Resource-Aware System Software for Replicated Services

Tobias Distler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Abstract

Replicated services such as distributed file systems, key-values stores, or blockchains are essential parts of today's computing infrastructures and therefore typically designed to withstand a wide spectrum of fault scenarios including hardware crashes and software failures. To handle the complexity associated with fault-tolerant replication, the systems providing these services rely on replication system software, that is special-purpose system software that implements a replication protocol and, among many other things, is responsible for ensuring consistency across replicas. Representing the foundation of an entire dependable system, it is crucial for replication system software to meet strict reliability and efficiency requirements in order to not become a vulnerability or performance bottleneck.

This habilitation treatise explores different approaches to improve the efficiency of reliable replication system software by making it resource aware. Specifically, this means that the system software is supplied with detailed knowledge about all available resources and enabled to use such information to adapt its own architecture and configuration. In this context, a major goal of the underlying research was to develop techniques that are generic and neither depend on a specific fault model nor a certain replication protocol.

In particular, this habilitation treatise makes research contributions in three areas: (1) It presents a parallelization scheme that allows replication system software to use processing resources more effectively by adjusting its degree of parallelism to the number of cores available on a replica server. (2) It describes a generic approach for replication system software to save energy during periods of low and medium utilization by autonomously controlling its configuration at multiple system layers in an adaptive and coordinated manner. (3) It introduces a set of techniques that are tailored to specific use cases and improve the performance and resource efficiency of replication system software for heterogeneous environments, services with large states, and geo-replicated settings.

1 Introduction

Online services represent essential building blocks of today's computing infrastructures, either as part of applications directly facing the user (e.g., e-mail, online banking) or integrated with secondary platforms at lower data-center tiers (e.g., distributed file systems, blockchains). Being such crucial components, it is important for these services to remain correct and available even when they are confronted with a wide spectrum of fault scenarios including communication failures, hardware crashes, and arbitrary faults caused by software bugs. In addition, highly critical services must be prepared and able to deal with security-related issues and offer resilience in the face of attacks and possible intrusions.

1.1 System Support for Replication-based Fault Tolerance

As illustrated in Figure 1, a common approach to make network-based services dependable is to replicate their server side across multiple machines. That is, to prevent a single point of failure multiple servers each host their own instance ("replica") of the service application. Clients can remotely invoke an operation at the service by sending a corresponding request to the server side and waiting for a response carrying the operation's result. To ensure consistency in the absence as well as the presence of faults, all communication between clients and replicas, and also the interaction within the group of replicas, is handled by a fault-tolerant replication protocol. Among other things, the replication protocol ensures that the service states of all correct replicas remain consistent, for example by guaranteeing that replicas perform all state modifications in the same order [147]. In addition, a replication protocol typically provides means to synchronize replica states via checkpoints, thereby enabling faulty servers to catch up after they have been repaired.

Fault Models. The number of replicas required at the server side usually depends on how many concurrent replica failures a system is supposed to tolerate at any given time. Furthermore, the size of the replica group is a result of the degree of fault tolerance to be provided. In general, two fault models are of particular interest in the context of this treatise:

- **Crash Fault Tolerance (CFT):** Systems and protocols designed for this fault model assume that clients and replicas only fail by crashing and otherwise always behave according to specification. In particular, if a client or replica sends a message to others in which it confirms the completion of a certain protocol task (e.g., the execution of a request), the recipients of the message can be sure that the sender indeed has performed the task. Hence, in crash-tolerant systems clients and repli-

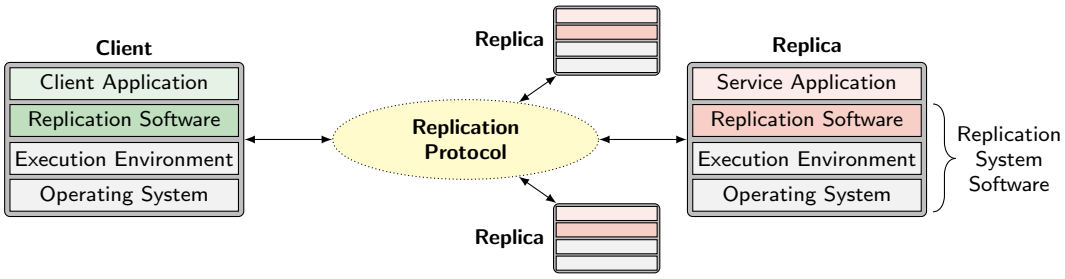


Figure 1. Basic system architecture for replication-based fault tolerance

cas commonly trust each other and the messages they exchange. In production, this fault model is often used for replicated services at lower layers (e.g., key-value stores [48, 128] or coordination services [35, 92]) which do not directly interact with the outside world.

- **Byzantine Fault Tolerance (BFT):** This fault model assumes that clients and replicas may fail in arbitrary ways [107]. Specifically, this includes the possibility that a client or replica continues to participate in the replication protocol after becoming faulty, potentially distributing messages that do not properly reflect its actual state. Since no assumptions are made regarding the origins of a fault, the fault model also addresses scenarios in which a malicious adversary has successfully managed to take over a client or replica and now tries to use it to deliberately cause disarray in the system; in the worst case, corrupted clients or replicas might even collude with each other in order to reach their goal. As a consequence, clients and replicas in Byzantine fault-tolerant systems do not trust each other, which makes this fault model a good fit for user-facing replicated services, systems whose replicas are hosted by different independent organizations, or services with strong fault-tolerance requirements (e.g., blockchains [86, 149, 158], firewalls [31, 83], or SCADA systems [17, 127]).

Providing resilience against a wider spectrum of failures, BFT systems in general are more complex than CFT systems. For example, to tolerate up to f simultaneous replica faults, without further assumptions a BFT system must comprise at least $3f + 1$ replicas [134], whereas in CFT systems $2f + 1$ replicas are sufficient for the same fault-tolerance threshold. Apart from the number of replicas involved, BFT systems also have a higher complexity with regard to replication protocols [37, 172], which typically require additional phases of message exchange between replicas compared with CFT protocols [98, 129].

Replication System Software. Independent of the specific fault model, a common approach to handle the complexity associated with building replicated systems is to separate the service application from the replication protocol and implement all application-independent functionality inside the *replication system software*. In this treatise, replication system software is used as an umbrella term referring to all software at either clients or replicas that is not part of the service to be replicated. Specifically, the term combines (1) the replication software implementing the CFT/BFT replication protocol (e.g., BFT-SMaRt [30]), (2) if available, the execution environment in which the replication software runs (e.g., the Java virtual machine), as well as (3) a server's underlying operating system.

To fulfill its diverse set of responsibilities, the replication system software itself needs to manage and utilize a significant amount of resources. Primarily, this includes processor cycles for the authentication of messages (e.g., the computation and verification of signatures), memory for the management of data structures required by the replication protocol, disk space for the storage of checkpoints, and network connections for the interaction within the replica group as well as between clients and replicas. Ideally, the system software uses the available resources as effectively and efficiently as possible in order to not become a performance bottleneck itself and to leave most resources to the application.

1.2 Contributions

A main goal of my research in recent years has been (and still is) to improve the performance and efficiency of replicated systems by increasing the resource awareness of replication system software. Among other things, resource awareness in this context can for example mean to enable replication system software to flexibly adjust the degree of parallelism in the implementation to the number of cores available on a server. Furthermore, it can also mean to provide knowledge on the performance and energy-consumption characteristics of different system-parameter configurations in order to allow the system software to autonomously select a suitable configuration for the current conditions. Overall, this treatise reports on contributions and research results in the following three areas:

Effective and Efficient Use of Multi-Cores. In an effort to improve the scalability of replication system software, the treatise presents *consensus-oriented parallelization*, a technique that allows replication system software to sustain high

throughput levels by exploiting the processing power available on multi-core server hardware. The approach has been integrated with a variety of replication protocols and laid the foundation for the first BFT system that is able to handle more than one million client requests per second [23].

Energy-Aware Adaptation to Varying Workloads. Since replicated services in practice are not continuously operated at the highest performance level, the treatise proposes *energy-aware reconfigurations* as a means for replicas to save energy (without impeding quality of service) during periods of reduced workloads. The reconfigurations are performed at runtime and their effectiveness stems from the fact that the replication system software coordinates the adaptation across multiple system layers, including the hardware.

Techniques Tailored to Specific Use Cases. As a result of the wide range of use-case scenarios for replicated services, replication system software is not only confronted with general goals such as high performance and low energy consumption (see above), but also needs to address application-specific resource demands. To this end, the treatise presents (1) a solution to deploy replicated systems in *environments with heterogeneous servers*, (2) an efficient mechanism to create *consistent checkpoints of replicas with large states*, and (3) a *cloud-based replication system software* for geo-replicated services that exploits the special properties of today's public-cloud infrastructures to minimize end-to-end latency.

1.3 Papers of this Treatise

This document is a cumulative habilitation treatise and includes the following 10 peer-reviewed publications, which represent the main contributions of my research on resource-aware system software for replicated and distributed services. For reprints of these papers please refer to Appendix B. My full publication list is available in Appendix A.

System Software for Replicated Services

CSUR '21	*Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective* Tobias Distler [53]

Effective and Efficient Use of Multi-Cores

Middleware '15	*Consensus-Oriented Parallelization: How to Earn Your First Million* Johannes Behl, Tobias Distler, and Rüdiger Kapitza [23]
DSN '17	*Agora: A Dependable High-Performance Coordination Service for Multi-Cores* Rainer Schiekofer, Johannes Behl, and Tobias Distler [146]
PaPoC '19	*In Search of a Scalable Raft-based Replication Architecture* Christian Deyerl and Tobias Distler [49]

Energy-Aware Adaptation to Varying Workloads

ARM '15	*Towards Energy-Proportional State-Machine Replication* Christopher Eibel and Tobias Distler[62]
IC2E '18	*Empya: Saving Energy in the Face of Varying Workloads* Christopher Eibel, Thao-Nguyen Do, Robert Meißner, and Tobias Distler [64]
DAIS '18	*Strome: Energy-Aware Data-Stream Processing* Christopher Eibel, Christian Gulden, Wolfgang Schröder-Preikschat, and Tobias Distler [65]

Techniques Tailored to Specific Use Cases

Computing '19	*Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers* Michael Eischer and Tobias Distler [72]
SRDS '19	*Deterministic Fuzzy Checkpoints* Michael Eischer, Markus Büttner, and Tobias Distler [68]
Middleware '20	*Resilient Cloud-based Replication with Low Latency* Michael Eischer and Tobias Distler [73]

A note on the order of authors: The first author (in most cases a doctoral or master student) did most of the actual implementation and experimental work, in close collaboration with me as the responsible scientific project lead. As such, I am listed as last author on most of the publications above. For the Middleware '15 paper, Rüdiger Kapitza (TU Braunschweig) and I both acted as scientific project lead at our respective institutions.

1.4 Structure of this Treatise

The remainder of this document is structured as follows:

Chapter 2 provides background on replication system software thereby highlighting typical responsibilities, discussing implementation challenges, and describing common building blocks. Furthermore, the chapter also reviews existing works from the general field of replicated services that had an impact on the dependability and performance of replication system software.

Chapter 3 presents the concept of consensus-oriented parallelization as a means to improve the scalability and parallelizability of replication system software. Specifically, the approach enables a replica to more effectively utilize multiple cores and to minimize synchronization overhead.

Chapter 4 proposes an approach that enables the replication system software to reduce the energy consumption of a replicated service during periods of low and medium workloads. The technique relies on the dynamic reconfiguration of resource parameters (e.g., the number of active cores on a replica server) at multiple system levels, including the hardware.

Chapter 5 examines several ways of how replication system software can be tailored to use-case scenarios with specific characteristics and requirements. In particular, this includes replicated services running on heterogeneous hardware, applications with large states, as well as cloud-based geo-distributed systems whose replicas are located at different geographic sites.

Chapter 6 summarizes the contributions of this treatise and elaborates on how the different techniques can be combined with each other. Moreover, the chapter discusses additional application scenarios and outlines open problems that may serve as a starting point for future research on replication system software.

Appendix A provides a complete list of my (peer-reviewed and non-refereed) research publications.

Appendix B contains reprints of the 10 peer-reviewed papers listed in Section 1.3, which together represent the main part of this cumulative treatise.

2 System Software for Replicated Services

Since replication system software represents an essential building block of fault-tolerant replicated services, it is also the main object of research in the context of this treatise. This chapter provides background on this type of special-purpose system software by discussing common architectural components as well as the tasks replication system software is usually assigned with. In addition, based on an analysis of previous works, the chapter formulates research challenges in improving different aspects of replication system software.

2.1 Replication System Software

The purpose of replication system software is to provide distributed applications with an interface they can use to achieve fault tolerance through replication. Figure 2 shows an example of such an interface that is common in replicated systems relying on the principle of state-machine replication [147]. Essentially, the replication system software offers clients the possibility to invoke operations at the server side by performing remote procedure calls [126]. In contrast to remote procedure calls in client–server architectures, however, the replication system software ensures that the operation is executed at all correct replicas, not just a single server. Furthermore, the system software is responsible for eventually providing the client with a result to its operation even if some of the replicas have failed. To fulfill this requirement, the system software executes a replication protocol that takes

Client Interface

```
/* Request processing */
Result invoke(Operation op);
```

Replica Interface

```
/* Request processing */
Result invoke(Operation op);

/* State transfer */
State getState();
void setState(State st);
```

Figure 2. Example interfaces between application and replication system software

care of keeping the states of correct replicas consistent. Occasionally, this task makes it necessary to exchange state information between replicas, which is why the replication system software's interface typically also includes means for state transfer (see Figure 2). Among other things, these methods enable a new replica to join a running system.

Although the interface offered to the application in many cases is comparably small, implementations of this interface are usually rather complex. This is a direct consequence of (1) the strong fault-tolerance guarantees the replication system software provides and (2) the fact that the system software itself is a layer that is distributed across multiple servers. As shown in Figure 3, to handle the associated complexity replication system software is commonly implemented as a collection of several modules with dedicated responsibilities. The following paragraphs discuss the four most important modules in detail.

Client Library. This system-software component is located at the client and handles all interaction with the replicated server side. In particular, this task includes the creation and transmission of requests as well as the collection of responses arriving from replicas. Since in BFT systems clients do not trust individual replicas, client libraries there typically also perform a result verification before delivering the result to their local client application. The most common approach to verify a result is to compare the responses obtained from multiple replicas, since in the presence of at most f faults $f + 1$ matching responses from different replicas represent a proof of correctness for the result [37]. In CFT systems, on the other hand, if a replica provides a result to a request, the result is assumed to be inherently correct (see Section 1.1). Consequently, client libraries in these systems are usually able to accept and deliver the first result they receive. Apart from fulfilling basic duties such as communication and result verification, client libraries may also be assigned with additional tasks [53, 135]. Some systems, for example, rely

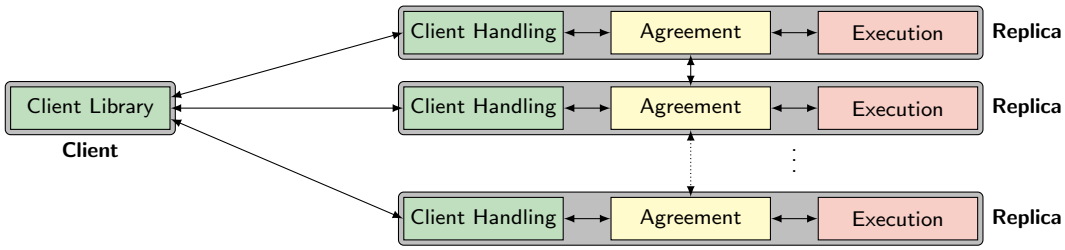


Figure 3. Overview of replication system software modules

on clients to detect and/or resolve inconsistencies between replicas [15, 99] or to supply replicas with performance statistics that are afterwards analyzed and used to improve efficiency [70].

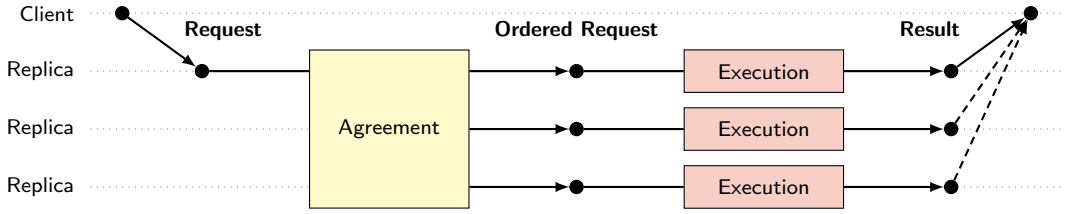
Client Handling. This module is the counter part of the client library at the server side. Besides managing network connections, in many systems the component is also responsible for validating whether a client is indeed permitted to access the service. If this check is successful the client's request will be further processed, otherwise a replica simply ignores the message. In most replicated systems, there is no need for the client-handling modules of different replicas to directly interact with each other, although there are exceptions. Some systems, for example, ask replicas to immediately distribute client requests within the replica group after they receive them [9, 44], thereby enabling more efficient agreement protocols by decoupling request distribution from the subsequent consensus process.

Agreement. This component implements the core of the replication protocol: the agreement process guaranteeing that the states of correct replicas in the system remain consistent despite the presence of a number of potentially faulty replicas. To fulfill this requirement, replicas do not process incoming client requests directly once they receive them over the network, because doing so might cause the states of two correct replicas to diverge. For example, if there are two state-modifying requests (one that newly creates a file at the server side and another one that writes data to this file) and these two requests arrive at different replicas in a different order, then the replicas involved would not end up in the same state. Specifically, those replicas that execute the create request first would eventually have a copy of the file that includes the data written by the second request, whereas the replicas that try to process the write request first would remain with an empty file.

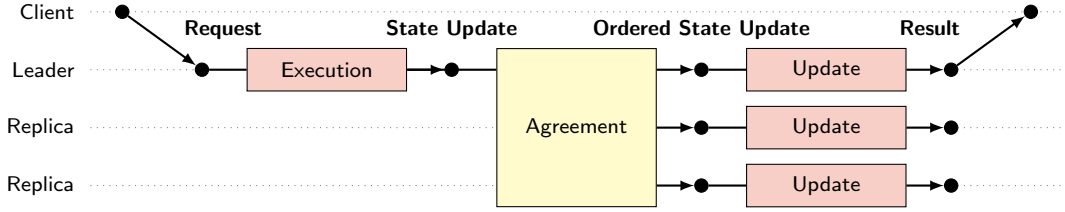
To prevent scenarios such as the one outlined above, replicas first reach consensus on how they are going to change their state before actually making these state changes persistent. More precisely, replicas run an agreement protocol whose main purpose is to reliably assign unique sequence numbers to state modifications. As further discussed in Section 2.2, the specific representation of state modifications in this totally ordered sequence may differ between replication protocols. While some replication protocols agree on client requests before processing them [37, 98], other protocols execute requests first and afterwards reach consensus on the order in which to apply the corresponding state updates [94, 149]. Either way, correct replicas in the system remain consistent by adhering to the same agreed sequence and thereby advancing their states in a coordinated manner.

From a researcher and developer perspective, the agreement stage is one of the most challenging parts of replication system software. Establishing a total order on state modifications, for example, requires multiple rounds of message exchanges among replicas, often times with quadratic message complexity [37, 98]. As a result, overall system throughput and latency in many use-case scenarios are significantly impacted by how efficient the replication system software is able to perform consensus. Unfortunately, the sequential nature of the agreement process makes it inherently difficult to parallelize this system software component, causing this problem to be a major research challenge (see Section 2.3).

Execution. This component of the replication system software is responsible for the interaction with the local service-application instance and consequently provides an interface such as the one exemplified in Figure 2. As input, the execution stage uses the totally ordered sequence of requests/updates generated by the agreement stage, and it ensures that the application processes them in a consistent manner. If the application produces a result, the execution stage relays the result back to the client. Apart from request/update processing, a second major task performed by the execution stage is to create periodic checkpoints of the application state, which mainly serve two purposes: (1) Checkpoints lay the foundation for a state-transfer mechanism that allows new replicas to catch up after having joined the system at runtime. (2) Since a checkpoint resembles the effects a request/update had on the state, the agreement stage can use the existence of new checkpoint as opportunity to garbage collect consensus information on old messages [37]. As further discussed in Section 2.5, an efficient checkpointing mechanism provided by the replication system software is especially crucial for service applications with large states.



(a) Active replication



(b) Passive replication

Figure 4. Comparison of replication-protocol architectures

2.2 Variety in Replication Protocols

Decades of research have led to a myriad of replication protocols with different characteristics. Consequently, it is particularly challenging to develop new techniques that both improve replication system software and are applicable to a wide spectrum of protocols. To illustrate the variety in protocols, this section compares two common protocol architectures.

Active Replication. As shown in Figure 4a, protocols relying on active replication [147] first perform an agreement on the incoming client requests and then require each replica to execute each request. For replicas to remain consist, this form of replication demands deterministic service implementations, meaning that for the same sequence of requests each correct replica must produce the same results and reach the same states. Active replication serves as basis for the vast majority of BFT replication protocols because the fact that multiple replicas execute all requests independently of each other makes it possible for clients to verify the corresponding results by comparing the responses of different replicas.

Passive Replication. Using passive replication [34], an elected leader replica processes all incoming requests and records the associated state modifications in the form of an update message (see Figure 4b). In a next step, all replicas then reach consensus on the state update and finally apply the update to their local states. In contrast to active replication, passive replication does not necessarily require deterministic applications as each request is only executed by a single replica. On the other hand, this property also means that results cannot be verified by comparing responses, which is why passive replication can be primarily found in CFT protocols where clients and replicas are assumed to trust each other.

2.3 Research Challenge 1: Parallelization

The vast majority of protocol specifications deliberately apply sequential execution as a means to keep replicas consistent. While this approach simplifies protocol design, it also makes it inherently difficult to build replication system software for multi-core servers.

Parallelized Execution. A number of works approached the problem by proposing mechanisms targeting the execution stage of a replica, which can be parallelized in several different ways. CBASE [75, 100], for example, assumes that requests carry hints about the state parts they access during execution and uses this knowledge to identify requests that conflict with each other when being processed concurrently. Based on this information, CBASE then only enforces sequential execution for conflicting requests while processing non-conflicting requests in parallel. OptSCORE [89] and Storyboard [96] exploit the insight that to ensure consistency among correct replicas it is usually not necessary to sequentialize the entire execution of requests as long as all critical sections are processed in the same order on all replicas. Assuming that such sections are protected by locks, these approaches rely on custom schedulers guaranteeing that threads acquire locks in a consistent manner. Also focusing on synchronization primitives, Rex [87] traces nondeterministic decisions made during parallel execution at a leader replica and afterwards enforces the recorded order of decisions when processing the same requests at other

replicas. In contrast, in Eve [97] all replicas independently execute requests in parallel; if replicas diverge, they roll back their states and for the retry resort to sequential execution.

State Partitioning. An alternative parallelization approach that goes beyond the execution stage is to partition the state of the service application and handle each partition separately. As a consequence, the consensus process for requests accessing different partitions can be distributed across multiple replica groups [2, 119, 140] or parallel agreement instances running on the same set of servers [111]. Since for many applications a clean division into disjoint partitions is not feasible (e.g., due to the state being organized as a tree with dependencies between parent and child nodes), a common challenge for such protocols is to ensure consistency for requests that span two or more partitions [111, 119].

Analysis. Existing approaches to introduce parallelism into replication protocols either focus on the execution stage of a system or they impose additional restrictions on the application such as the requirement of a partitioned state or the need to identify the target of an operation from its request message. As a consequence, the proposed techniques are only applicable to replicated services that meet these criteria and even then integrating the service application with the replication system software usually involves further efforts. Ideally, the replication system software of a replica should be able to exploit parallelism in all of its components (including the agreement stage) and without making additional assumptions about the application; note that this does not rule out the possibility of extending the parallelism to the application for services that already are parallelized and/or naturally support state partitioning. Chapter 3 presents a generic solution that has the mentioned properties and can be combined with a variety of existing replication protocols.

2.4 Research Challenge 2: Resource Efficiency

Replicating a service for fault tolerance in general is associated with a significant increase in consumption of network and processing resources compared with the unreplicated variant of the same service. As a consequence, a notable amount of works addressed the challenge of improving the resource efficiency of both CFT as well as BFT replication protocols.

Phase-Specific Approaches. One direction of research pursued to solve the problem is to distinguish between (1) phases of benign conditions in which the network timely delivers messages and all replicas behave correctly and (2) phases of rough conditions in which the network is unreliable and/or some of the replicas are faulty. As a key benefit, in a replicated system making progress under benign conditions requires less resources than are necessary to actually tolerate faults. This general insight can be exploited in various forms: In some replication protocols, for example, only a subset of replicas normally comprise a full copy of the application state, while the other replicas maintain enough information to be able to obtain the latest state and assist if a full replica fails [58, 106, 115, 133, 169]. Other systems actively involve all replicas during normal-case operation but execute each request on only a subgroup of replicas; the subgroup is sufficiently large so that the effects a request had on the application state do not get lost in case of faults [57, 102]. Apart from these approaches, it is also possible to increase resource efficiency by enabling a system to dynamically switch its replication protocol depending on the current fault conditions [15, 55, 95], for example selecting a high-performance protocol for benign conditions while changing to a more robust protocol once faults are suspected or detected.

Relaxed Fault Models. A second research direction, which in particular aims at circumventing the high resource demand of full-fledged BFT systems, is to apply hybrid fault models that represent intermediate points on the spectrum between the traditional CFT and BFT fault models (see Section 1.1). On the one hand, this concept for example may be used to minimize the size of a replica group by assuming that a certain number of faults will cause a replica to crash but not actually manifest in arbitrary behavior [44, 116, 137]. One the other hand, there are approaches to reduce replica-group size by considering specific system components to be trusted and to only fail by crashing, while still assuming and tolerating Byzantine faults in the untrusted rest of the replicated system [24, 43, 109, 157].

Analysis. Previous works targeting resource efficiency in replicated systems primarily take effect at the replication-protocol level and are based on the idea of cutting down redundancy that is considered unnecessary, either temporarily (i.e., during periods without replica failures or network problems) or permanently (i.e., for use cases where relaxed fault guarantees are acceptable). Integrated with the replication system software, the techniques above make it possible to (dynamically or statically) adapt the resource footprint of a replicated system to the current fault-tolerance requirements. As a complement to these concepts, Chapter 4 presents an approach that leverages resource savings at multiple system levels (including the operating system and hardware) and instead of fault conditions uses a system's current workload as indicator for when to reduce resource consumption.

2.5 Research Challenge 3: Application-Specific Requirements

Replicated services in practice cover a wide spectrum of application scenarios ranging from small deployments in which all replicas are located within the same building [92] to widely distributed systems with replicas on different continents [152]. To properly support all these services, replication system software needs to address their different charac-

teristics and requirements, not necessarily in a one-fits-all manner (which is typically inefficient or even infeasible) but at least in the form of customized implementations. The next paragraphs discuss three examples of specific use cases that are of relevance for this treatise.

Services in Heterogeneous Environments. Most replication-protocol designs (often implicitly) assume all replicas to run on homogeneous servers with identical performance capabilities (e.g., being equipped with the same number and type of processors). Unfortunately, in practical deployments this assumption does not always hold, which is especially true if services are executed in cloud environments where the performance can significantly vary across virtual machines [130]. Furthermore, the use of diverse replica implementations, which has been proposed as a method to minimize the probability of multiple replicas experiencing correlated faults [16], can further increase heterogeneity. So far, previous works primarily focused on exploring replica diversity as a means to improve robustness [39, 42, 60, 80, 150, 154], however they did not study the performance and resource-usage implications associated with heterogeneous replication system software.

Services with Large States. For applications managing a large amount of data, the need to create periodic checkpoints (see Section 2.1) usually results in significant performance overhead [30]. In particular, this problem is caused by the fact that replication system software implementations typically suspend request execution during checkpoint creation in order to obtain a consistent snapshot of the application state. For BFT replication protocols this is especially problematic since all replicas have to produce snapshots for the same sequence numbers in order for the checkpoints to be verifiable by comparison [37].

Geo-Replicated Services. Distributing a replicated system across several geographic sites can be an effective way to increase the system's resilience against building-wide power outages and natural disasters. On the other hand, such a measure generally comes at the cost of increased response times due the replication protocol now involving multiple steps of wide-area communication. Existing solutions to minimize end-to-end latencies under such conditions include modified agreement quorums [26, 148], protocols that allow each replica to initiate consensus [61, 117, 118, 123, 124, 156], and hierarchical system architectures that deploy an entire replica group at each site [10, 88]. On the downside, all of these approaches still require the execution of complex protocols over long-distance links.

Analysis. The examples illustrate that there is room for improvement when it comes to tailoring replication system software to use cases with special characteristics. Chapter 5 addresses these problems by presenting approaches that enable replication system software to support heterogeneous environments, large states, and efficient geo-replication.

2.6 Main Paper

Replication is a powerful means to provide fault tolerance, unfortunately many research papers on this topic primarily focus on the description of algorithms and protocols, but do not sufficiently discuss how to implement system software for them. This is especially true for BFT protocols, which compared with CFT protocols in general exhibit higher complexity. To mitigate this problem, I wrote a survey article that summarizes the state of the art in BFT state-machine replication and specifically discusses issues that are commonly associated with building actual systems, but are often neglected in research publications. The paper is part of this cumulative treatise and its reprint can be found in Appendix B.

CSUR '21 *Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective*
Tobias Distler [53]

The paper gives an overview of different BFT system architectures and identifies common building blocks for their implementation. In addition, the survey presents a novel abstraction for the analysis of agreement protocols that facilitates the task of deciding whether or not the underlying concepts of different protocols can be combined. As another important part of the survey, the paper discusses existing solutions to essential problems such as efficiently creating checkpoints, recovering replicas from faults, and dynamically changing the composition of the replica group. Despite the paper's focus on Byzantine fault tolerance, many of the topics discussed in the survey also (either fully or in parts) apply to CFT systems.

3 Effective and Efficient Use of Multi-Cores

While the vast majority of servers today consist of processors with multiple cores, most replication protocols (still) are designed as sequential algorithms [37, 94, 104, 129]. To nevertheless utilize at least some of the available computing resources, replication system-software implementations typically introduce parallelism by applying a concept that in the following is referred to as *task-oriented parallelization* [23]. Specifically, they model a replication protocol as a chain of individual processing stages that are connected through message queues and each possess their own threads, similar to a SEDA architecture [168].

One example for this implementation technique being used in practice is the ZooKeeper coordination service [92], a fault-tolerant distributed system that manages configuration data and provides synchronization primitives such as message queues or distributed locks. With this functionality, ZooKeeper represents an essential building block for several large-scale distributed platforms including Hadoop [12], and Storm [13], Spark [173], as it enables their processes to interact with each other. Although a key component for the well-functioning of other services, there have been in-

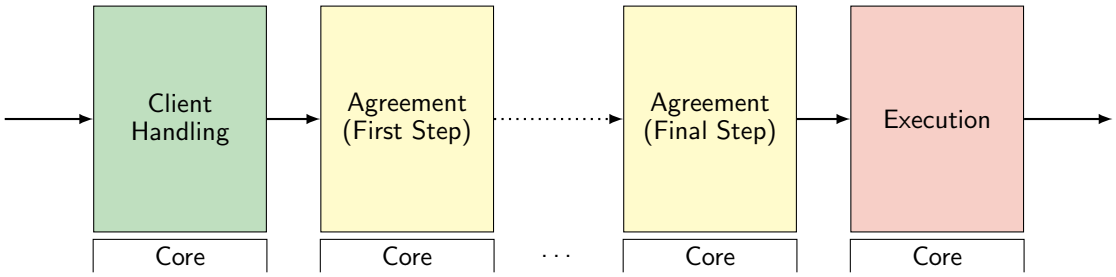


Figure 5. Replica architecture based on a task-oriented parallelization

cidents in which ZooKeeper due to its task-oriented replica architecture became a performance bottleneck of the overall system [41], requiring system designers to find ways to mitigate this problem. Among other things existing solutions to the bottleneck issue include workarounds that keep the coordination service off the critical path [54], migrate parts of its workload to other systems [41], or rely on a composition of multiple ZooKeeper instances [108]. Consequently, they result in more complex designs as well as additional work for programmers and administrators.

This chapter presents *consensus-oriented parallelization*, an alternative solution for scenarios in which the traditional task-oriented parallelization leads to performance bottlenecks. The approach is based on a novel replica architecture that enables the replication system software to use multicore machines both more effectively and more efficiently. The concept behind consensus-oriented parallelization is generic and therefore can be applied to different fault models as well as different replication techniques. Results show that the approach can significantly improve performance using standard server hardware.

3.1 State of the Art

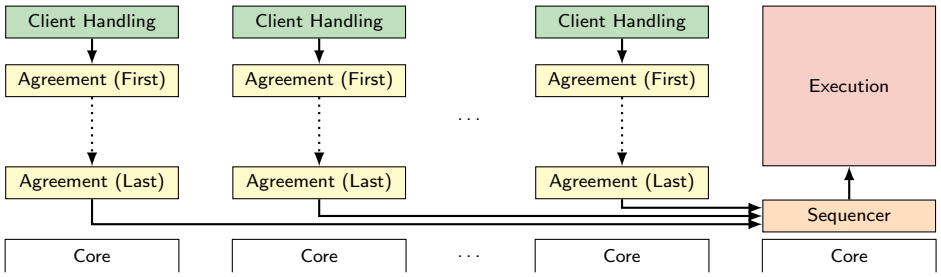
As illustrated in Figure 5, the traditional method for achieving parallelism in replication system software is to divide the sequential replication protocol into multiple modules that each represent a different task in the processing pipeline, for example the reception of a client request, a stage of the agreement process, or the execution of the client request in the application [30, 92, 145]. In a next step, the responsibility for each of these tasks is then delegated to a dedicated thread, thereby enabling the system to use more than one core. This way, it for example becomes possible for a replica to execute a request while in parallel already reaching consensus on a subsequent request. Another advantage of this replica architecture is the fact that it usually is straightforward to deduce the modularization of a specific protocol from its specification. On the other hand, the task-oriented parallelization scheme also has two drawbacks that in practice can significantly impede performance: limited scalability and considerable scheduling and synchronization overhead.

Limited Scalability. With modules representing different protocol tasks, their number is bounded by the number of individual steps the replication protocol consists of, and so is the degree of parallelism that can be achieved for this kind of replica architecture. Although it is possible to further divide some of the tasks into subtasks (e.g., the computation of signatures for different messages), the underlying problem still remains: If a replica has more cores at its disposal than (sub)tasks to perform, the task-oriented parallelization scheme prevents the replica from effectively utilizing all of these cores. For ZooKeeper, for example, multiple studies [145, 146] independently of each other showed that the system's performance only increases up to four cores, which is a small fraction of the number of cores today's (and future) servers are (and will be) typically equipped with.

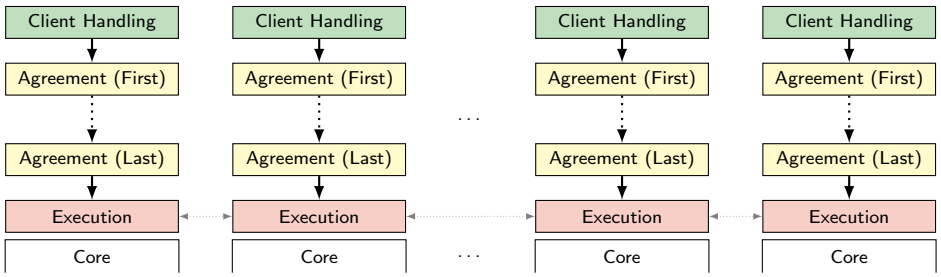
Scheduling and Synchronization Overhead. When implementing a replication protocol as a chain of tasks that are each handled by separate threads, several context switches become necessary to pass a request along the thread chain. In addition, the transport of message data from one core to another in practice usually has a measurable impact on performance [46] since many of the protocol tasks themselves involve only a few operations (e.g., inexpensive checks of message attributes or comparisons of message contents). Besides affecting performance, the scheduling and synchronization overhead often also leads to a replica not being able to fully utilize its available network resources, especially for small requests [49, 146]. Experiments with ZooKeeper for example show that due to task-oriented parallelization replicas are unable to saturate the network for writes of 512 bytes and below [146], which for ZooKeeper are common request sizes [92].

3.2 Approach: Consensus-Oriented Parallelization

In an effort to overcome the limitations of the traditional replica architecture, my research led to the development of a new parallelization scheme for replication system software: consensus-oriented parallelization. As shown in Figure 6, this approach is based on the idea of achieving scalability by relying on multiple largely independent partitions that each are responsible for handling the entire replication process



(a) Parallel agreement + sequential execution (COP-SE)



(b) Parallel agreement + parallel execution (COP-PE)

Figure 6. Replica architectures based on consensus-oriented parallelization

for a dedicated request subset. On all replica servers, each partition runs on a separate core and executes its own instance of the replication protocol to reach consensus on the requests assigned to the partition.

Depending on the properties and requirements of the application, the concept can be applied in two variants: (1) In COP-SE (see Figure 6a), only the agreement on requests is parallelized into partitions, their execution in contrast still occurs sequentially, as in traditional implementations. For this purpose, a new sequencer component is introduced to merge the local request sequences determined by the partitions into a global sequence used by the application. Since the agreement protocol ensures that the local sequences are consistent across replicas and the sequencer's merge algorithm is deterministic, it is guaranteed that all correct replicas in the system produce the same global request sequence. (2) In the COP-PE variant (see Figure 6b), not only agreement is parallelized but also execution. This architecture is suitable for applications that are able to guarantee consistency in the presence of concurrent execution, and compared with COP-SE it has the main advantage of eliminating the merge overhead for creating the global request sequence.

Independent of the specific variant used in an implementation, consensus-oriented parallelization enables replication system software to mitigate the problems associated with task-oriented parallelization. This applies to both issues that have been discussed in Section 3.1, that is, scalability as well as scheduling and synchronization overhead.

Improved Scalability. While task-oriented parallelization distributes work by assigning an individual protocol task to a thread (which the thread then has to perform for all requests), consensus-oriented parallelization partitions the responsibilities for requests (for which a partition then has to perform all protocol tasks). With the number of tasks that are involved in the replication protocol in general being dwarfed by the number of requests a system needs to handle over the course of its lifetime, consensus-oriented parallelization therefore enables a much more fine-grained degree of parallelism. As a consequence, the number of partitions can simply be configured based on the number of available cores.

Minimized Scheduling and Synchronization Overhead. Due to the fact that a partition performs all necessary tasks on a request, consensus-oriented parallelization no longer requires requests to be passed along a chain of threads. Instead, the vast majority of requests never leave their partition until the completion of the agreement process (COP-SE) or even the end of the execution (COP-PE). Occasionally, partitions do need to communicate with each other (e.g., to balance load, handle faults, or create checkpoints in a coordinated manner [23, 146]), however such events are rare compared with the workload involved in normal request handling. As a result, the scheduling and synchronization costs associated with consensus-oriented parallelization for many use-case scenarios are negligible.

3.3 Results

As part of my research, consensus-oriented parallelization has been throughly studied as a means to make replication system software ready for multi-core machines. The following paragraphs summarize important insights and observations obtained from these works.

Applicability. Up to now, the concept of consensus-oriented parallelization has been successfully integrated with four replication system software implementations with individual properties and characteristics [23, 24, 49, 146]. This includes systems addressing different fault models such as crash tolerance [49, 146], Byzantine fault tolerance [23], and even a hybrid setting [24]. The underlying replication protocols cover both major types of replication (i.e., active [23, 24] and passive [146], see Section 2.2) and parallelizing them in a consensus-oriented manner in no case required modifications to complex parts such as the consensus algorithm. All in all, this illustrates the concept's broad applicability.

Performance. Results obtained from experimental evaluations with a variety of different use cases confirmed consensus-oriented parallelization to enable a replica to use its available computing resources both effectively as well as efficiently. For Agora [146], a COP-PE version of the ZooKeeper service, for example, experiments showed the write throughput to scale linearly with the number of cores on a server until reaching network saturation, which is the case for requests as small as 128 bytes. When configured to use only a single partition (i.e., one core per server), Agora's throughput exceeds ZooKeeper's throughput by more than a factor of two for both reads and writes. This illustrates the efficiency improvements possible due to the minimized scheduling and synchronization overhead.

Relying on consensus-oriented parallelization, the prototype of COP [23], a COP-SE implementation of the Byzantine fault-tolerant replication protocol PBFT [37], is able to complete the consensus processes of more than 1.2 million requests per second using standard server hardware. For comparison, prior to the publication of the COP measurement results, the highest published throughput provided by a replication system software implementation from this domain had been about 90.000 agreed requests per second [30].

Consistency Guarantees. Relaxing the order in which requests are agreed on and processed by different replicas raises the question which impact consensus-oriented parallelization has on the consistency guarantees that the overall replicated system is able to provide. A key insights obtained from the development of multiple prototypes is that when answering this question for COP-SE it is important to distinguish between two types of requests: (1) For requests that go through the consensus process (i.e., typically writes), the guarantees are the same as in traditional systems relying on task-oriented parallelization, since all correct replicas still sequentially execute such requests in the same order. This means that correct replicas advance their states in an identical manner and clients observe these changes in the same way as in traditional systems. Consequently, if a system for example originally offers strong consistency (i.e., linearizability [91]) for agreed requests, it remains strongly consistent in its parallelized form without requiring any further modifications. (2) For requests that bypass the consensus process (i.e., typically reads), on the other hand, additional measures may be necessary to preserve the original consistency guarantees, especially for protocols in which the leader ensures strong consistency for such requests [129]. Since partitions in COP-SE are largely independent and each comprise their own leader, there is no single global entity with the same capabilities as a leader in a traditional replica architecture. Nevertheless, as shown in the Niagara paper [49], by extending the functionality of the sequencer it is possible to design mechanisms to achieve strong consistency in COP-SE even for request that bypass the agreement stage.

Parallelizing both agreement and execution with COP-PE, replicas do not have a global request sequence they can adhere to. Specifically, with partitions being only loosely coupled, the relative order of requests handled by different partitions can diverge between replicas. Providing linearizability under such conditions would make it necessary to insert synchronization points and hence would reintroduce most of the overhead saved by parallel execution. To avoid this problem, it is advisable for COP-PE systems to resort to weaker consistency guarantees that require less synchronization. Agora for example implements causal serializability [138] by using vector clocks [103] to identify and respect causal dependencies between writes across partitions [146]. Despite offering slightly weaker consistency guarantees on paper, Agora still supports the same use cases as ZooKeeper.

3.4 Main Papers

Three papers constitute the core of the research performed on consensus-oriented parallelization and therefore are included in this cumulative habilitation treatise (see reprints in Appendix B). Below, the specific contribution of each of these papers is described in detail.

Middleware '15 *Consensus-Oriented Parallelization: How to Earn Your First Million*
Johannes Behl, Tobias Distler, and Rüdiger Kapitza [23]

The paper introduces the concept of consensus-oriented parallelization and applies it to the domain of Byzantine fault tolerance. The developed system implements the COP-SE replica architecture and relies on the PBFT [37] protocol to perform active replication. Besides confirming the effectiveness of consensus-oriented parallelization for multi-cores, the system also shows the approach to be compatible with existing optimization techniques such as batching [79] and

leader rotation [155]. Equipped with 12 cores and 4 Gigabit network adapters per replica, the prototype is the first BFT system to agree on more than a million requests per second.

DSN '17 *Agora: A Dependable High-Performance Coordination Service for Multi-Cores*
Rainer Schiekofer, Johannes Behl, and Tobias Distler [146]

The paper shows consensus-oriented parallelization to be a technique that is not only limited to Byzantine fault tolerance, but also applicable to other fault models. For this purpose, the paper presents the design of a crash-tolerant coordination service called Agora, which relies on Zab [94] for replication and supports the same use cases as ZooKeeper [92]. Agora is the first case study to investigate the integration of consensus-oriented parallelization into a service that is widely used in production. Furthermore, the paper is the first to apply the approach to passive replication, and to describe and evaluate load balancing for the COP-PE architecture.

PaPoC '19 *In Search of a Scalable Raft-based Replication Architecture*
Christian Deyerl and Tobias Distler [49]

The paper presents Niagara, a second case study with a replication system software used in production, here the crash-tolerant Raft [129] implementation of the key-value store etcd [76]. Integrating consensus-oriented parallelization did not require any modifications to the core mechanisms responsible for consensus, leader election, and batching. Relying on the COP-SE replica architecture, this paper is the first to introduce the sequencer as a separate component and to examine additional tasks it may be entrusted with (e.g., the coordination of strongly consistent reads).

The initial and BFT-related work on consensus-oriented parallelization was conducted with Johannes Behl during his PhD, supervised by Rüdiger Kapitza and me. Under my supervision, Rainer Schiekofer and Christian Deyerl (both Master students) then developed and implemented parallelized variants for the CFT protocols Zab and Raft, respectively.

4 Energy-Aware Adaptation to Varying Workloads

Avoiding performance bottlenecks by enabling replication system software to utilize all available resources (as addressed by Chapter 3) is crucial during periods of high workloads in which a replicated service needs to respond to a large number of requests within a small amount of time. In practice, many distributed services experience these kinds of conditions on a regular basis, however they do not operate under them all of the time. Instead, the workload on services such as key-value stores [77, 112, 131] in production usually follows diurnal patterns. Specifically, often as a byproduct of the day/night cycle, the workload level typically varies over the course of a day [5, 14, 48]. This means that there are periods of lower utilization during which the performance requested from a service significantly drops below its maximum, causing the bottleneck problem to no longer be imminent.

This chapter presents an approach that enables replication system software to minimize the energy consumption of a service during such phases of low and medium workloads. To achieve this, the system software automatically adapts its configuration by adjusting a set of parameters at different system levels including, for example, the mapping of program modules to threads or the number of active cores on a server. Results show that switching to less powerful, but more resource-efficient configurations in the absence of high workloads can result in significant energy savings without impeding performance.

4.1 State of the Art

In today's computing infrastructures, processors are a major contributor to a system's overall energy consumption [32, 120]. How much energy exactly needs to be spent in order to handle a certain workload to a significant degree depends on a system's configuration, which for example includes the number of threads that are available to a service for processing requests. For services whose implementations are able to leverage parallelism, the provision of additional threads (up to a certain point) in general leads to an improvement in the maximum throughput they are able to achieve. On the other hand, for the type of services considered in this treatise more threads typically also result in higher energy consumption due to the increased inter-thread synchronization overhead. As a consequence, there commonly is a tradeoff between providing high performance and saving energy. To illustrate this tradeoff, Figure 7 presents a set of measurement results [64] obtained from a case study with a key-value store and two different configurations C_{perf} and C_{energy}. While the C_{perf} configuration has been designed for peak performance and therefore executes the service with 24 threads, the C_{energy} configuration in contrast addresses periods of medium workloads and hence comprises only 2 threads. With regard to the mentioned tradeoff these numbers offer two important insights: (1) Relying on more threads, C_{perf} is able to achieve a 46% higher maximum throughput than C_{energy}. (2) At a throughput of 70.000 requests per second, a level which both configurations are able to sustain, C_{energy} consumes 21% less power than C_{perf} despite handling the same workload. These two observations indicate that (with high probability) there is no single optimal configuration that would enable the service to provide peak performance when utilization is high and to minimize energy consumption when utilization is low. Since existing replication system software [30, 55] is workload-agnostic and therefore does not include support for dynamic reconfigurations, administrators are forced to statically select a configuration and thereby compromise between high performance and energy efficiency.

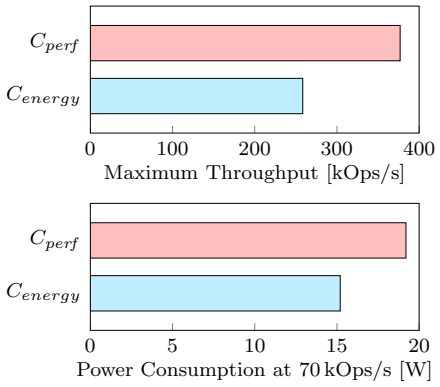


Figure 7. Measurement results for two configurations of a key-value store [64]

Independent of a specific replication system software configuration, modern servers typically adjust their power consumption by applying the concept of dynamic voltage and frequency scaling (DVFS). DVFS enables the operating system of a server to save energy during periods of low utilization by limiting the CPU's maximum speed. As it is controlled by the operating system, the technique has the advantage of being effective without requiring any modifications to the software running at higher levels. On the other hand, this property also prevents DVFS from always identifying and applying the ideal tradeoff between performance and energy consumption [65]. Specifically, there may be cases in which it is possible to further minimize power consumption without decreasing performance by disabling DVFS and instead using a power governor [114] to set the CPU to its lowest frequency. This is a direct result of the fact that DVFS has no knowledge about the specific characteristics of a service and consequently needs to make pessimistic assumptions on the extent to which itself negatively impacts the performance of a service.

4.2 Approach: Energy-Aware Reconfigurations

In order to be able to automatically adapt to varying workloads in an energy-aware manner, replication system software needs to be flexible enough to (1) offer several configurations with different characteristics regarding performance and energy consumption and (2) dynamically switch between these configurations whenever it is beneficial. Furthermore, the system software should provide interfaces to both upper layers (i.e., the service applications) as well as lower layers (i.e., the underlying operating system and hardware) that allow the software to collect feedback on the effects of its reconfiguration decisions. Figure 8 presents a replica architecture that is a result of my research and has been developed to address these requirements. Its core component is a controller whose main responsibility is to store information on the impact of system parameters at different levels and to use this knowledge to select the most suitable configuration for the current workload.

Saving Energy at Different System Levels. To create a variety of configurations with heterogeneous performance and energy-consumption characteristics, the controller relies on techniques that take effect at different system levels. The three most important ones are:

- **Mapping of Modules to Threads:** At the replication-software level, the controller is able to dynamically change the number of threads that execute the replication software and its service. To increase performance and minimize overhead such reconfigurations typically also involve a remapping of software modules to threads; a module, for example, includes a task in the processing pipeline (see Section 3.1) or a consensus partition (see Section 3.2). The dynamic remapping can be simplified by implementing modules as actors [3] so that they do not share state and only communicate by exchanging messages. As a consequence, this makes it significantly easier to migrate modules between threads in case the controller decides to perform a reconfiguration that either increases or decreases the number of active threads.
- **Mapping of Threads to Cores:** At the operating-system level, the controller can influence the thread-to-core mapping, for example by pinning each thread to a separate core and thereby reducing scheduling overhead. During periods in which there are more cores than threads, the controller deactivates unused cores to save energy.
- **Capping Power Consumption of Cores:** At the hardware level, the controller exploits modern features such as Intel's (RAPL) [93] to specify the maximum amount of power a processor is allowed to use. Low thresholds conserve energy but at the same time also decrease the performance a processor is able to provide. In contrast to DVFS, RAPL not only changes voltages and frequencies but enables further savings by making use of additional hardware features such as throttling a CPU's clock.

By combining these techniques, a controller is able to produce configurations that are tailored for different workload levels. When service utilization is high, the workload is usually distributed across multiple threads/cores, whereas during low utilization a single thread on a (power-capped) core can sometimes be sufficient to timely process all requests.

Dynamic Reconfiguration. While the system is running, the controller continuously collects information on the current workload from the replication software and if suitable adjusts the configuration. As a basis for reconfiguration decisions, the controller maintains a database containing knowledge about the characteristics of different configurations (e.g., processing latency, maximum throughput, power consumption at different throughput levels). The database is compiled during a training phase and later updated based on performance and energy-consumption measurements the controller conducts at runtime.

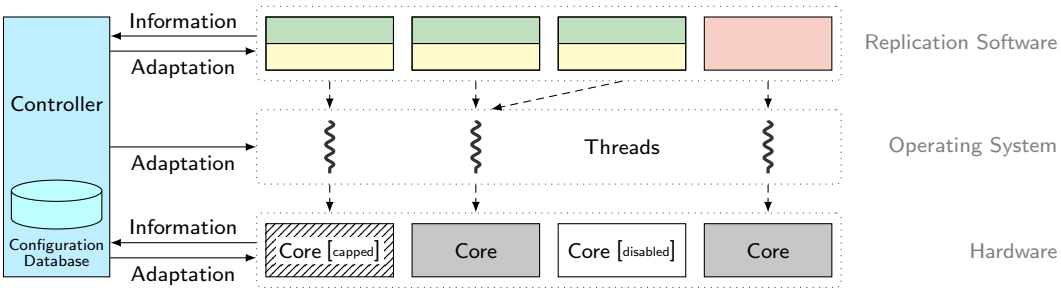


Figure 8. Energy-aware replica architecture enabling reconfigurations at different system levels

4.3 Results

To study its strengths and weaknesses, as part of my research the approach presented in Section 4.2 was integrated with systems from multiple domains [62, 64, 65], not limited to replication system software. In all cases, the combination of information collected at different layers enabled the controller to optimize energy consumption while sustaining the requested level of performance. The following paragraphs summarize further results.

Power Caps as a Means to Save Energy in Replicated Systems. One of the key outcomes of the conducted experiments is the insight that power caps can be an effective tool to minimize the energy consumption of a replicated service while handling low and medium workloads. This was not obvious from the beginning since power-capping features such as RAPL originally were designed for a different purpose: as a protection mechanism against peaks in power consumption. RAPL specifically, for example, has been applied to maintain acceptable thermal load [45] and to prevent circuit breakers from tripping [170]. Measurement results obtained from a BFT coordination service show that even in the presence of low power caps a replicated system can still perform a significant amount of work [62]. Despite the CPU-package power consumption being limited to 12 W, the coordination service for example is able to process more than 40,000 read requests per second with a latency of less than 5 milliseconds, which is often sufficient during periods of low utilization.

Integration with Execution Platforms. Since the concept of energy-aware reconfiguration does not depend on specifics of the replication logic, there is no necessity to implement it inside the replication software. As shown by Empya [63, 64] and Strome [65], the controller for example may also be placed within an underlying execution environment, which in case of Empya and Strome is a general-purpose platform for actor-based applications [4] and data-stream processing applications [101], respectively. As a key benefit, this makes energy-aware reconfigurations available to a wide spectrum of different services.

Coordinating Reconfigurations Across Servers. In a replicated system in which all replicas handle the same workload, it is usually feasible for the controller to make reconfiguration decisions solely based on information it collects from its local server. On the other hand, for distributed applications that split the workload among servers or rely on different machines to perform different tasks, it can be beneficial to coordinate reconfigurations across servers. In Strome, for example, where the output of one server represents the input of another, a central controller monitors the impact of server-local reconfigurations on overall system performance and fine-tunes the adaptations in an effort to reach a global optimum.

4.4 Main Papers

Three papers represent the core of my research on energy-aware adaptation in system software and hence are included in this cumulative habilitation treatise (see reprints in Appendix B). Below, the specific contribution of each of these papers is described in detail.

ARM '15 *Towards Energy-Proportional State-Machine Replication*
Christopher Eibel and Tobias Distler [62]

The paper introduces the idea of relying on dynamic reconfigurations at multiple system levels to improve energy efficiency in the presence of varying workloads. Specifically, the paper investigates the approach as a means to achieve energy proportionality [18] in infrastructures controlled by replication system software. A system is energy-proportional if its energy consumption correlates with the provided performance. For evaluation purposes, the paper uses a Byzantine fault-tolerant coordination service that is implemented on top of an existing replication library [55].

IC2E '18 *Empya: Saving Energy in the Face of Varying Workloads*
Christopher Eibel, Thao-Nguyen Do, Robert Meißner, and Tobias Distler [64]

The paper and its extended version [63] examine the challenges and implications of integrating the controller with a general-purpose programming and execution platform,

using the actor-based Akka [4] as example. The resulting energy-aware platform Empya can serve as a basis for the implementation of replication system software, but also for the execution of data-processing applications such as MapReduce [47] and Spark [173] jobs, which do not rely on state-machine replication for fault tolerance.

DAIS '18 *Strome: Energy-Aware Data-Stream Processing*
Christopher Eibel, Christian Gulden, Wolfgang Schröder-Preikschat, and Tobias Distler [65]

The paper is the first to present coordinated reconfigurations. The evaluated use case is Twitter's data-stream processing platform Heron [101] whose master and worker nodes have been extended with controller components that interact with each other to determine the effects of local decisions on overall system performance and energy consumption. The paper received the *Best Paper Award* at the 18th International Conference on Distributed Applications and Interoperable Systems (DAIS '18).

The work presented in this chapter was conducted with Christopher Eibel as a part of his doctoral studies that was supervised by me. His doctoral studies also included research on other energy-related topics for which he was supervised by Wolfgang Schröder-Preikschat and Timo Hönig. Under Christopher's and my supervision, three Bachelor/Master students contributed to the project: Thao-Nguyen Do implemented parts of the Empya platform. Robert Meißner created an extension for Empya that enables the platform to execute Spark applications. Christian Gulden developed and implemented the Strome prototype.

5 Techniques Tailored to Specific Use Cases

Leveraging multiple cores on a server (Chapter 3) and minimizing the energy consumption of replicas (Chapter 4), the approaches presented in the previous two chapters can support a broad spectrum of replication use cases. To complement these works, this chapter examines possibilities of further improving the efficiency of replication system software through mechanisms and concepts that are tailored to the characteristics and requirements of particular application scenarios such as heterogeneous environments, services with large states, and geo-replicated settings. Although designed with a specific use case in mind, the approaches presented in this chapter are nevertheless generally applicable in the sense that they are neither limited to a certain fault model nor a specific replication protocol.

5.1 Services in Heterogeneous Environments

Replication protocols commonly assume replicas to be fault independent, meaning that a single cause cannot lead to the simultaneous failures of multiple replicas. In practice, the probability of correlated failures can be reduced by placing replicas at different geographic locations (see Section 5.3) and/or diversifying their implementations [16, 78], for example by relying on heterogeneous off-the-shelf software components [39, 42, 60, 80–82, 84, 150, 154]. So far, the primary focus of these works was laid on the impact diversification can have on the overall resilience of a replicated system against faults.

In general, existing studies do not address the fact that heterogeneity not only has an influence on fault tolerance, but typically also affects the individual performance of replicas, for example as a result of the use of different programming languages, service implementations, and/or operating systems. The same is true for environments in which the server hardware or the amount of available resources (e.g., the number of cores a replica has at its disposal) differs among replicas. A common example of such environments are cloud infrastructures where the performance can vary significantly across virtual machines, even if the virtual machines on paper are of the same instance type [130].

For simplicity, most replication protocols either implicitly or explicitly assume replicas to be equally powerful and thus provide a homogeneous level of performance, independent of whether their implementations are diversified or not. As discussed in the following by example of BFT replication, in application scenarios in which a replicated system runs in an heterogeneous environment, this approach may prevent replication system software from fully utilizing all available resources on servers that are more powerful than others.

State of the Art. Figure 9 illustrates the problem based on an example scenario in which the replicated system is required to tolerate one Byzantine fault. In the shown setting, there are two categories of servers: (1) *fast* machines with 2.4 GHz processors and (2) *slow* machines with 1.6 GHz processors, meaning that the fast servers have 50% more processing resources than the slow servers, as indicated by the respective height of a server's bar.

A common solution to tolerate $f = 1$ Byzantine fault is to rely on the PBFT [37] protocol, which for this purpose requires a minimum of $n = 3f + 1 = 4$ replicas. However, as depicted in Figure 9a, applying PBFT with four replicas would result in the replication system software leaving unused about one third of the fast server's processing resources. This is a consequence of the fact that in PBFT, as it is the case for many other fault-tolerant protocols, the system can only make progress if a sufficient number of replicas participate in the agreement process; in PBFT this quorum threshold is $\lceil \frac{n+f+1}{2} \rceil = 3$ replicas. Or in other words, the overall performance of PBFT is limited by the $\lceil \frac{n+f+1}{2} \rceil$th fastest server, independent of whether or not some of the faster servers have spare processing resources.

Unfortunately, providing further resources in the form of additional servers in this scenario also does not solve the problem. Technically, it is possible to operate PBFT with more than the minimum number of required replicas (see Figure 9b), however instead of improving performance such

(a) PBFT [37] with 4 replicas

(b) PBFT [37] with 5 replicas

(c) Omada [72]

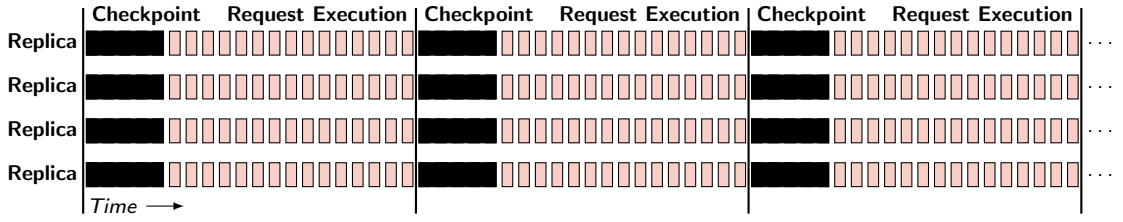
Figure 9. CPU resource utilization in different replicated systems tolerating one Byzantine fault. Bar heights symbolize the capabilities of individual servers, with higher bars indicating more powerful processors. Replicas consist of (combined) modules for agreement (A) and execution (E).

a measure typically has the opposite effect. This has mainly two reasons: (1) Increasing the replica-group size to $n = 5$, PBFT needs to resort to larger agreement quorums, which in turn means that for the example setting the performance bottleneck shifts from the third fastest server to the fourth fastest server, that is in the wrong direction. (2) With more replicas participating in the consensus process, PBFT needs to exchange more messages between replicas, which not only requires additional processing and network resources but usually also negatively impacts performance [72].

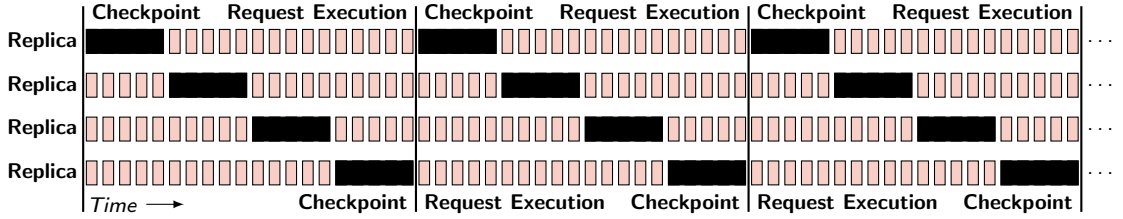
Approach: Group-based System Architecture. To minimize the amount of resources lying idle in heterogeneous environments, replication system software must be enabled to flexibly move work to the locations at which the necessary resources are actually available. The Omada system architecture [72], a result of my research, solves this problem by organizing a replicated system into different module groups, specifically one group that comprises the execution stage, and a configurable number of groups responsible for handling request agreement. Given this basic concept, the Omada system architecture can be seen as an extension of the partition-based COP-SE replica architecture presented for consensus-oriented parallelization in Section 3.2. However, there are three important differences:

- **Flexible Distribution:** Using COP-SE all replicas host all partitions, whereas in the Omada architecture the module groups are distributed across different replicas, as shown in Figure 9c. In this example, the agreement stage is split into two groups A_1 and A_2 to enable a fine-grained distribution of work. This way, fast servers with many resources can be assigned more tasks (e.g., the responsibility for both agreement groups as well as the execution group) compared with the work that needs to be performed by slow servers (e.g., hosting only two of the three groups).
- **Efficient Execution:** Omada exploits the insight that when agreement and execution are separated, $2f + 1$ execution replicas are sufficient to tolerate up to f Byzantine faults [171]. In the example depicted in Figure 9c, this enables Omada to save resources my limiting the execution-group size to three servers (i.e., Servers 1 to 3).
- **Heterogeneous Groups:** While COP-SE uniformly distributes the agreement workload among its partitions, agreement groups in Omada may possess heterogeneous configurations. By relying on group-specific maximum batch sizes (i.e., upper thresholds for the number of requests each group is allowed to handle within a single consensus instance), it is for example possible to create agreement groups with diverse resource-consumption characteristics. This is important since it enables Omada to adapt the workload distribution to the amount of resources available in the system.

In order to assist practitioners in deploying their systems as well as to increase resource utilization, Omada comes with a systematic approach to determine the number and placement of groups in a heterogeneous environment. In a first step, this includes a measurement-based assessment of the actual processing resources available on each participating server, which typically involves the execution of a small microbenchmark performing cryptographic computations, as are used by replication protocols to authenticate messages [37]. Based on the results obtained from these measurements, each server is assigned a certain number of *performance points*, which is a unitless metric to quantify



(a) Lockstep checkpoints



(b) Checkpoint rotation

Figure 10. Comparison of state-of-the-art checkpointing techniques

the relative performance differences between servers. Next, it is necessary to estimate the ratio between the resource demands of an agreement group and the resource demands of the execution group. This property is also expressed in terms of performance points and depends on the specific protocol used for replication. In a final step, all the information gathered up to this point serves as input for the formulation of an integer linear program [132]. This program then allows a solver to automatically analyze all possible assignments of groups to servers and thereby determine an optimal Omada configuration.

Results. So far, the Omada system architecture has been implemented and evaluated in conjunction with two different BFT protocols, namely PBFT [37] and Spinning [155]. In both cases, Omada enables the replication system software to rely on agreement groups that employ the respective consensus-protocol implementation as a black-box component, requiring no modifications to the essential protocol parts. In general, there is no limitation preventing the approach to also be applied to CFT agreement protocols that establish a stable total order on incoming client requests. The rationale behind the primary focus on BFT was the fact that Byzantine fault tolerance typically requires significantly more resources than crash tolerance, and consequently it becomes all the more costly if resources lie idle.

Another major result of the research on Omada is the insight that there are several opportunities for the replication system software to optimize the interaction between groups in cases in which two modules of different groups reside on the same server and consequently can be assumed to trust each other, as they do in traditional non-group-based architectures. For example, if an execution module receives an ordered request from a co-located agreement module, the execution module may immediately accept the message without the need to consult other agreement-group members, thereby saving resources.

Experiments conducted with a coordination service show that for a heterogeneous environment such as the one in Figure 9 (i.e., two fast servers and three slow servers) Omada is able to improve performance by utilizing resources more effectively [72]. Specifically, integrated with the Omada architecture the maximum throughputs achieved by PBFT and Spinning increase by 15% and 10%, respectively. A second important outcome of the experiments is the observation that the performance improvements possible with Omada further increase if the differences in available resources between servers become larger, that is if servers are even more heterogeneous. For a six-server setting (i.e., two fast servers and four slow servers) in which each slow server only has a third of the computing power of a fast server (instead of two thirds as in the scenario in Figure 9), Omada for example enables throughput improvements of 108% (PBFT) and 168% (Spinning).

5.2 Services with Large States

Replication protocols usually make no specific assumptions on the size of the application state they need to keep consistent across replicas. From a theoretical point of view this approach is reasonable since the state size in general has no influence on the correctness of a protocol. However, when it comes to building actual systems, the amount of application data involved suddenly can become an important factor impacting the efficiency of replication system software. Specifically, this is true with regard to the problem of creating

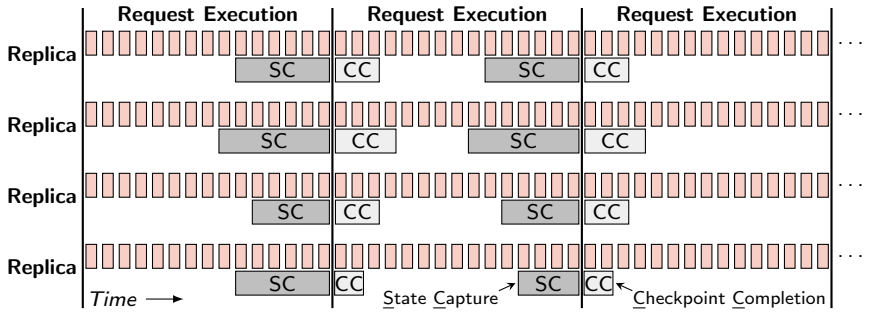


Figure 11. Creation of deterministic fuzzy checkpoints in parallel with request execution

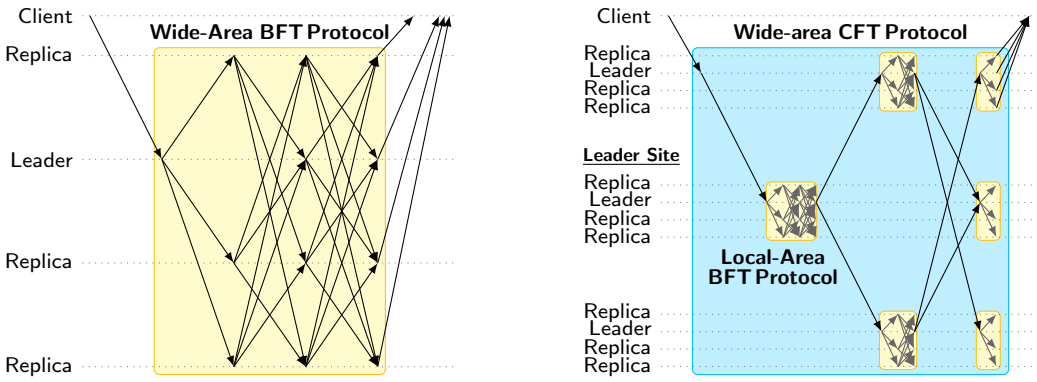
periodic checkpoints, which as explained in Section 2.1 are necessary to perform garbage collection and to update new or trailing replicas via state transfer. In replicated systems, checkpoints are a representation of the application state at a particular point in time, typically in between the execution of two client requests with neighboring sequence numbers. Persisting the contents of such a checkpoint by writing it to stable storage commonly requires means to capture and serialize all relevant application objects, which is an operation that can result in significant overhead for replicated services with large states.

State of the Art. As illustrated in Figure 10a, the most widely used checkpointing method in replicated systems is to instruct replicas to create checkpoints in lockstep, meaning that all replicas in the system produce snapshots for the same sequence numbers [37]. As a key benefit, this approach makes it possible to directly compare the checkpoints generated by different replicas and thereby detect faulty copies of the application state. For this to work, it is crucial that checkpoints consistently reflect the contents of the replicated application after a certain sequence number. A straightforward way to ensure this is to first suspend the execution of requests, then serialize all state objects to a suitable location (e.g., a file on disk), and afterwards resume request processing. Although leading to consistent checkpoints, this technique unfortunately has the main drawback of reducing the availability of services with large states, for which the state serialization can take a significant amount of time during which the system does not process requests. The period of unavailability can be shortened by only storing the changes since the latest checkpoint [37, 39], however this still leaves the problem that all replicas in the system are affected at the same time.

To address this issue, Bessani et al. [30] proposed an approach that enables replicas to snapshot their application instances at different sequence numbers, as shown in Figure 10b. This way, one replica at a time can focus on request execution while the rest of the group continues to provide the service. Notice that the improvement in availability comes at the cost of checkpoints from different replicas not being directly comparable anymore. To verify a checkpoint cp_A based on another (later) checkpoint cp_B using this technique, a replica first needs to load the cp_A contents received from one replica, process subsequent requests, and finally compare the resulting application state to the checkpoint cp_B provided by a different replica. In CFT systems in which replicas trust each other this does not pose a problem, however in BFT systems the necessity to load unverified state data (i.e., checkpoint cp_A) entails the risk of being exploitable by adversaries [53].

Approach: Deterministic Fuzzy Checkpoints. To be efficient and generally applicable in both CFT and BFT systems with large states, a checkpoint technique should not require the temporary suspension of request execution, and still produce comparable checkpoints that can be verified without loading them first. This problem served as a starting point for my research on *deterministic fuzzy checkpoints (DFC)* [68], which was inspired by the fuzzy-checkpointing techniques proposed for early in-memory databases [90, 113, 144]. As shown in Figure 11, using DFC the replication system software produces snapshots while continuously processing requests. The checkpointing process itself consists of two phases:

- **State Capture Phase:** In this phase, the replication system software collects all information necessary to later be able to produce the actual checkpoint. Specifically, a separate checkpointer thread iterates over all state objects and copies them. With request execution continuing, the resulting overall snapshot is likely to neither reflect a consistent state nor the exact state of the application at the intended sequence number. Therefore, to not miss any intermediate changes the system software in addition also records all modifications that take place after a state object has been copied.
- **Checkpoint Completion Phase:** This phase transforms the fuzzy snapshot obtained during state capture into a comparable and consistent checkpoint. For this purpose, also in parallel with request execution, the replication system software takes the state-objects snapshot from the previous phase and applies all recorded



(a) Single-tier protocols (e.g., PBFT [37])

(b) Hierarchical protocols (e.g., Steward [10])

Figure 12. Message flow in wide-area environments for state-of-the-art replication protocols

changes to it. At the end, the checkpoint produced by DFC is indistinguishable from the checkpoints traditional lockstep techniques would create for the same sequence number.

Depending on the characteristics and requirements of the particular use-case scenario, there are different ways to implement the DFC checkpointing mechanism in replication system software. For example, if the application interface already contains means to notify the system software about imminent object modifications [39], a straightforward approach is to perform most of the work inside the replication system software and in an application-independent manner. On the other hand, there is the possibility to let the replication system software only coordinate the checkpointing process and outsource the monitoring of state changes to the service application. Compared with the generic solution, this variant requires additional implementation work but in return enables further service-specific optimizations such as a more efficient management of state modifications [68].

Results. As part of the research on DFC, the technique has been successfully used to not only create full checkpoints (i.e., copies of the entire application state) but also differential checkpoints (i.e., snapshots of the changes since the latest checkpoint). In both cases, DFC was found to already improve the availability of replicated services with comparably small states. For a key-value store with a 3 GB database the use of DFC for example resulted in no notable downtime during the creation of full checkpoints; without DFC, request execution had to be suspended for about 4.7 seconds once every checkpoint interval [68].

A second important insight is the fact that DFC is able to minimize the duration between (1) the point in time at which the state reflected by a checkpoint exists in the application and (2) the point in time at which the corresponding fully serialized checkpoint becomes ready. More specifically, traditional approaches start the creation of the checkpoint for a sequence number s immediately after having processed the request assigned to sequence number s. In contrast, using DFC a replica at this point already initiates the final checkpoint completion phase, after having finished most of the work in advance.

5.3 Geo-Replicated Services

For use-case scenarios in which the clients of a replicated service are scattered across the globe, a common approach is to also geopraphically distribute the server-side of the system. The rationale behind this strategy is to give clients faster access to the service by placing replicas in close distance to them and thereby reduce transmission times between clients and replicas. On the downside, hosting replicas near clients typically also means that they are farther apart from each other and as a result it takes longer for them to reach consensus. This predicament illustrates the necessity for research on how replication system software can minimize end-to-end response times in geo-replicated settings.

State of the Art. Replicating a service across multiple geographic sites is a highly relevant problem in practice and therefore has been extensively studied in the context of both CFT and BFT. Overall, the proposed solutions can be divided into two high-level categories:

- **Single-Tier Protocols:** Protocols in this category [37, 117, 124, 156] share the commonality that they only host a single replica at each participating location and thus require all protocol mechanisms (e.g., consensus algorithm, leader election, state transfer) to be entirely executed across wide-area connections (see Figure 12a).
- **Hierarchical Protocols:** Systems belonging to this category [8, 10, 88] deploy multiple replicas at each site and implement the overall replication protocol as a combination of subprotocols at two distinct hierarchical levels, as shown in Figure 12b: (1) a local-area

subprotocol that is executed among replicas residing at the same geographic location and (2) a wide-area subprotocol that connects different sites.

A major benefit of the hierarchical approach is the fact that by designing the local-area subprotocol in a fault-tolerant manner, the set of replicas at each site is able to provide stronger resilience guarantees than a single replica. Compared with single-tier protocols, this makes it possible to reduce the complexity of the protocol run at the wide-area level.

Independent of the specific category, in leader-based replication protocols client requests need to pass through the agreement leader in order to be considered in the consensus process. As a consequence, the end-to-end response times provided by a geo-replicated system can vary significantly depending on where the current leader is located [70, 117, 148]. Some protocols aim at addressing this issue by rotating the leader role among replicas [117, 118, 123, 155, 156], there enabling a client to save time by submitting requests to the replica closest to its own location. Unfortunately, experiments in real-world environments have shown that the strategy of rotating the leader role overall does not necessarily always lead to better response times than a fixed leader at a well-connected site [148].

When a service is used by clients from all over the world, it is likely that not all of those clients continuously access the service. Instead, accesses in practice often follow diurnal patterns (see Chapter 4) and as a consequence of the day/night cycle the origins of client requests usually change over the course of a day. Since only few replication protocols support dynamic reconfigurations of the replica group [30, 105, 129, 142], leaderless protocols [61, 124] and weighted-voting schemes [26, 148] can be used to reduce response times in deployments with a larger number of static replica sites. However, this comes at the cost of an increased network overhead necessary to keep replicas consistent.

Approach: Clustered System Architecture. Geo-replication protocols should provide low latency and support for flexibly adding/removing new replica sites to/from the system at runtime. The cluster system architecture Spider [73], a result of my research, addresses these issues by splitting the replication system software into two main parts, agreement and execution [171], and instantiating these parts in different replica clusters and at different geographic locations. Precisely, Spider comprises one agreement cluster and multiple execution clusters, which are all dimensioned to tolerate a preconfigured number of replica faults. The agreement cluster is responsible for running the consensus protocol and establishing a global order on client requests, while the execution clusters host copies of the service application, handle the interaction with clients, and process the requests.

The Spider architecture is specifically designed for deployment in public clouds and tailored to leverage the characteristics of the infrastructures providing them. In particular, the approach exploits the fact that today's clouds are organized as a collection of *regions*, which are further divided into *availability zones* [7, 85, 122]. Availability zones are constructed in such a way that they represent distinct fault domains. They are typically hosted in data centers that have dedicated power supply systems and networking and are located several tens of kilometers apart from each other. All these measures are taken to minimize the probability of two availability zones being affected by correlated failures even if they reside in the same region. Spider benefits from these properties by distributing each agreement/execution cluster across different availability zones of a region. This way, the communication within a cluster can be efficiently performed with low latency while still enabling the cluster as a whole to serve as a fault-tolerant entity. In contrast to traditional single-tier or hierarchical systems, this strategy circumvents the need to execute expensive replication protocols over long-distance links. Instead, all complex protocols (e.g., the consensus algorithm) in Spider are run inside a cluster and therefore limited to a single region. Wide-area connections between regions, on the other hand, are primarily responsible for forwarding the protocol outputs of one cluster to another cluster.

As illustrated in Figure 13, clients in Spider submit requests to their nearest execution cluster. Since the execution replicas themselves manage copies of the service state and process all modifications in the same order, they are able to immediately respond with weakly consistent results providing consistent-prefix guarantees [151]. For clients requiring strongly consistent reads or writes (i.e., linearizability [91]), an execution cluster forwards the corresponding requests to the agreement cluster where the consensus protocol assigns each request with a unique and stable sequence number. Due to the consensus process being limited to the agreement cluster, the specific algorithm used for the agreement of requests is fully independent of execution-cluster implementations, making it possible to integrate Spider with different agreement protocols. After the consensus is complete, the agreement cluster delivers the requests to all execution clusters, which process them in the order of their sequence numbers and (if connected to the client) return the result.

To simplify system design and implementation, the research on Spider also involved the development of a replication system software component for handling the exchange of information between clusters: the inter-regional message channel (IRMC) [73]. IRMCs are unidirectional m-to-n channels that connect m replicas of one cluster to n replicas of another cluster. For the messages sent through them, the channels offer first-in-first-out semantics as well as a flow-control mechanism that prevents the cluster of sender replicas from overwhelming the cluster of receiver replicas. Besides enabling Spider's modular design, IRMCs also greatly simplify the dynamic addition/removal of execution clusters.

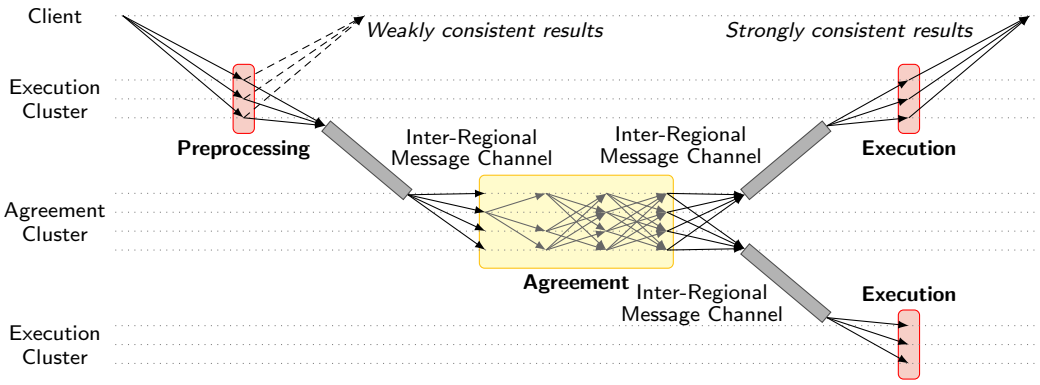


Figure 13. Clustered system architecture for geo-distributed replicated systems

Results. The Spider system architecture has been integrated with an existing BFT replication system software implementation and evaluated for a geo-replicated key-value store in the public Amazon EC2 cloud [73]. The measurement results indicate that Spider enables low end-to-end response times for several operations. For strongly consistent writes, for example, the overall latency in Spider is dominated by a single wide-area round trip between a client's nearest execution cluster and the agreement cluster. In general, this constitutes an improvement over the number of long-distance hops required by single-tier BFT protocols such as PBFT [37] or hierarchical protocols such as Steward [10], as previously shown in Figure 12. As a consequence, in the evaluated setting Spider completed writes up to 95% (PBFT) and 94% (Steward) faster than the other replication protocols.

Another key takeaway from the experiments is that in Spider end-to-end response times are fairly stable independent of where the current agreement leader is located. This is a result of the fact that, while in PBFT and Steward the leader role can move between all participating sites, the leader replica in Spider is guaranteed to be one of the replicas in the agreement cluster. Hence, instead of hundreds or thousands of kilometers as in traditional system architectures, the Spider leader solely shifts among different availability zones of the same cloud region, which typically has only a minor impact on overall latency.

An analysis of existing public-cloud infrastructures has shown a wide range of deployment options for Spider-based systems. At the time the study was conducted in 2020, the three major public-cloud providers Amazon, Microsoft, and Google together already offered a total of 54 compute-cloud regions that comprise three or more availability zones and therefore are qualified to host a Spider execution cluster tolerating one Byzantine fault. Four of those regions furthermore are sufficiently equipped to accommodate a corresponding agreement cluster due to having four or more availability zones. To support deployments with larger agreement clusters there are mainly two possible ways to achieve this: (1) If the designated agreement region lacks the required number of availability zones, some of the agreement replicas may be placed in neighboring regions of the same cloud provider. Although this approach does not endanger correctness, it may come at the cost of increasing volatility in response times in the presence of leader changes. (2) The agreement cluster may be distributed across availability zones of different cloud providers that are located in relatively close distance to each other. Apart from keeping response times stable, this strategy also has the benefit of making the system resilient against faults that affect an entire provider. Such diversification in deployment has previously been proposed for various use-case scenarios including, for example, storage services [1, 28].

5.4 Main Papers

The core of my research on application-specific replication system software comprises three papers included in this cumulative habilitation treatise (see reprints in Appendix B). In the following, the specific contribution of each of these papers is described in detail.

Computing '19 *Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers*
Michael Eischer and Tobias Distler [72]

The paper and its conference version [69] are the first to analyze the impact of servers with heterogeneous characteristics on the performance and resource efficiency of replication system software. To address the identified problems, the papers present the Omada approach and implementation, which improves the flexibility and scalability of replication system software by parallelizing the agreement stage into heterogeneous groups. Omada was evaluated with two protocols: PBFT [37] and Spinning [155].

SRDS '19 *Deterministic Fuzzy Checkpoints*
Michael Eischer, Markus Büttner, and Tobias Distler [68]

The paper proposes deterministic fuzzy checkpoints as a means for replication system software to efficiently snapshot replicated BFT services with large states. To underline the broad applicability of the technique, the paper discusses alternatives for integrating the fuzzy-checkpointing mechanism with existing replication system software architectures. In addition, it evaluates the approach in conjunction with both full as well as differential checkpoints, showing significant improvements in service availability.

Middleware '20 *Resilient Cloud-based Replication with Low Latency*
Michael Eischer and Tobias Distler [73]

The paper introduces the clustered system architecture SPIDER in the context of BFT geo-replication and discusses details on how to integrate the architecture with today's public-cloud infrastructures. Specifically, the paper explores ways to support both strongly and weakly consistent operations, to dynamically add new replica sites to the system, and to offer multiple implementations of inter-regional message channels with different characteristics. Experiments conducted with a prototype on Amazon EC2 [6] show SPIDER to provide lower latency than existing approaches for wide-area environments and response times to remain stable even across different agreement-leader replicas. The paper received the *Best Student Paper Award* at the 21st Middleware Conference (Middleware '20).

The work presented in this chapter was conducted with Michael Eischer during his doctoral studies, supervised by me. Under our both supervision, in his Bachelor thesis Markus Büttner implemented parts of the mechanism for creating deterministic fuzzy checkpoints.

6 Conclusion

Dedicated to improving the efficiency of reliable replication system software, this treatise has presented several approaches and techniques to advance the state of the art. This section summarizes the key contributions and outlines ways to combine them with each other. In addition, the section includes a discussion on how the developed concepts can be applied to a category of replicated services that in recent years became highly relevant in practice: blockchains. Finally, the section describes open challenges in the domain of replication system software research that can serve as basis for future work in this area.

6.1 Summary of Contributions

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

Consensus-Oriented Parallelization is a parallelization scheme that distributes the responsibilities of a replica across loosely coupled partitions. This enables the system software to effectively utilize multiple cores on each server in an efficient manner.

Energy-Aware Reconfigurations take effect at different system levels and for example change the mapping of program modules to threads or the power-consumption threshold of a processor. They are a means for the replication system software to reduce the energy footprint of a replica during periods of low and medium workloads.

Group-based System Architectures divide the agreement stage of a replicated system into multiple non-uniform groups. By instructing powerful replicas to participate in more groups, the system architecture makes it possible to increase performance as a result of improving resource utilization in environments with heterogeneous servers.

Deterministic Fuzzy Checkpoints enable the replication system software to create snapshots of the service state in parallel with request execution. Such a mechanism is particularly useful to achieve high availability for applications with large states.

Clustered System Architectures exploit the special properties of today's public-cloud infrastructures to minimize the amount of long-distance network traffic. For geo-replicated systems, this leads to less complex system designs and enables low latency.

For all of these approaches, there exists at least one prototype implementation that has been used for evaluation. The measurement results obtained from these experiments have been key in identifying the individual strengths and weaknesses of the proposed concept.

6.2 Compatibility of Approaches

The approaches presented in this treatise have been designed for different objectives (see Table 1), which raises the question whether they are compatible with each other. Technically, there is no reason that would prevent an integrated replication system software implementation combining all of the proposed concepts. However, this does not necessarily mean that building such an implementation would actually be reasonable in practice since the common use cases of replicated services typically do not require all the developed mechanisms within a single system. Consequently, it is likely to make more sense to combine a few selected approaches as part of a special purpose replication system software. The remainder of this section discusses two examples of composability in more detail.

Supporting the Full Spectrum of Workloads. Addressing periods of high and low utilization, respectively, consensus-oriented parallelization (see Section 3.2) and energy-aware reconfigurations (see Section 4.2) are ideal candidates for integration within the same implementation. By effectively using all available cores on a server the former could ensure that the replicated service is able to sustain high throughput during workload peaks, whereas the latter would

Approach	Description	Primary Objective	Scope
Consensus-oriented parallelization	Section 3.2	High throughput	Replica
Energy-aware reconfigurations	Section 4.2	Saving energy	Replica
Group-based system architecture	Section 5.1	Scalability	Replica cluster
Deterministic fuzzy checkpoints	Section 5.2	High availability	Replica
Clustered system architecture	Section 5.3	Low latency	Entire system

Table 1. Comparison of the approaches presented in this treatise

be responsible for saving energy at times when clients issue fewer requests to the service. On a technical level, there is especially one question that should be answered when combining the two approaches: How to deal with the fact that there are multiple partitions, which on the one hand enables parallelism when utilization is high, but on the other hand constitutes overhead (e.g., additional network connections, need for load balancing) when utilization is low? One way to handle this issue is to keep the number of partitions static and only vary the assignments of partitions to threads, for example instructing a single thread to process all partitions during periods of low workloads. Another possibility is to minimize overhead by providing an auxiliary mechanism to dynamically change the number of partitions as part of the energy-aware reconfigurations.

Nesting Architectures. As shown in Table 1, the proposed architectures take effect at different scopes. The clustered Spider system architecture (see Section 5.3) separates a replicated system into an agreement cluster and multiple execution clusters. In contrast, the group-based architecture (see Section 5.1) focuses on a single cluster, for which it enables an extended distribution across additional servers. Finally, consensus-oriented parallelization defines the architecture of individual replicas (see Section 3.2). Consequently, if considered useful, the agreement cluster of a Spider system could use a group-based approach to include additional replicas for scalability, and internally every replica could organize each of these groups as a collection of partitions handled by different threads.

6.3 Applicability to Permissioned Blockchains

Apart from the use cases considered in previous chapters (e.g., key-value stores, coordination services), in recent years an additional domain of application scenarios for replicated systems, and especially BFT protocols, has emerged: permissioned blockchains [11, 19, 27, 86]. In contrast to permissionless blockchains, as for example used for Bitcoin [125], in permissioned blockchains an access-control mechanism ensures that only a selected set of nodes (instead of arbitrary nodes) are able to participate in system operations. Hence, permissioned blockchains better fit the system model of existing CFT and BFT protocols, which as discussed in Section 1.1 typically assume a replica group whose size and composition is a matter of controlled configuration, for example by a trusted administrator.

Permissioned blockchains in general rely on system architectures that follow the execute-verify principle [97], which means that replicas first process client requests and afterwards agree on the order in which to apply the corresponding state updates [11]. Compared with traditional replicated systems, which first reach consensus on requests and then execute them (see Section 2.1), from an overall perspective this constitutes a reordering of protocol steps and therefore can be considered a new paradigm. On the other hand, as discussed in detail my survey paper [53], it is also possible to regard permissioned-blockchain systems as compositions of two services: (1) a user-facing front-end application that transforms user requests into state transactions and (2) a back-end application that verifies the transactions and maintains a history of the corresponding blockchain state. As illustrated in Figure 14, from this point of view processes of the front-end application are essentially clients of a traditionally replicated back-end application, with the exception that these processes themselves manage a considerable amount of application state. Specifically, for the back-end part of the blockchain, system designers commonly assume a similar system model as is used for many existing replicated architectures and protocols.

With regard to the approaches of this treatise, this means that they are also applicable to the domain of permissioned blockchains, where they especially could be of benefit to improve efficiency and availability of the back ends of such systems. In particular, consensus-oriented parallelization (see Section 3.2) for example may enable a back end to sustain high throughputs of transactions. Furthermore, the deterministic fuzzy checkpointing technique presented in Section 5.2 could be used to create snapshots of the (often large) blockchain history without the need to temporarily suspend the processing of transactions.

6.4 Open Challenges

Despite decades of advances in research, there is still room for improvement when it comes to the implementation of replication system software. This section discusses several open issues that have been identified as a byproduct of the work presented in this treatise.

Closing the Gap Between Specification and Implementation. One of the most important problems of both CFT and BFT replication is the fact that it is inherently difficult to implement replication system software based on existing

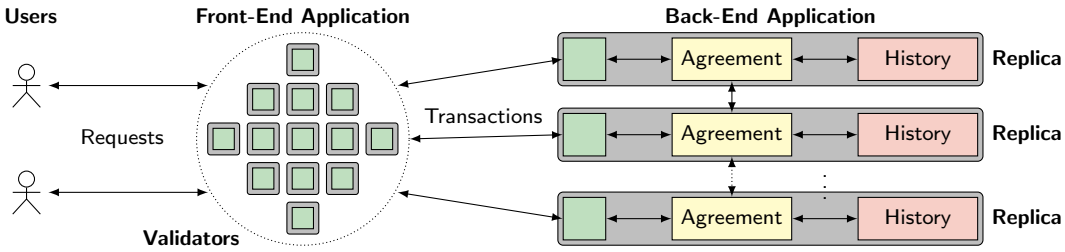


Figure 14. Composite system architecture used for permissioned blockchains

protocol specifications. This is a result of the common approach to develop protocols by primarily focusing on the tasks essential for safety and liveness, thereby for example abstracting from practical problems related to replica interaction such as flow control. For the crash-tolerant Paxos [104], for example, this led to a significant amount of secondary literature whose sole purpose is to discuss how the protocol works and how to implement it [33, 40, 98, 121, 153]. Causing already complications for CFT protocols, the problems become even larger for the more complex BFT protocols. To address these issues, research should aim at finding means to specify replication protocols in a way that makes it possible to infer a direct implementation.

Minimizing the Impact of Message Authentication on Protocol Design. Many replication protocols, especially from the BFT domain, for correctness require their replicas to communicate via authenticated messages. Unfortunately, with different authentication algorithms providing different properties, the particular scheme in use commonly has an impact on protocol design. MAC-based authenticators [37], for example, typically are less powerful than signatures [141] due to MACs only being able to prove the authenticity of a message to a specific recipient, but not to others. Although this difference may seem small, it generally affects the number of message exchanges needed to reach consensus. For PBFT, Castro for example discusses several authentication-scheme-specific protocol variants in his dissertation [36], in addition to the already differing versions presented in the corresponding conference [37] and journal [38] papers. Most other researchers did not go to the same length as Castro and designed their protocols for particular authentication schemes, often requiring different schemes for different sub-protocols [9, 43, 99, 157]. From a system software perspective this situation is not ideal since it limits flexibility and minimizes the reusability of components. As a result, it would be beneficial to develop protocols in which authentication schemes are not hard-wired but replaceable without further modifications.

Diversifying Replication System Software at Multiple Levels. Due to the high costs for code development and maintenance, existing replication system software usually relies on the same implementation for all replicas. Consequently, a single bug or vulnerability can lead to the failure of the entire system. Although there are approaches to automatically diversify implementations [136, 143], they so far are not broadly used in practice and furthermore cannot be directly applied to software implemented in interpreted programming languages (e.g., Java [30, 44, 55, 157]). Hence, to increase the resilience of replication system software it is crucial to develop means that facilitate the introduction of heterogeneity at all system levels including communication, authentication, and agreement protocols.

6.5 Concluding Remarks

My work on replication system software started in the context of my diploma thesis [50] more than a decade ago. The approaches and techniques described in this habilitation treatise represent the most important of my latest research contributions to this area. The following paragraphs summarize three key insights gained throughout all this time.

Resource Awareness. Managing the provided resources and making them available to applications is one of the main responsibilities of system software in general, and thus the same of course also applies to replication system software. While traditional works in this domain usually leave most of the resource management to lower layers, especially the operating system, this treatise examined the benefits of making all replication system software layers resource aware, including the replication logic and its execution environment. Among other things, this approach enabled the development of replication protocols that use the available processing and network resources more effectively and efficiently than existing protocols. Furthermore, resource awareness makes it possible to save energy during periods of low and medium utilization without affecting the quality of service of the replicated application. As illustrated in the context of geo-replication, it is not always sufficient to only know the amount and type of the resources available. For some settings in addition it is also crucial where the resources can be used in order to improve performance.

Protocol Independence. Although technically feasible, replication system software implementations ideally should not only support a single fault model or consensus algorithm. As a consequence, to increase reusability it is beneficial to

develop new concepts in such a way that they are applicable to both CFT and BFT. The research presented in this treatise was guided by this principle and consequently many of the proposed approaches have been integrated with a variety of replication protocols. The group-based system architecture, for example, has been examined for two different agreement processes. Moreover, consensus-oriented parallelization has been explored for two fault models (i.e., CFT and BFT) and both main forms of replication (i.e., active and passive). A crucial step towards the reusability of system parts is modularity, as for instance shown by the clustered system architecture for geo-replicated services that enables the use of different consensus algorithms inside the agreement cluster and without the need to modify execution clusters. Another example of a modular mechanism is deterministic fuzzy checkpointing, which creates checkpoints that are indistinguishable from those produced by traditional methods.

Prototype-based Evaluation. In replication system software research, new ideas sometimes appear promising on paper but do not actually provide the anticipated benefits in real-world deployments. Therefore, it is essential to assess the advantages and disadvantages of developed approaches based on experimental evaluations, not just analytical analyses or simulations. This treatise accounts for this demand by relying on at least one prototype implementation for each of the presented approaches and drawing conclusions based on measurement results obtained with replicated applications such as key-value stores and coordination services, which constitute typical examples of use cases in practice.

A Publications

All journal, conference, and workshop publications that are listed in the following underwent a formal peer-review process, which in some cases involved several rounds. After having been selected for publication, the papers were printed in journals or conference proceedings, and/or added to widely recognized digital libraries (e.g., ACM, IEEE).

Journal Papers: [53, 55, 72, 165]

Conference Papers: [21, 23, 24, 54, 57, 58, 60, 64, 65, 68, 69, 73, 95, 110, 111, 146, 160–162, 164]

Workshop Papers: [20, 22, 49, 56, 59, 62, 66, 70, 71, 74, 96, 139, 163]

Other Peer-Reviewed Contributions: [67, 150, 159, 166, 167]

Invited Papers: [52]

Theses: [50, 51]

Technical Reports: [25, 63]

Workshop Proceedings: [29]

B Paper Reprints

The following 10 peer-reviewed publications constitute the main part of this cumulative habilitation treatise. They are provided as personal reprints here. All copyrights remain with the authors and/or the respective publishers (i.e., ACM, IEEE, and Springer).

System Software for Replicated Services

CSUR '21 *Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective*
Tobias Distler [53]

Effective and Efficient Use of Multi-Cores

Middleware '15 *Consensus-Oriented Parallelization: How to Earn Your First Million*
Johannes Behl, Tobias Distler, and Rüdiger Kapitza [23]

DSN '17 *Agora: A Dependable High-Performance Coordination Service for Multi-Cores*
Rainer Schiekofer, Johannes Behl, and Tobias Distler [146]

PaPoC '19 *In Search of a Scalable Raft-based Replication Architecture*
Christian Deyerl and Tobias Distler [49]

Energy-Aware Adaptation to Varying Workloads

ARM '15 *Towards Energy-Proportional State-Machine Replication*
Christopher Eibel and Tobias Distler [62]

IC2E '18 *Empya: Saving Energy in the Face of Varying Workloads*
Christopher Eibel, Thao-Nguyen Do, Robert Meißner, and Tobias Distler [64]

DAIS '18 *Strome: Energy-Aware Data-Stream Processing*
Christopher Eibel, Christian Gulden, Wolfgang Schröder-Preikschat, and Tobias Distler [65]

Techniques Tailored to Specific Use Cases

Computing '19 *Scalable Byzantine Fault-tolerant State-Machine Replication on Heterogeneous Servers*
Michael Eischer and Tobias Distler [72]

SRDS '19 *Deterministic Fuzzy Checkpoints*
Michael Eischer, Markus Büttner, and Tobias Distler [68]

Middleware '20 *Resilient Cloud-based Replication with Low Latency*
Michael Eischer and Tobias Distler [73]

[The paper reprints have been excluded from this PDF. Please access the papers through the author's website at https://www4.cs.fau.de/~distler/ *or the respective digital libraries.]*

References

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st Symposium on Cloud Computing (SoCC '10)*. 229–240.

[2] 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. 2002. 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)*. 1–14.

[3] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

[4] Akka. 2020. https://akka.io/.

[5] Ahmad Al-Shishtawy and Vladimir Vlassov. 2013. ElastMan: Autonomic Elasticity Manager for Cloud-Based Key-Value Stores. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*. 115–116.

[6] Amazon EC2. 2020. https://aws.amazon.com/ec2/.

[7] Amazon EC2. 2020. Regions and Availability Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[8] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2007. Customizable Fault Tolerance for Wide-Area Replication. In *Proceedings of the 26th International Symposium on Reliable Distributed Systems (SRDS '07)*. 65–82.

[9] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine Replication Under Attack. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (2011), 564–577.

[10] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. 2010. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (2010), 80–93.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*. Article 30, 15 pages.

[12] Apache Hadoop. 2020. https://hadoop.apache.org/.

[13] Apache Storm. 2020. https://storm.apache.org/.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-Value Store. In *Proceedings of the 12th Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. 53–64.

[15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems* 32, 4, Article 12 (2015), 45 pages.

[16] Algirdas Avižienis. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering* 12 (1985), 1491–1501.

[17] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying Intrusion-Tolerant SCADA for the Power Grid. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*. 328–335.

[18] Luiz André Barroso and Urs Hölzle. 2007. The Case for Energy-Proportional Computing. *IEEE Computer* 40, 12 (2007), 33–37.

[19] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. *State Machine Replication in the Libra Blockchain*. Technical Report. Calibra.

[20] Johannes Behl, Tobias Distler, Florian Heisig, Rüdiger Kapitza, and Matthias Schunter. 2012. Providing Fault-Tolerant Execution of Web-service-based Workflows within Clouds. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms (CloudCP '12)*. 39–44.

[21] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2012. DQMP: A Decentralized Protocol to Enforce Global Quotas in Cloud Environments. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '12)*. 217–231.

[22] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2014. Scalable BFT for Multi-Cores: Actor-based Decomposition and Consensus-oriented Parallelization. In *Proceedings of the 10th Workshop on Hot Topics in System Dependability (HotDep '14)*. 49–54.

[23] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2015. Consensus-Oriented Parallelization: How to Earn Your First Million. In *Proceedings of the 16th Middleware Conference (Middleware '15)*. 173–184.

[24] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. 222–237.

[25] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. *Hybster – A Highly Parallelizable Protocol for Hybrid Fault-Tolerant Service Replication*. Technical Report 64440. TU Braunschweig.

[26] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani. 2019. Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*. 183–192.

[27] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. 2020. From Byzantine Replication to Blockchain: Consensus is only the Beginning. In *Proceedings of the 50th International Conference on Dependable Systems and Networks (DSN '20)*. 424–436.

[28] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage* 9, 4, Article 12 (2013), 33 pages.

[29] Alysson Bessani, Hans P. Reiser, Marko Vukolić, and Tobias Distler. 2018. Workshop on Byzantine Consensus and Resilient Blockchains (BCRB '18). In *Proceedings of the 48th International Conference on Dependable Systems and Networks Workshops (DSN-W '18)*. 121.

[30] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*. 355–362.

[31] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2008. The CRUTIAL Way of Critical Infrastructure Protection. *IEEE Security & Privacy* 6, 6 (2008), 44–51.

[32] William Lloyd Bircher and Lizy K. John. 2012. Complete System Power Estimation Using Processor Performance Events. *IEEE Trans. Comput.* 61, 4 (2012), 563–577.

[33] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing Paxos. *ACM Sigact News* 34, 1 (2003), 47–67.

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

[35] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. 335–350.

[36] Miguel Castro. 2001. *Practical Byzantine Fault Tolerance*. Ph.D. Dissertation. Massachusetts Institute of Technology.

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

[38] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems* 20, 4 (2002), 398–461.

[39] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. 2003. BASE: Using Abstraction to Improve Fault Tolerance. *ACM Transactions on Computer Systems* 21, 3 (2003), 236–269.

[40] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th Symposium on Principles of Distributed Computing (PODC '07)*. 398–407.

[41] Sanket Chintapalli, Derek Dagit, Robert Evans, Reza Farivar, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, and Boyang Peng. 2016. PaceMaker: When ZooKeeper Arteries Get Clogged in Storm Clusters. In *Proceedings of the 9th International Conference on Cloud Computing (CLOUD '16)*. 448–455.

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

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

[44] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. UpRight Cluster Services. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*. 277–290.

[45] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th International Symposium on Low Power Electronics and Design (ISPLED '10)*. 189–194.

[46] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 33–48.

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

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

[49] Christian Deyerl and Tobias Distler. 2019. In Search of a Scalable Raft-based Replication Architecture. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19)*. 1–7.

[50] Tobias Distler. 2008. Unterstützung eines schnellen verteilten Proactive Recovery unter Verwendung eines Hypervisors.

[51] Tobias Distler. 2014. *Resource-efficient Fault and Intrusion Tolerance*. Ph.D. Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg.

[52] Tobias Distler. 2015. Ressourceneffiziente Fehler- und Einbruchstoleranz. In *Ausgezeichnete Informatikdissertationen 2014*. Gesellschaft für Informatik, 71–80.

[53] Tobias Distler. 2021. Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective. *Comput. Surveys* 54, 1, Article 24 (2021), 38 pages.

[54] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. 2015. Extensible Distributed Coordination. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*. 143–158.

[55] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. 2016. Resource-efficient Byzantine Fault Tolerance. *IEEE Trans. Comput.* 65, 9 (2016), 2807–2819.

[56] Tobias Distler, Frank Fischer, Rüdiger Kapitza, and Siqi Ling. 2012. Enhancing Coordination in Cloud Infrastructures with an Extendable Coordination Service. In *Proceedings of the 1st Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management (SDMCMM '12)*. 1–6.

[57] Tobias Distler and Rüdiger Kapitza. 2011. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*. 91–105.

[58] Tobias Distler, Rüdiger Kapitza, Ivan Popov, Hans P. Reiser, and Wolfgang Schröder-Preikschat. 2011. SPARE: Replicas on Hold. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS '11)*. 407–420.

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

[60] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. 2010. State Transfer for Hypervisor-Based Proactive Recovery of Heterogeneous Replicated Services. In *Proceedings of the 5th "Sicherheit, Schutz und Zuverlässigkeit" Conference (SICHERHEIT '10)*. 61–72.

[61] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2014. Clock-RSM: Low-Latency Inter-Datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *Proceedings of the 44th International Conference on Dependable Systems Networks (DSN '14)*. 343–354.

[62] Christopher Eibel and Tobias Distler. 2015. Towards Energy-Proportional State-Machine Replication. In *Proceedings of the 14th Workshop on Adaptive and Reflective Middleware (ARM '15)*. 19–24.

[63] Christopher Eibel, Thao-Nugyen Do, Robert Meißner, and Tobias Distler. 2018. *Empya: An Energy-Aware Middleware Platform for Dynamic Applications*. Technical Report CS-2018-01. Friedrich-Alexander-Universität Erlangen-Nürnberg.

[64] Christopher Eibel, Thao-Nguyen Do, Robert Meißner, and Tobias Distler. 2018. Empya: Saving Energy in the Face of Varying Workloads. In *Proceedings of the 6th International Conference on Cloud Engineering (IC2E '18)*. 134–140.

[65] Christopher Eibel, Christian Gulden, Wolfgang Schröder-Preikschat, and Tobias Distler. 2018. Strome: Energy-Aware Data-Stream Processing. In *Proceedings of the 18th International Conference on Distributed Applications and Interoperable Systems (DAIS '18)*. 40–57.

[66] Christian Eichler, Tobias Distler, Peter Ulbrich, Peter Wägemann, and Wolfgang Schröder-Preikschat. 2018. TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses. In *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET '18)*. 6:1–6:12.

[67] Christian Eichler, Peter Wägemann, Tobias Distler, and Wolfgang Schröder-Preikschat. 2017. Demo Abstract: Tooling Support for Benchmarking Timing Analysis. In *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*. 159–160.

[68] Michael Eischer, Markus Büttner, and Tobias Distler. 2019. Deterministic Fuzzy Checkpoints. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*. 153–162.

[69] Michael Eischer and Tobias Distler. 2017. Scalable Byzantine Fault Tolerance on Heterogeneous Servers. In *Proceedings of the 13th European Dependable Computing Conference (EDCC '17)*. 34–41.

[70] Michael Eischer and Tobias Distler. 2018. Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems. In *Proceedings of the 1st Workshop on Byzantine Consensus and Resilient Blockchains (BCRB '18)*. 140–145.

[71] Michael Eischer and Tobias Distler. 2019. Efficient Checkpointing in Byzantine Fault-Tolerant Systems. In *Tagungsband des FB-SYS Herbsttreffens 2019*. 1–2.

[72] Michael Eischer and Tobias Distler. 2019. Scalable Byzantine Fault-Tolerant State-Machine Replication on Heterogeneous Servers. *Computing* 101, 2 (2019), 97–118.

[73] Michael Eischer and Tobias Distler. 2020. Resilient Cloud-based Replication with Low Latency. In *Proceedings of the 21st Middleware Conference (Middleware '20)*. 14–28.

[74] Michael Eischer, Benedikt Straßner, and Tobias Distler. 2020. Low-Latency Geo-Replicated State Machines with Guaranteed Writes. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20)*. 13:1–13:8.

[75] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. 2019. Boosting Concurrency in Parallel State Machine Replication. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 228–240.

[76] etcd. 2020. https://etcd.io/.

[77] Brad Fitzpatrick. 2004. Distributed Caching with memcached. *Linux Journal* 2004, 124 (2004), 72–74.

[78] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS '97)*. 67–72.

[79] Roy Friedman and Robbert Van Renesse. 1997. 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)*. 233–242.

[80] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2014. Analysis of Operating System Diversity for Intrusion Tolerance. *Software: Practice and Experience* 44, 6 (2014), 735–770.

[81] Miguel Garcia, Alysson Bessani, and Nuno Neves. 2019. Lazarus: Automatic Management of Diversity in BFT Systems. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 241–254.

[82] Miguel Garcia, Alysson Neves Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2011. OS Diversity for Intrusion Tolerance: Myth or Reality?. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN '11)*. 383–394.

[83] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2016. SieveQ: A Layered BFT Protection System for Critical Services. *IEEE Transactions on Dependable and Secure Computing* 15, 3 (2016), 511–525.

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

[85] Google Compute Engine. 2020. Regions and Zones. https://cloud.google.com/compute/docs/regions-zones/.

[86] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*. 568–580.

[87] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. 2014. Rex: Replication at the Speed of Multi-Core. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*. Article 11, 14 pages.

[88] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proceedings of the VLDB Endowment* 13, 6 (2020), 868–883.

[89] Gerhard Habiger, Franz J. Hauck, Johannes Köstler, and Hans P. Reiser. 2018. Resource-Efficient State-Machine Replication with Multithreading and Vertical Scaling. In *Proceedings of the 14th European Dependable Computing Conference (EDCC '18)*. 87–94.

[90] Robert B. Hagmann. 1986. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Trans. Comput.* 9 (1986), 839–843.

[91] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[92] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC '10)*. 145–158.

[93] Intel Corporation. 2015. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide.

[94] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of 41st International Conference on Dependable Systems and Networks (DSN '11)*. 245–256.

[95] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*. 295–308.

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

[97] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*. 237–250.

[98] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: An Overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*. 14–18.

[99] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems* 27, 4, Article 7 (2009), 39 pages.

[100] Ramakrishna Kotla and Mike Dahlin. 2004. High Throughput Byzantine Fault Tolerance. In *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN '04)*. 575–584.

[101] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale.. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 239–250.

[102] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability using Lazy Replication. *ACM Transactions on Computer Systems* 10, 4 (1992), 360–391.

[103] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[104] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[105] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a State Machine. *SIGACT News* 41, 1 (2010), 63–73.

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

[107] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401.

[108] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2016. Modular Composition of Coordination Services. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*. 251–264.

[109] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design*

and Implementation (OSDI '09). 1–14.

[110] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. 2018. Troxy: Transparent Access to Byzantine Fault-Tolerant Systems. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 59–70.

[111] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. 2016. SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)*. 77–88.

[112] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*. 429–444.

[113] Jun-Lin Lin and Margaret H. Dunham. 1996. Segmented Fuzzy Checkpointing for Main Memory Databases. In *Proceedings of the 11th Symposium on Applied Computing (SAC '96)*. 158–165.

[114] Linux CPUFreq. 2020. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

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

[116] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. 2016. XFT: Practical Fault Tolerance beyond Crashes. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI '16)*. 485–500.

[117] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*. 369–384.

[118] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2009. Towards Low Latency State Machine Replication for Uncivil Wide-Area Networks. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep '09)*. Article 9, 6 pages.

[119] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-Ring Paxos. In *Proceedings of the 42nd International Conference on Dependable Systems and Networks (DSN '12)*. 1–12.

[120] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: Eliminating Server Idle Power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. 205–216.

[121] Hein Meling and Leander Jehl. 2013. Tutorial Summary: Paxos Explained from Scratch. In *Proceedings of the 17th International Conference on Principles of Distributed Systems (OPODIS '13)*. 1–10.

[122] Microsoft Azure. 2020. Azure Regions. https://azure.microsoft.com/en-us/global-infrastructure/regions/.

[123] Zarko Milosevic, Martin Biely, and André Schiper. 2013. Bounded Delay in Byzantine-Tolerant State Machine Replication. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems (SRDS '13)*. 61–70.

[124] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more Consensus in Egalitarian Parliaments. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 358–372.

[125] Satoshi Nakamoto. 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Technical Report.

[126] Bruce Jay Nelson. 1981. *Remote Procedure Call*. Ph.D. Dissertation. Carnegie-Mellon University.

[127] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. 2018. On the Challenges of Building a BFT SCADA. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 163–170.

[128] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX ATC '99)*. 183–191.

[129] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*. 305–320.

[130] Zhonghong Ou, Hao Zhuang, Andrey Lukyanenko, Jukka K. Nurminen, Pan Hui, Vladimir Mazalov, and Antti Ylä-Jääski. 2013. Is the Same Instance Type Created Equal? Exploiting Heterogeneity of Public Clouds. *IEEE Transactions on Cloud Computing* 1, 2 (2013), 201–214.

[131] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems* 33, 3, Article 7 (2015), 55 pages.

[132] Christos H. Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications.

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

[134] Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (1980), 228–234.

[135] Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. 2016. On Choosing Server- or Client-Side Solutions for BFT. *Comput. Surveys* 48, 4, Article 61 (2016), 30 pages.

[136] Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, and Yair Amir. 2014. Towards a Practical Survivable Intrusion Tolerant Replication System. In *Proceedings of the 33rd International Symposium on Reliable Distributed Systems (SRDS '14)*. 242–252.

[137] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. 2015. Visigoth Fault Tolerance. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*. Article 8, 14 pages.

[138] Michel Raynal, Gérard Thia-Kime, and Mustaque Ahamad. 1997. From Serializable to Causal Transactions for Collaborative Applications. In *Proceedings of the 23rd EUROMICRO Conference (EUROMICRO '97)*. 314–321.

[139] Hans P. Reiser, Tobias Distler, and Rüdiger Kapitza. 2009. Functional Decomposition and Interactions in Hybrid Intrusion-Tolerant Systems. In *Proceedings of the 3rd Workshop on Middleware-Application Interaction (MAI '09)*. 7–12.

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

[141] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[142] Rodrigo Rodrigues, Barbara Liskov, Kathryn Chen, Moses Liskov, and David Schultz. 2012. Automatic Reconfiguration for Large-Scale Reliable Storage Systems. *IEEE Transactions on Dependable and Secure Computing* 9, 2 (2012), 146–158.

[143] Tom Roeder and Fred B. Schneider. 2010. Proactive Obfuscation. *ACM Transactions on Computer Systems* 28, 2, Article 4 (2010), 54 pages.

[144] Kenneth Salem and Hector Garcia-Molina. 1989. Checkpointing Memory-Resident Databases. In *Proceedings of the 5th International Conference on Data Engineering (ICDE '89)*. 452–462.

[145] Nuno Santos and André Schiper. 2013. Achieving High-Throughput State Machine Replication in Multi-core Systems. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS '13)*. 266–275.

[146] Rainer Schiekofer, Johannes Behl, and Tobias Distler. 2017. Agora: A Dependable High-Performance Coordination Service for Multi-Cores. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN '17)*. 333–344.

[147] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[148] João Sousa and Alysson Bessani. 2015. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *Proceedings of the 34th International Symposium on Reliable Distributed Systems (SRDS '15)*. 146–155.

[149] João Sousa, Alysson Bessani, and Marko Vukolić. 2018. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN '18)*. 51–58.

[150] Paulo Sousa, Alysson Neves Bessani, and Rafael R. Obelheiro. 2008. The FOREVER Service for Fault/Intrusion Removal. In *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS '08)*. 1–6.

[151] Doug Terry. 2013. Replicated Data Consistency Explained Through Baseball. *Commun. ACM* 56, 12 (2013), 82–89.

[152] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 309–324.

[153] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *Comput. Surveys* 47, 3, Article 42 (2015), 36 pages.

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

[155] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS '09)*. 135–144.

[156] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 12th Symposium on High-Assurance Systems Engineering (HASE '10)*. 10–19.

[157] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30.

[158] Marko Vukolić. 2015. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Proceedings of the International Workshop on Open Problems in Network Security (iNetSec '15)*. 112–125.

[159] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. 2018. Whole-System WCEC Analysis for Energy-Constrained Real-Time Systems (Artifact). *Dagstuhl Artifacts Series* (2018), 7:1–7:4.

[160] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. 2018. Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS '18)*. 24:1–24:25.

[161] Peter Wägemann, Tobias Distler, Christian Eichler, and Wolfgang Schröder-Preikschat. 2017. Benchmark Generation for Timing Analysis. In *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*. 319–330.

[162] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. 2015. Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS '15)*. 105–114.

[163] Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. 2015. GenE: A Benchmark Generator for WCET Analysis. In *Proceedings of the 15th International Workshop on Worst-Case Execution Time Analysis (WCET '15)*. 31–40.

[164] Peter Wägemann, Tobias Distler, Heiko Janker, Phillip Raffeck, and Volkmar Sieh. 2016. A Kernel for Energy-Neutral Real-Time Systems with Mixed Criticalities. In *Proceedings of the 22nd Real-Time and Embedded Technology and Applications Symposium (RTAS '16)*. 25–36.

[165] Peter Wägemann, Tobias Distler, Heiko Janker, Phillip Raffeck, Volkmar Sieh, and Wolfgang Schröder-Preikschat. 2017. Operating Energy-Neutral Real-Time Systems. *ACM Transactions on Embedded Computing Systems* 17, 1 (2017), 11:1–11:25.

[166] Peter Wägemann, Tobias Distler, Phillip Raffeck, and Wolfgang Schröder-Preikschat. 2016. Poster Abstract: Towards Code Metrics for Benchmarking Timing Analysis. In *Proceedings of the 37th Real-Time Systems Symposium (RTSS '16)*. 369.

[167] Peter Wägemann, Tobias Distler, Phillip Raffeck, and Wolfgang Schröder-Preikschat. 2016. Towards Code Metrics for Benchmarking Timing Analysis. In *Proceedings of the 37th Real-Time Systems Symposium Work-in-Progress Session (RTSS WiP '16)*. 1–4.

[168] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP '01)*. 230–243.

[169] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. 2011. ZZ and the Art of Practical BFT Execution. In *Proceedings of the 6th European Conference on Computer Conference (EuroSys '11)*. 123–138.

[170] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. 2016. Dynamo: Facebook's Data Center-Wide Power Management System. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 469–480.

[171] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP '03)*. 253–267.

[172] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 38th Symposium on Principles of Distributed Computing (PODC '19)*. 347–356.

[173] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*. 15–28.