to improve performance during normal-case operation (see Section 6.1.1), clients may experience a service disruption while accessing state objects that are the subject of fault-handling procedures currently in progress. In Section 6.6.1, we have discussed several approaches to minimize the fault-handling latency of ODRC. Leaving few application-specific techniques aside, there is a general tradeoff between high performance and low fault-handling latency in ODRC and its origin is the same as for the tradeoff between high resource savings and low fault-handling latency in SPARE (see Section 4.12.1): Omitting request executions that are unnecessary in the absence of faults leads to parts of the service state of an execution replica becoming outdated; this in turn results in a higher updating overhead and longer latencies during fault handling. However, note that there is an important difference between SPARE and ODRC: While SPARE relies on passive execution replicas that have to be activated first in order to be able to assist in fault handling (see Section 4.4.2), ODRC for this purpose uses execution replicas that are already active. In consequence, ODRC saves the overhead of waking up an execution replica from resource-saving mode, a procedure our evaluation in Section 4.11.5 has revealed to be responsible for a significant part of SPARE's fault-handling latency.

6.9.4 Transferability of Results

Our experiments have revealed that certain characteristics allow a network file system as well as a coordination service to profit from the combination of selective request execution and on-demand replica consistency. In consequence, we expect other applications sharing the same characteristics to benefit to a similar extent from our approach: As discussed in Section 6.7.2.2.1, for example, response times in ODRC_{NFS} are influenced in great part by the execution stage; as a result, a reduction of individual execution-replica load in ODRC is effective. Comparably long processing times are a property of many non-trivial service applications. With regard to $\text{ODRC}_{\text{ZooKeeper}}$, our experiments in Section 6.8.2.2 have shown that a limited interface, which only provides write methods modifying the content of a state object in its entirety, is beneficial due to enabling an optimized fault handling. Interfaces with such properties, for example, can also be found in key-value stores [55, 68].

Throughout our evaluation of ODRC, we have relied on execution-replica implementations that support concurrent execution of client requests [97]. In contrast to execution replicas requiring requests to be processed sequentially, which are used in the majority of existing Byzantine fault-tolerant systems [34, 35, 41, 79, 152, 153, 154, 161], ODRC's execution replicas are capable of achieving a higher maximum throughput. However, as selective request execution does not make any assumptions on whether an execution replica processes requests concurrently or sequentially, we expect existing Byzantine fault-tolerant system implementations to also benefit from the concept: For a cell providing the same overall throughput, our approach minimizes the number of requests an individual execution replica has to execute. As a result, operating execution replicas at their maximum while applying selective request execution leads to an increase in overall system throughput. Therefore, the absolute improvements achievable may be smaller in existing system implementations, but the relative performance increase is likely to be similar as in ODRC.

6.10 Chapter Summary

In this chapter, we have shown that Byzantine fault tolerance does not necessarily come with a performance penalty. Instead, the additional resources a system requires in order to provide Byzantine fault tolerance can be utilized in parts to increase performance. This way, as our evaluation results underline, Byzantine fault-tolerant systems can be built that achieve better performance for medium and high workloads than their unreplicated counterparts running the same service.

Besides revealing that Byzantine fault-tolerant systems have a hidden performance potential, in this chapter, we have also presented a methodology to unlock it: In contrast to traditional systems that process all client requests on all execution replicas in a cell, our approach limits request execution to the smallest subset of execution replicas necessary to make progress during normal-case operation. This way, resources that in traditional systems are used to perform redundant work become available and can be utilized to process additional requests. Although being designed with a focus on the absence of faults, our approach does not impair a system's ability to tolerate Byzantine faults. In order to guarantee this property, we presented a switching mechanism that ensures the consistency of execution replicas on demand, that is, at the time and to the extent required for fault-handling procedures.

7

Summary, Conclusions, and Further Ideas

In this thesis, we have investigated how to save resources in fault and intrusion-tolerant systems by handling the normal case and the worst case separately. In this chapter, we give an overview of our findings presented in previous chapters and summarize the use of passive replication as a central building block for Byzantine fault-tolerant systems. Furthermore, we outline possible directions for future research in this area.

7.1 Summary and Conclusions

High resource usage has been identified as one of the main reasons why Byzantine fault-tolerant systems have not yet been broadly adopted by industry [80, 101]. So far, most research aiming to address this problem has focused on ways that minimize the resource consumption of a fault and intrusion-tolerant system by reducing the minimum number of replicas required to ensure safety [41, 152, 154].

Basic Approach In this thesis, we have explored an approach (which is orthogonal to previous works) that relies on different operation modes to increase resource efficiency: Instead of always consuming the amount of resources necessary to tolerate the maximum number of faults a system has been dimensioned for, we have proposed a normal-case operation mode, in which a system reduces its resource usage to a level that only guarantees progress under benign conditions; assuming that faults are rare [3, 50, 77, 78, 79, 159] this is the mode a system executes most of the time. In order to actually tolerate faults, the system switches to a second mode, in which additional resources are allocated.

Research Questions Starting from the basic idea of relying on multiple operation modes, we have investigated different alternatives to increase the resource efficiency of a Byzantine fault-tolerant system. In this context, we have followed two main research questions:

- Can the approach enable a Byzantine fault-tolerant system to provide the **same performance** by utilizing **less resources**?
- Can the approach enable a Byzantine fault-tolerant system to provide a **better performance** by utilizing an **equal amount of resources**?

Regarding the first research question, our evaluations of SPARE and REBFT have shown that by applying passive replication at the granularity of replicas, it is possible to implement a normal-case operation mode that decreases the resource footprint of a system without impairing performance. In contrast, in REBFT, the reduced communication overhead at the agreement stage even leads to a performance increase. For both cases, our results also show that a switch between normal-case operation mode and fault-handling mode can be performed efficiently.
With regard to the second research question, our ODRC case studies have revealed that passive replication at the granularity of state objects can allow a system to free resources, which consequently may be used to process more requests. As confirmed by our experiments, applying this approach, the overall performance of a Byzantine fault-tolerant system is no longer bound by the performance of the corresponding unreplicated system providing the same service. Instead, a fault and intrusion-tolerant system is able to utilize the additional resources kept available for the handling of faults.

Summary Making use of different operation modes can be an effective means to increase the resource efficiency of a Byzantine fault-tolerant system in the absence of faults. As a result, the overall benefits in practice depend on the fraction of time a system is able to run in normal-case operation mode.

7.2 Contributions

The following list summarizes the most important contributions of this thesis:

- An extensive study on how making use of **different operation modes can increase the resource efficiency of fault and intrusion-tolerant systems**. In particular, the method of handling the normal case and the worst case separately, which has already been successfully applied to other categories of computer systems, has been identified as an effective means in this context.
- An evaluation of **passive replication as a central building block for Byzantine fault-tolerant systems**. The results presented in this thesis confirm that passive replication can serve as a basis for realizing efficient switching mechanisms that enable fault and intrusion-tolerant systems to transition from normal-case operation mode to fault-handling mode without major disruption.
- **SPARE**, a system design and implementation that relies on virtualization to provide resilience against Byzantine faults in user domains running the service application. Although addressing a more complex fault, SPARE consumes almost the same amount of CPU, network, and power as a comparable crash-tolerant system.
- The **REBFT** approach to implement a resource-saving mode for the normal case based on existing agreement protocols. In this thesis, the method has been applied to two traditional Byzantine fault-tolerant system architectures; in both cases, resource consumption in the absence of faults has been significantly reduced.
- **ODRC**, a system design and implementation proving that it is possible to build Byzantine fault-tolerant systems that achieve better performance than their unreplicated counterparts running the same service application. One of the main contributions in this context is ODRC's normal-case operation mode, which allows the system to reinvest the resources saved in order to process additional requests.

7.3 Further Ideas

In this thesis, we have developed and evaluated approaches that utilized different modes of operation in order to allow Byzantine fault-tolerant systems to adjust their resource usage to current conditions. With the focus of this work being on increasing the resource efficiency of fault and intrusion-tolerant systems during normal-case operation, in all the cases investigated in previous chapters, the presence of suspected or detected faults triggered reconfiguration procedures. Note that we expect resource savings to not be the only use case for relying on different operation modes in Byzantine fault-tolerant systems, and reacting to changes in fault conditions to not be the only reason for performing system reconfigurations [77]. In the following, we identify a number of problems that match one or both of these categories and outline how the mechanisms and techniques presented in this thesis may be modified and/or extended to address them.

Balanced Resource Usage Byzantine fault-tolerant agreement protocols like PBFT [34] rely on a single replica, the leader, to propose the client requests to be ordered (see Section 3.1.1). As a result of this asymmetric distribution of responsibilities, the leader usually uses more resources than the other replicas in the system. One solution to this problem is to rotate the leader role among different replicas [152, 153], thereby balancing the additional load. Combining such an approach with the replication mechanisms of SPARE or REBFT could enable systems to not only save resources during normal-case operation, but also to balance the load across different replicas. Besides reacting to faults, adapting to changes in the workload may be a reason for such systems to initiate reconfiguration procedures.

Capacity-aware Resource Usage In general, it is highly unlikely that all replicas belonging to the same fault and intrusion-tolerant system provide exactly the same performance. In particular, this is true if replicas run on different hardware platforms and/or if heterogeneous replica implementations are used in order to decrease the probability of common faults (see Section 2.1.1.2). Operating on the granularity of single state objects, ODRC offers the possibility to take differences in replica capacities into account. Utilizing this property, replicas with higher capacities could be configured to maintain more objects than replicas with lower capacities. Starting with such an asymmetric object assignment, variations in replica performances at runtime could then be addressed by dynamically reassigning the responsibility to maintain certain state objects.

Location-aware Resource Usage In a geo-replicated Byzantine fault-tolerant system, in which replicas are linked via a wide-area network, the locations at which resources are spent may be as important as the total amount of resources spent: For example, in such a scenario, processing requests on replicas with fast network connections to clients could minimize the time it takes the replies to reach their destinations. We believe that it is possible to take such considerations into account and still allow a fault and intrusion-tolerant system to be resource efficient: Extending the basic idea behind REBFT, a system could, for example, assign the roles of active and passive replicas based on knowledge about network latencies. This way, replicas close to the majority of clients might be selected as active replicas, while other replicas farther away remain passive. Note that such an assignment does not necessarily have to be static: If the system detects a change in client-access characteristics, it might trigger a dynamic reconfiguration of replica roles, a procedure that could be implemented as a mode switch.

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.

[2] Keith Adams and Ole Agesen. “A Comparison of Software and Hardware Techniques for x86 Virtualization.” In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’06)*. 2006, pages 2–13.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. “FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment.” In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI’02)*. 2002, pages 1–14.

[4] Peter A. Alsberg and John D. Day. “A Principle for Resilient Sharing of Distributed Resources.” In: *Proceedings of the 2nd International Conference on Software Engineering (ICSE’76)*. 1976, pages 562–570.

[5] Lorenzo Alvisi, Thomas C. Bressoud, Ayman El-Khashab, Keith Marzullo, and Dmitrii Zagorodnov. “Wrapping Server-Side TCP to Mask Connection Failures.” In: *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM’01)*. 2001, pages 329–337.

[6] Lorenzo Alvisi, Evelyn Tumlin Pierce, Dahlia Malkhi, Michael K. Reiter, and Rebecca N. Wright. “Dynamic Byzantine Quorum Systems.” In: *Proceedings of the 30th International Conference on Dependable Systems and Networks (DSN’00)*. 2000, pages 283–292.

[7] Amazon EC2. `http://aws.amazon.com/ec2/`.

[8] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. “Prime: Byzantine Replication Under Attack.” In: *IEEE Transactions on Dependable and Secure Computing* 8.4 (2011), pages 564–577.

[9] 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: *Transactions on Dependable and Secure Computing* 7.1 (2010), pages 80–93.

[10] Yair Amir and Jonathan Stanton. *The Spread Wide Area Group Communication System*. Technical report CNDS-98-4. Johns Hopkins University, 1998.

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. "A View of Cloud Computing." In: *Communications of the ACM* 53.4 (2010), pages 50–58.

[12] Algirdas Avižienis. "The N-version Approach to Fault-tolerant Software." In: *IEEE Transactions on Software Engineering* 12 (1985), pages 1491–1501.

[13] Algirdas Avižienis and Jean-Claude Laprie. "Dependable Computing: From Concepts to Design Diversity." In: *Proceedings of the IEEE* 74.5 (1986), pages 629–638.

[14] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. "Measurements of a Distributed File System." In: *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP'91)*. 1991, pages 198–212.

[15] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. "Shifting Gears: Changing Algorithms on the Fly to Expedite Byzantine Agreement." In: *Information and Computation* 97.2 (1992), pages 205–233.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization." In: *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP'03)*. 2003, pages 164–177.

[17] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing." In: *Computer* 40.12 (2007), pages 33–37.

[18] Claudio Basile, Zbigniew Kalbarczyk, and Ravi Iyer. "A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas." In: *Proceedings of the 33rd International Conference on Dependable Systems and Networks (DSN'03)*. 2003, pages 149–158.

[19] Claudio Basile, Keith Whisnant, Zbigniew Kalbarczyk, and Ravi Iyer. "Loose Synchronization of Multithreaded Replicas." In: *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*. 2002, pages 250–255.

[20] Diogo Behrens, Christof Fetzer, Flavio P. Junqueira, and Marco Serafini. "Towards Transparent Hardening of Distributed Systems." In: *Proceedings of the 9th Workshop on Hot Topics in System Dependability (HotDep'13)*. 2013, pages 13–18.

[21] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. "Deterministic Process Groups in dOS." In: *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*. 2010, pages 1–16.

[22] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Fraga. "DepSpace: A Byzantine Fault-tolerant Coordination Service." In: *Proceedings of the 3th European Conference on Computer Systems (EuroSys'08)*. 2008, pages 163–176.

[23] Alysson Neves Bessani, Hans P. Reiser, Paulo Sousa, Ilir Gashi, Vladimir Stankovic, Tobias Distler, Rüdiger Kapitza, Alessandro Daidone, and Rafael Obelheiro. "FOREVER: Fault/intrusiOn REmoVal through Evolution & Recovery." In: *Proceedings of the Middleware 2008 Conference Companion (Middleware'08, Poster Session)*. 2008, pages 99–101.

[24] Alysson Neves 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 (ATC'13)*. 2013, pages 169–180.

[25] BFT-SMaRt. `http://code.google.com/p/bft-smart/`.

[26] Sapan Bhatia, Murtaza Motiwala, Wolfgang Muhlbauer, Yogesh Mundada, Vytautas Valancius, Andy Bavier, Nick Feamster, Larry Peterson, and Jennifer Rexford. "Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware." In: *Proceedings of the 4th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'08)*. 2008, pages 427–432.

[27] Bill Wilkins. *Tips for Improving INSERT Performance in DB2 Universal Database*. `http://www.ibm.com/developerworks/data/library/tips/dm-0403wilkins/`.

[28] Thomas C. Bressoud and Fred B. Schneider. "Hypervisor-based Fault-tolerance." In: *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP'95)*. 1995, pages 1–11.

[29] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. "The Primary-Backup Approach." In: *Distributed Systems (2nd Edition)*. Addison-Wesley, 1993, pages 199–216.

[30] 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.

[31] Christian Cachin and Jonathan A. Poritz. "Secure Intrusion-tolerant Replication on the Internet." In: *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN'02)*. 2002, pages 167–176.

[32] 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.

[33] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance." In: *Proceedings of the 3th Symposium on Operating Systems Design and Implementation (OSDI'99)*. 1999, pages 173–186.

[34] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery." In: *ACM Transactions on Computer Systems* 20.4 (2002), pages 398–461.

[35] 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.

[36] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." In: *ACM Transactions on Computer Systems* 26.2 (2008), 4:1–4:26.

[37] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. "Managing Energy and Server Resources in Hosting Centers." In: *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*. 2001, pages 103–116.

[38] Liming Chen and Algirdas Avižienis. "N-version Programming: A Fault-tolerance Approach to Reliability of Software Operation." In: *Proceedings of 8th International Symposium on Fault-Tolerant Computing (FTCS-8)*. 1978, pages 3–9.

[39] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[40] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. "Diverse Replication for Single-Machine Byzantine-Fault Tolerance." In: *Proceedings of the 2008 USENIX Annual Technical Conference (ATC'08)*. 2008, pages 287–292.

[41] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. "Attested Append-only Memory: Making Adversaries Stick to their Word." In: *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*. 2007, pages 189–204.

[42] 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.

[43] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. "BFT: The Time is Now." In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)*. 2008, pages 81–84.

[44] 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.

[45] Vinicius Cogo, André Nogueira, João Sousa, Marcelo Pasin, Hans P. Reiser, and Alysson Neves 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.

[46] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John 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: *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI'12)*. 2012, pages 251–264.

[47] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. "Practical Hardening of Crash-tolerant Systems." In: *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*. 2012, pages 453–466.

[48] Miguel Correia, Nuno Ferreira Neves, Lau Cheuk Lung, and Paulo Veríssimo. "Worm-IT – A Wormhole-based Intrusion-tolerant Group Communication System." In: *Journal of Systems and Software* 80.2 (2007), pages 178–197.

[49] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems." In: *Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS'04)*. 2004, pages 174–183.

[50] Pedro Costa, Marcelo Pasin, Alysson Neves Bessani, and Miguel Correia. "Byzantine Fault-Tolerant MapReduce: Faults Are Not Just Crashes." In: *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science (CLOUDCOM'11)*. 2011, pages 32–39.

[51] 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.

[52] Mache Creeger. "Cloud Computing: An Overview." In: *ACM Queue* 7.5 (2009), pages 3–4.

[53] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. “Remus: High Availability via Asynchronous Virtual Machine Replication.” In: *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI’08)*. 2008, pages 161–174.

[54] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI’04)*. 2004, pages 137–150.

[55] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. “Dynamo: Amazon’s Highly Available Key-value Store.” In: *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP’07)*. 2007, pages 205–220.

[56] 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.

[57] 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.

[58] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. “Efficient State Transfer for Hypervisor-Based Proactive Recovery.” In: *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS’08)*. 2008, pages 7–12.

[59] 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.

[60] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. “Byzantine Fault Tolerance, from Theory to Reality.” In: *Computer Safety, Reliability, and Security*. Springer, 2003, pages 235–248.

[61] 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.

[62] eBay. `http://www.ebay.com/`.

[63] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. “Passive NFS Tracing of Email and Research Workloads.” In: *Proceedings of the 2nd Conference on File and Storage Technologies (FAST’03)*. 2003, pages 203–216.

[64] Amazon S3 Availability Event. `http://status.aws.amazon.com/s3-20080720.html`. 2008.

[65] Pascal Felber and Priya Narasimhan. “Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems.” In: *IEEE Transactions on Computers* 53.5 (2004), pages 497–511.

[66] Christof Fetzer. “Perfect Failure Detection in Timed Asynchronous Systems.” In: *IEEE Transactions on Computers* 52.2 (2003), pages 99–112.

[67] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process.” In: *Journal of the ACM* 32.2 (1985), pages 374–382.

[68] Brad Fitzpatrick. “Distributed Caching with memcached.” In: *Linux Journal* 2004.124 (2004), pages 72–74.

[69] Roy Friedman and Robbert 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.

[70] Daniel F. García and Javier García. “TPC-W E-Commerce Benchmark Evaluation.” In: *Computer* 36.2 (2003), pages 42–48.

[71] Miguel Garcia, Alysson Neves Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. “Analysis of Operating System Diversity for Intrusion Tolerance.” In: *Software: Practice and Experience* (2013).

[72] 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.

[73] Felix C. Gärtner. *Byzantine Failures and Security: Arbitrary is not (always) Random*. Technical report IC/2003/20. Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, 2003.

[74] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. “On Designing Dependable Services with Diverse Off-the-Shelf SQL Servers.” In: *Architecting Dependable Systems II*. Springer, 2004, pages 191–214.

[75] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System.” In: *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP’03)*. 2003, pages 29–43.

[76] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. “Scalable, Distributed Data Structures for Internet Service Construction.” In: *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI’00)*. 2000, pages 319–332.

[77] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. “The Next 700 BFT Protocols.” In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys’10)*. 2010, pages 363–376.

[78] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. “Low-Overhead Byzantine Fault-Tolerant Storage.” In: *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP’07)*. 2007, pages 73–86.

[79] James Hendricks, Shafeeq Sinnamohideen, Gregory R. Ganger, and Michael K. Reiter. “Zzyzx: Scalable Fault Tolerance through Byzantine Locking.” In: *Proceedings of the 40th International Conference on Dependable Systems and Networks (DSN’10)*. 2010, pages 363–372.

[80] Chi Ho. “Reducing Costs of Byzantine Fault Tolerant Distributed Applications.” PhD thesis. Cornell University, 2011.

[81] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. “A Case for High Performance Computing with Virtual Machines.” In: *Proceedings of the 20th International Conference on Supercomputing (ICS’06)*. 2006, pages 125–134.

[82] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. “Software Rejuvenation: Analysis, Module and Applications.” In: *Proceedings of 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*. 1995, pages 381–390.

[83] Yih Huang and Arun Sood. “Self-Cleansing Systems for Intrusion Containment.” In: *Proceedings of the Workshop on Self-Healing, Adaptive, and Self-Managed Systems (SHAMAN’02)*. 2002.

[84] 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 (ATC’10)*. 2010, pages 145–158.

[85] JGroups. `http://www.jgroups.org/`.

[86] 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.

[87] Rüdiger Kapitza, Matthias Schunter, Christian Cachin, Klaus Stengel, and Tobias Distler. “Storyboard: Optimistic Deterministic Multithreading.” In: *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HotDep’10)*. 2010, pages 1–6.

[88] 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.

[89] Manos Kapritsos and Flavio P. Junqueira. “Scalable Agreement: Toward Ordering as a Service.” In: *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HotDep’10)*. 2010, pages 7–12.

[90] Manos Kapritsos, Yang Wang, Vivien Quéma, 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.

[91] Jeffrey Katcher. *Postmark: A New File System Benchmark*. Technical report 3022. Network Appliance Inc., 1997.

[92] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[93] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. “kvm: the Linux Virtual Machine Monitor.” In: *Proceedings of the Ottawa Linux Symposium (OLS 07)*. 2007, pages 225–230.

[94] John C. Knight and Nancy G. Leveson. “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming.” In: *IEEE Transactions on Software Engineering* 12.1 (1986), pages 96–109.

[95] Ruppert R. Koch, Sanjay Hortikar, Louise E. Moser, and Peter Michael Melliar-Smith. “Transparent TCP Connection Failover.” In: *Proceedings of the 33rd International Conference on Dependable Systems and Networks (DSN’03)*. 2003, pages 383–392.

[96] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. “Zyzzyva: Speculative Byzantine Fault Tolerance.” In: *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP’07)*. 2007, pages 45–58.

[97] 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.

[98] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. 1997.

[99] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. “Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring.” In: *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS’13)*. 2013, pages 1–17.

[100] Klaus Kursawe. “Optimistic Byzantine Agreement.” In: *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS’02)*. 2002, pages 262–267.

[101] Petr Kuznetsov and Rodrigo Rodrigues. “BFTW3: Why? When? Where? Workshop on the Theory and Practice of Byzantine Fault Tolerance.” In: *SIGACT News* 40.4 (2009), pages 82–86.

[102] Leslie Lamport. “Proving the Correctness of Multiprocess Programs.” In: *IEEE Transactions on Software Engineering* 3.2 (1977), pages 125–143.

[103] Leslie Lamport. “The Part-time Parliament.” In: *ACM Transactions on Computer Systems* 16.2 (1998), pages 133–169.

[104] Leslie Lamport and Mike Massa. “Cheap Paxos.” In: *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN'04)*. 2004, pages 307–314.

[105] 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.

[106] Butler W. Lampson. “Hints for Computer System Design.” In: *Proceedings of the 9th Symposium on Operating Systems Principles (SOSP'83)*. 1983, pages 33–48.

[107] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. “Measurement and Analysis of Large-scale Network File System Workloads.” In: *Proceedings of the 2008 USENIX Annual Technical Conference (ATC'08)*. 2008, pages 213–226.

[108] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. “TrInc: Small Trusted Hardware for Large Distributed Systems.” In: *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI'09)*. 2009, pages 1–14.

[109] Jinyuan Li and David Maziéres. “Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems.” In: *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI'07)*. 2007, 131–144.

[110] Shu Lin and Daniel J. Costello. *Error Control Coding*. Prentice Hall, 2004.

[111] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. “Replication in the Harp File System.” In: *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP'91)*. 1991, pages 226–238.

[112] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. “Boxwood: Abstractions as the Foundation for Storage Infrastructure.” In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. 2004, pages 105–120.

[113] Dahlia Malkhi and Michael Reiter. “Byzantine Quorum Systems.” In: *Distributed Computing* 11.4 (1998), pages 203–213.

[114] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. “Minimal Byzantine Storage.” In: *Proceedings of the 16th International Conference on Distributed Computing (DISC'02)*. 2002, pages 311–325.

[115] Manish Marwah, Shivakant Mishra, and Christof Fetzer. “TCP Server Fault Tolerance Using Connection Migration to a Backup Server.” In: *Proceedings of the 33rd International Conference on Dependable Systems and Networks (DSN'03)*. 2003, pages 373–382.

[116] Arun C. Murthy, Chris Douglas, Mahadev Konar, Owen O'Malley, Sanjay Radia, Sharad Agarwal, and Vinod Kumar Vavilapalli. *Architecture of Next Generation Apache Hadoop MapReduce Framework*. Technical report. 2011.

[117] Athicha Muthitacharoen, Benjie Chen, and David Mazières. “A Low-bandwidth Network File System.” In: *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP’01)*. 2001, pages 174–187.

[118] Balachandran Natarajan, Aniruddha Gokhale, Shalini Yajnik, and Douglas C. Schmidt. “DOORS: Towards High-performance Fault Tolerant CORBA.” In: *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA’00)*. 2000, pages 39–48.

[119] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. “The Eucalyptus Open-source Cloud-computing System.” In: *Proceedings of the 9th International Symposium on Cluster Computing and the Grid (CCGrid’09)*. 2009, pages 124–131.

[120] ObjectWeb Consortium. *RUBiS: Rice University Bidding System*. `http://rubis.ow2.org/`.

[121] Mike Osier. *Netflix Blog – Shipping Delay Recap*. `http://blog.netflix.com/2008/08/shipping-delay-recap.html`. 2008.

[122] Rafail Ostrovsky and Moti Yung. “How to Withstand Mobile Virus Attacks.” In: *Proceedings of the 10th Symposium on Principles of Distributed Computing (PODC’91)*. 1991, pages 51–59.

[123] Google App Engine Outage. `http://groups.google.com/group/google-appengine/browse_thread/thread/f7ce559b3b8b303b?pli=1`. 2008.

[124] Jehan-Francois Pâris. “Voting with Witnesses: A Consistency Scheme for Replicated Files.” In: *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS’06)*. 1986, pages 606–612.

[125] Marshall Pease, Robert Shostak, and Leslie Lamport. “Reaching Agreement in the Presence of Faults.” In: *Journal of the ACM* 27.2 (1980), pages 228–234.

[126] Jesse Pool, Ian Sin Kwok Wong, and David Lie. “Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical.” In: *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HOTOS’07)*. 2007, pages 1–6.

[127] Jon Postel. *Transmission Control Protocol*. RFC 793. 1981.

[128] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “IRON File Systems.” In: *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP’05)*. 2005, pages 206–220.

[129] Sean Quinlan and Sean Dorward. “Venti: A New Approach to Archival Storage.” In: *Proceedings of the 1st Conference on File and Storage Technologies (FAST’02)*. 2002, pages 89–101.

[130] Hans P. Reiser and Rüdiger Kapitza. “Hypervisor-based Efficient Proactive Recovery.” In: *Proceedings of the 26th Symposium on Reliable Distributed Systems (SRDS’07)*. 2007, pages 83–92.

[131] Robbert van Renesse, Chi Ho, and Nicolas Schiper. “Byzantine Chain Replication.” In: *Principles of Distributed Systems*. Springer, 2012, pages 345–359.

[132] Robbert van Renesse and Fred B. Schneider. “Chain Replication for Supporting High Throughput and Availability.” In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI’04)*. 2004, pages 91–104.

[133] Sean C. Rhea, Patrick R. Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y. Zhao, and John Kubiatowicz. “Pond: The OceanStore Prototype.” In: *Proceedings of the 2nd Conference on File and Storage Technologies (FAST’03)*. 2003, pages 1–14.

[134] Jorge Salas, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Bettina Kemme. “Lightweight Reflection for Middleware-based Database Replication.” In: *Proceedings of the 25th Symposium on Reliable Distributed Systems (SRDS’06)*. 2006, pages 377–390.

[135] Raúl Salinas-Monteagudo and Francesc D. Muñoz-Escoí. “Almost Triggerless Writeset Extraction in Multiversioned Databases.” In: *Proceedings of the 2nd International Conference on Dependability (DEPEND’09)*. 2009, pages 136–142.

[136] David Sames, Brian Matt, Brian Niebuhr, Gregg Tally, Brent Whitmore, and David E. Bakken. “Developing a Heterogeneous Intrusion Tolerant CORBA System.” In: *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN’02)*. 2002, pages 239–248.

[137] Nuno Santos and André Schiper. “Tuning Paxos for High-throughput with Batching and Pipelining.” In: *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN’12)*. 2012, pages 153–167.

[138] Fred B. Schneider. “Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial.” In: *ACM Computer Survey* 22.4 (1990), pages 299–319.

[139] Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman. “Prophecy: Using History for High-Throughput Fault Tolerance.” In: *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI’10)*. 2010, pages 345–360.

[140] Gurudatt Shenoy, Suresh K. Satapati, and Riccardo Bettati. “HYDRANET-FT: Network Support for Dependable Services.” In: *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS’00)*. 2000, pages 699–706.

[141] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. “Zeno: Eventually Consistent Byzantine-Fault Tolerance.” In: *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI’09)*. 2009, pages 169–184.

[142] Matthew Smith, Christian Schridde, and Bernd Freisleben. “Securing Stateful Grid Servers through Virtual Server Rotation.” In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC’08)*. 2008, pages 11–22.

[143] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery." In: *IEEE Transactions on Parallel and Distributed Systems* 21.4 (2010), pages 452–465.

[144] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "Resilient Intrusion Tolerance through Proactive and Reactive Recovery." In: *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC'07)*. 2007, pages 373–380.

[145] Sun Microsystems. *NFS: Network File System Protocol Specification*. Internet RFC 1094. 1989.

[146] Symantec. *Veritas Cluster Server from Symantec*. 2012.

[147] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Reiner Schmid. "Eliminating Single Points of Failure in Software-Based Redundancy." In: *Proceedings of the 9th European Dependable Computing Conference (EDCC'12)*. 2012, pages 49–60.

[148] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. "Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling." In: *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*. 2007, pages 59–72.

[149] Paulo Esteves Veríssimo, Nuno Ferreira Neves, Christian Cachin, Jonathan A. Poritz, David Powell, Yves Deswarte, Robert Stroud, and Ian Welch. "Intrusion-Tolerant Middleware: The Road to Automatic Security." In: *IEEE Security & Privacy* 4.4 (2006), pages 54–62.

[150] Paulo Esteves Veríssimo, Nuno Ferreira Neves, and Miguel Pupo Correia. "Intrusion-tolerant Architectures: Concepts and Design." In: *Architecting Dependable Systems*. Springer, 2003, pages 3–36.

[151] Giuliana Santos Veronese. "Intrusion Tolerance in Large Scale Networks." PhD thesis. University of Lisbon, 2010.

[152] 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.

[153] 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 Symposium on Reliable Distributed Systems (SRDS'09)*. 2009, pages 135–144.

[154] 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 (2011), pages 16–30.

[155] VMware. `http://www.vmware.com/`.

[156] VMware. *VMware High Availability*. 2009.

[157] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. "XenLoop: A Transparent High Performance Inter-VM Network Loopback." In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC'08)*. 2008, pages 109–118.

[158] 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.

[159] 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.

[160] Xen.org. *Xen Networking*. `http://wiki.xen.org/wiki/Xen_Networking`.

[161] 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.

[162] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. "COCA: A Secure Distributed Online Certification Authority." In: *ACM Transactions on Computer Systems* 20.4 (2002), pages 329–368.

[163] Piotr Zieliński. *Paxos at War*. Technical report UCAM-CL-TR-593. University of Campbridge, 2004.