universität ulm

Optimizing Deterministically Scheduled Fault-Tolerant Distributed Systems

Dissertation zur Erlangung des Doktorgrades **Dr.rer.nat.**
der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
der Universität Ulm

vorgelegt von
Gerhard Habiger
aus Ulm

Institut für Verteilte Systeme
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Universität Ulm

2023

OPTIMIZING DETERMINISTICALLY SCHEDULED FAULT-TOLERANT DISTRIBUTED SYSTEMS

GERHARD ROLAND MARTIN HABIGER

Institut für Verteilte Systeme
Universität Ulm

OPTIMIZING DETERMINISTICALLY SCHEDULED FAULT-TOLERANT DISTRIBUTED SYSTEMS

DISSERTATION

zur Erlangung des Doktorgrades Dr. rer. nat. der
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie der
Universität Ulm

vorgelegt von

GERHARD ROLAND MARTIN HABIGER

aus Ulm, Deutschland

2023

Amtierende Dekanin
Prof. Dr. Anke Huckauf

Gutachter
Prof. Dr.-Ing. Franz J. Hauck
Prof. Dr.-Ing. Rüdiger Kapitza

Tag der Promotion
2023-01-30

KONTAKT

Gerhard Habiger

gerhard.habiger@alumni.uni-ulm.de
ghabiger@gmail.com

ABSTRACT

Motivation

Fault-tolerant systems are ubiquitous—not only in the world of IT, but in every critical system we design and build in our natural environment. Approaches aimed at equipping systems with the resilience they need to withstand both internal and external disturbances are multitudinous. Among these approaches, State Machine Replication (SMR) stands out due to the remarkable guarantees it provides system designers with in regard to the types of failures it can tolerate. Unfortunately, however, replicating state machines comes with high resource costs, and the architectural requirements of this technique are difficult to fulfill without incurring additional, significant decreases in performance.

Problem Statement

This thesis contributes to several research endeavors addressing the performance issues SMR solutions commonly face. The primary goal of this work is the advancement of the state-of-the-art regarding solutions that equip SMR systems with the means to self-optimize during runtime.

Approaches

The thesis first identifies a common drawback of existing optimization methods that are based on deterministic multithreading, and provides an immediate solution in the form of a novel, runtime-reconfigurable scheduling algorithm. This forms the basis for the following research efforts contained in this work, which improve on prior systems using two main approaches: (i) The first approach aims at enabling SMR systems to scale vertically during runtime, which unlocks significant potential in regard to cost efficiency of these solutions. (ii) The second approach looks at the performance of SMR systems as experienced by clients, i.e., using metrics such as throughput and request latencies, and improves these by introducing several novel optimization solutions.

Background

In addition to detailing these approaches and their results, the thesis also introduces background knowledge about fault-tolerant systems, deterministic multithreading, and optimization methods that is required to understand and follow this work.

Contributions

The resulting contributions consist of multiple novel algorithms in the contexts of deterministic scheduling, deterministic distributed measurements, and performance optimization. Additionally, several proof-of-concept implementations are presented and used for thorough evaluations, showing significant advantages over unoptimized approaches or related work. Among these implementations, an architecture for vertically scalable SMR systems is presented, and a fully self-optimizing SMR setup is introduced and evaluated. Finally, the thesis details the development of a Reinforcement Learning-based framework for the creation of autonomous agents capable of self-optimizing SMR systems during runtime. Some of these novel mechanisms are orthogonal to prior work and can be combined with existing solutions to further boost the performance of deterministically multithreaded systems in the context of SMR.

ZUSAMMENFASSUNG

Motivation

Fehlertolerante Systeme sind weltweit nicht mehr aus unserer Gesellschaft wegzudenken, sichern sie doch in vielerlei Hinsicht unseren hohen Lebensstandard. Die Ansätze mit denen DesignerInnen und IngenieurInnen die diese Systeme entwerfen letztendlich Fehlertoleranz sicherstellen, sind vielfältig und Kernthema verschiedenster Forschungsdisziplinen. Unter all diesen Ansätzen stechen replizierte Zustandsautomaten (State Machine Replication, Abk. SMR) hervor, da sie ein hohes Level an Fehlertoleranz bieten, welches von den meisten anderen Mechanismen nicht erreicht werden kann. Leider wird diese hohe Fehlertoleranz jedoch durch intensiven Ressourceneinsatz erkauft, und die zahlreichen Anforderungen an die Architektur von SMR-Systemen haben oftmals starke Performanceeinbußen zur Folge.

Problemstellung

In dieser Arbeit werden mehrere Lösungsansätze verfolgt, die Performance von SMR-Systemen zu verbessern. Das Hauptziel dieser Dissertation ist, SMR-Systeme mit der Fähigkeit auszustatten, sich während der Laufzeit selbst und so autonom wie möglich zu optimieren, um auf Änderungen an Umgebungsvariablen reagieren zu können und ihre Performance im Vergleich zu unoptimierten Systemen allgemein zu steigern.

Herangehensweise

Als erstes wird diesbezüglich eine gemeinsame Schwäche aller vorheriger auf deterministischem Multithreading basierender Optimierungsmethoden identifiziert, und ein neuartiger deterministischer Schedulingalgorithmus als Lösung vorgeschlagen. Dieser bildet zugleich die Basis für die weiteren Forschungsansätze der Arbeit, welche den aktuellen Stand der Forschung hinsichtlich der folgenden Themen vorantreiben: (i) Der erste Ansatz zielt darauf ab, SMR-Systeme zu befähigen, während ihrer Laufzeit vertikal zu skalieren. Dies ermöglicht immense Kosteneinsparungen im Betrieb cloudbasierter SMR-Installationen. (ii) Im zweiten Ansatz wird die Leistung von SMR-Systemen aus Perspektive der Clients betrachtet, und die entsprechenden Metriken, wie Request-Durchsatz und -Latenzzeiten mit innovativen Optimierungsmethoden verbessert.

Hintergrundwissen

Zusätzlich zur Beschreibung dieser Forschungsansätze beinhaltet die Arbeit ausführliche Hintergrundinformationen zu den Kernthemen des deterministischen Multithreadings und der Systemoptimierung, um LeserInnen die Möglichkeit zu geben, der Arbeit auch ohne viel Vorwissen folgen zu können.

Ergebnisse und Forschungsbeiträge

Die aus den Forschungsansätzen resultierenden Ergebnisse und Beiträge der Arbeit zum aktuellen Forschungsstand bestehen aus mehreren neuartigen Algorithmen, mitunter für deterministisches Multithreading, für fehlertolerante Messungen in verteilten Systemen, und für die Leistungsoptimierung von SMR-Systemen. Zusätzlich zeigen verschiedene Proof-of-concept Implementierungen anhand ausführlicher Messungen und Evaluationen die Vorteile unserer Lösungen im Vergleich zu vorherigen oder unoptimierten Systemen. Unter diesen Prototypen findet sich zum einen eine Architektur die SMR-Systemen vertikales Skalieren zur Laufzeit ermöglicht, und zum

anderen ein umfassender, lauffähiger Prototyp eines selbstoptimierenden SMR-Systems. Zu guter Letzt wird eine Plattform für das Trainieren von Reinforcement Learning-Agenten vorgestellt, durch die es ermöglicht wird, Agenten zu kreieren die mit komplexeren Kombinationen aus Metriken und Konfigurationsparametern umgehen können, um Systeme selbsttätig zu optimieren. Einige der vorgestellten Ansätze sind zudem orthogonal zu vorherigen Arbeiten und können mit diesen kombiniert werden, um zusätzliche Leistungssteigerungen im Kontext von SMR zu erzielen.

PUBLICATIONS

Peer-reviewed conference papers, workshop papers, and journal articles by the author.

Parts of the results presented in this thesis have been published in the following publications on deterministic multithreading, optimization approaches and SMR systems. The list also contains other peer-reviewed publications of the author up to the time of submitting this thesis, including contributions to concurrent programming, distributed graph computations, and resilience in the context of IoT:

1. C. Berger, P. Eichhammer, H. P. Reiser, J. Domaschka, F. J. Hauck, and **G. Habiger**. "A Survey on Resilience in the IoT: Taxonomy, Classification, and Discussion of Resilience Mechanisms." In: *ACM Comput. Surv.* 54.7 (Sept. 2021). ISSN: 0360-0300. DOI: 10.1145/3462513.

2. J. Domaschka, C. Berger, H. P. Reiser, P. Eichhammer, F. Griesinger, J. Pietron, M. Tichy, F. J. Hauck, and **G. Habiger**. "SORRIR: A Resilient Self-organizing Middleware for IoT Applications [Position Paper]." In: *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things - M4IoT '19*. Davis, CA, USA: ACM Press, 2019, pp. 13–16. ISBN: 978-1-4503-7028-8. DOI: 10.1145/3366610.3368098.

3. B. Erb, **G. Habiger**, and F. J. Hauck. "On the Potential of Event Sourcing for Retroactive Actor-based Programming." In: *First Workshop on Programming Models and Languages for Distributed Computing*. PMLDC '16. Rome, Italy: ACM, 2016, pp. 1–5. DOI: 10.1145/2957319.2957378.

4. B. Erb, D. Meißner, **G. Habiger**, J. Pietron, and F. Kargl. "Consistent Retrospective Snapshots in Distributed Event-sourced Systems." In: *Conference on Networked Systems (NetSys'17)*. NetSys'17. GI/IEEE. Göttingen, Germany, Mar. 2017. DOI: 10.1109/NetSys.2017.7903947.

5. **G. Habiger**, F. J. Hauck, J. Köstler, and H. P. Reiser. "Resource-Efficient State-Machine Replication with Multithreading and Vertical Scaling." In: *2018 14th European Dependable Computing Conference (EDCC)*. Sept. 2018, pp. 87–94. DOI: 10.1109/EDCC.2018.00024.

6. **G. Habiger**, F. J. Hauck, H. P. Reiser, and J. Köstler. "Self-optimising Application-agnostic Multithreading for Replicated State Machines." In: *2020 International Symposium on Reliable Distributed Systems (SRDS)*. 2020, pp. 165–174. DOI: 10.1109/SRDS51746.2020.00024.

7. F. J. Hauck, **G. Habiger**, and J. Domaschka. "UDS: A Novel and Flexible Scheduling Algorithm for Deterministic Multithreading." In: *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2016, pp. 177–186. DOI: 10.1109/SRDS.2016.030.

8. J. Köstler, H. P. Reiser, **G. Habiger**, and F. J. Hauck. "SmartStream: Towards Byzantine Resilient Data Streaming." In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. Virtual Event Republic of Korea: ACM, Mar. 2021, pp. 213–222. ISBN: 978-1-4503-8104-8. DOI: 10.1145/3412841.3441904.

9. J. Köstler, H. P. Reiser, **G. Habiger**, and F. J. Hauck. "SmartStream: Towards Efficient Byzantine Resilient Data Streaming through Speculation and Sharding." In: *ACM SIGAPP Applied Computing Review* 21.3 (Sept. 2021), pp. 19–32. ISSN: 1559-6915, 1931-0161. DOI: 10.1145/3493499.3493501.

CONTENTS

I

FOUNDATIONS

1 INTRODUCTION

As researchers and engineers, we design, build, and test systems—systems designed to prove or disprove our theories, systems that aid us in our daily work, or just simply systems for helping us with the discovery of the undiscovered. For any system to be useful, it has to function, i.e., provide the service it was designed to fulfill, preferably without interruption or downtime. This admittedly rather obvious insight contains the inherent requirement that a system be dependable, i.e., resilient against perturbances, regardless of whether they originate from outside the system or from within the system itself.

Dependability & Resilience

Nature has produced a manifold of resilient systems before humans ever started thinking and innovating their way out of caves. Fish lay thousands of eggs, although this requires more energy than producing just one egg would, simply so at least a few offspring can survive and fulfill the function of procreation. Biology favors diversity in any ecosystem, as it hardens the system against a wide variety of threats [101]. Similarly, human engineers have always liked to include safety margins and measures against potential disasters, especially in critical systems designed to support our civilization. By doing so, they too can create more impervious systems. As such, it can be argued that dependability and resilience should be core properties of almost any system—apart maybe only from those created for purely recreational or otherwise truly non-critical purposes.

Fault Tolerance

To achieve resilience, or—as the US Department of Homeland Security Risk Steering Committee describes it—the *ability to adapt to changing conditions and prepare for, withstand, and rapidly recover from disruption* [105], we generally want systems to have the ability to continue operating even when faced with faults. These faults can originate from within the system itself, or be introduced into the system by outside influence, but always take effect in components of the system. The main concept of tolerating faults within a system, regardless of their origin, is aptly named *fault tolerance* (FT), and lies at the heart of most strategies aimed at hardening systems against failures[1].

Even though these motivations and thoughts may seem a bit diffuse, they are in fact non-trivial: FT techniques always add to the total cost of a system (even if only at design time) and are usually resource intensive. However, the benefits that arise from their inclusion are a good motivation and justification. Having these carte-blanche reasons for adding resilience to systems—like the ones motivated in first two paragraphs of this introduction—simplifies our reasoning for expending more energy when building or running resilient systems. Nonetheless, this high energy expenditure is also, of course, reason enough to look into ways to minimize these added costs, to overall increase the attractiveness of employing FT techniques further.

1 There are clear terminological differences between *faults* and *failures*, the details of which are explained in Section 2.1.

1.1 OBJECTIVES AND CONTRIBUTIONS

Thesis Goals

The core goal of this thesis is to provide several optimizations to specific FT techniques under the mantle of the *OptSCORE* project, funded by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), so as to contribute to the state-of-the-art of building critical systems for general use. In particular, this thesis significantly contributes to the field of research that focuses on building Byzantine fault-tolerant distributed systems, which are optimized by way of introducing concurrency through deterministic multithreading. Of course, in order to convey what this means and to comprehensibly talk about our research in the pursuit of this as yet relatively vague goal, common ground and basic understanding about fault-tolerant distributed systems have to be established first. Afterwards, the research goals of this thesis will be more clearly defined using the established terminology and background knowledge. Nonetheless, for readers familiar with these topics, we will succinctly summarize our three main contributions in the following paragraphs, giving forward references to definitions where applicable.

1.1.1 *Developing Runtime-Configurable Deterministic Multithreading*

First Contribution

In the field of deterministic multithreading, especially in the context of state machine-replicated (SMR) systems, several prior works have established a common system model and working baselines. One of the central approaches utilizes deterministic schedulers to enable multithreading while preserving determinism. However, one of the major lacking features of these existing solutions is adaptability, in the sense that the system can be reconfigured or reconfigure itself during runtime, e.g., to react to changes in the system's environment or behavior. To remedy this situation, we first contributed to the initial development and publication of a novel runtime-configurable deterministic scheduling algorithm. Afterwards we implemented and tested this algorithm within an event-based simulation to gather results on the scheduler's effects, e.g., on system performance and efficiency metrics. This contribution in itself is a valuable addition to the field of deterministic multithreading, as it constitutes a condensation of prior works into a single algorithm capable of emulating not only the behavior of these previous algorithms, but opens up a larger configuration space that allows for exploration of novel scheduling strategies in the context of replicated systems. Lastly, several bug fixes and optimizations to the algorithm were added over the years, since integrating the scheduler into various systems was key to enabling many of the further research efforts presented in this thesis.

1.1.2 *Improving Resource-Efficiency for SMR Systems in Cloud Environments*

Second Contribution

SMR systems are frequently deployed to virtualized cloud environments to in-

crease their resilience against hardware failures[2]. However, due to their nature, these systems usually require the strict provisioning of hardware resources in order to meet performance targets under variable load conditions, e.g., while the number of connected clients fluctuates during a day. This strict provisioning wastes resources and costs money. We proposed an architecture for dynamical vertical rescaling of replicated state machines during runtime to adjust the provided hardware resources on-the-fly. To demonstrate the feasibility of this approach, we then implemented a prototype system for measuring the impact of vertical scaling decisions on system efficiency. An evaluation of the approach based on this prototype was then performed, revealing that significant reductions in operating costs of SMR systems are possible, thereby fitting into our general motivation of optimizing these systems.

1.1.3 *Enabling Performance Self-Optimization of SMR Systems*

Despite recent advancements in the field of SMR performance optimization, one feature that has persistently been a roadblock against further improvements is the efficient concurrent execution of requests. Firstly, only the most cutting-edge, prototypical research implementations of replicated state machines allow for concurrent execution, while the few actually available options to the public, i.e., those outside of the labs of research groups specializing in optimizing SMR systems, still utilize single-threading in order to preserve determinism. Secondly, among those few proposed solutions that do parallelize the execution stage of state machines, none can react to changes in the environment, e.g., fluctuating load, diverse application profiles, or switching of the replicated applications. Lastly, in order to improve the runtime-performance of a deterministically scheduled system, at least one deterministic metric is required to measure whether any employed optimization actions have a beneficial or detrimental effect. However, deterministically measuring a Byzantine fault-tolerant distributed system is no easy feat, as any measurement mechanism has to be resilient against any Byzantine influence, too. Our final contributions to solving these problems can be broken down as follows:

Deterministic Byzantine Fault-Tolerant Measurements

Third Contribution

First, we developed a way to deterministically measure current system load, and analyzed this algorithm's behavior under different failure models. Afterwards, we implemented the algorithm, and evaluated its real-world characteristics using an SMR benchmarking setup. We then integrated our previously mentioned runtime-reconfigurable scheduling solution (Contribution #1) with this prototype SMR system to enable concurrent execution of requests. This architecture itself is a significant contribution to the field because it allows for the development of algorithms that require a deterministic decision basis within a distributed, BFT SMR system. Additionally, the architecture

2 Chapter 2 will explain the background behind SMR from the ground up.

allowed us to research and evaluate two optimization strategies for enabling self-optimization in SMR in the following contributions.

Rule-Based Self-Optimization of SMR Systems

Fourth Contribution

To this end, we developed a rule-based self-optimization algorithm using inferred knowledge about our scheduler's behavior gathered during its initial implementation and evaluation stages while working towards Contribution #1. We implemented this algorithm in our research platform, and evaluated its behavior in scenarios with variable load. With this setup, we demonstrated that self-optimizing SMR deployments can experience considerable performance improvements utilizing reconfigurable scheduling during runtime. This again ties back to our initial motivations.

Agent-Based Self-Optimization of SMR Systems

Fifth Contribution

Finally, to generalize this previous approach and enable our system to react to a broader set of circumstances, we investigated the latest state-of-the-art in Reinforcement Learning (RL) techniques, with the goal of applying their innate potential to handle complex systems to this optimization problem. With this in mind, we transformed our environment to be suitable for the application of RL methods, and set out to training a neural network-based RL agent, which would in the best case take over parameter adjustments in our runtime-configurable prototype system. The main contribution of this research was the development of this fully working RL training platform based on cutting-edge approaches, alongside a collection of valuable insights into the feasibility of this approach in general.

1.2 ROADMAP

The remainder of Part I introduces the foundations and background knowledge on which most of the remaining thesis is built. It is aimed at readers who are familiar with general concepts of computer science and who have at least dipped their feet into the realm of FT before. Part II introduces our previously mentioned novel deterministic scheduling solution, called Unified Deterministic Scheduling/-er (UDS). It is one of the core mechanisms on which the optimizations developed during our research are based. In Part III we detail our contribution that improves the efficiency of deterministically scheduled fault-tolerant systems in cloud environments. This approach works especially well in regard to hardware resource utilization, resulting in significantly lower operating costs of such systems. Part IV, finally, introduces an additional selection of related work pertaining to performance optimization of SMR systems, and details our efforts into improving the performance of these systems by way of dynamically reconfiguring their parameters during runtime. To close, Part V summarizes our main results and contributions, and concludes by leaving

the reader with an outlook on open future work in this interesting subfield of distributed systems research.

Reading Paths

Depending on one's available time or interest in certain topics, we recommend alternative reading paths for readers only interested in specific themes.

EXECUTIVE SUMMARY This selection provides the most condensed summary of the contributions of this thesis.

Relevant chapters: Chapters 5, 7, 8, 11, 16, and 17.

UDS This chapter selection covers the development, specification, implementation, and debugging of UDS. It also includes complementary aspects such as background information on deterministic multithreading. Afterwards, it skips over large parts of the thesis directly to its closing chapter, where the main ways in which UDS was used in our contributions is summarized a last time.

Relevant chapters: Chapters 3, 6 to 8, and 17.

RESOURCE EFFICIENCY The next selection covers the chapters directly involved in our discussions about resource efficiency optimizations, including two of the three background chapters to prepare the terminology used while later describing our approaches.

Relevant chapters: Chapters 2, 3, 9 to 11, and 17.

PERFORMANCE OPTIMIZATION As one of the main goals of this thesis, if you are interested mainly in a summary of our performance optimization endeavors, we recommend reading the following chapters, including the initial chapters introducing foundational knowledge.

Relevant chapters: Chapters 2 to 4 and 12 to 17.

FAULT-TOLERANT DISTRIBUTED SYSTEMS

As motivated in Chapter 1, this thesis aims to improve the performance of specific FT mechanisms in a certain class of systems. This chapter aims to clarify exactly what kind of specific mechanisms and systems we are talking about, while also giving background information about their inner workings.

2

2.1 BACKGROUND

Distributed Systems

The very first important distinction to be made is already hidden in the heading of this chapter: We are exclusively interested in distributed systems and their unique properties. In fact, many FT techniques rely on a system being distributed, or, put differently, many systems can only be made reliable by transforming them to distributed systems.

Distributed systems in the context of computer science can—a little cheekily—be defined as follows:

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." [98]

Terminology

We will shortly introduce a slightly more technical definition of distributed systems, but chose this quote because it lets us segue to a fulfillment of the promise made in the introduction: That we briefly talk about terminology.

In the seminal work of Avižienis et al. from 2004, the most common terms used when describing dependable systems are succinctly differentiated [15]:

- *Failure*: A deviation from the state where the system correctly delivers its provided service, i.e., it does not fulfill its functional specification anymore. This is visible to the system's environment, so in our case, to the system's clients.
- *Error*: Systems internally transition through states in order to provide their service to clients; an error, then, is a deviation from any correct state, which can — but does not have to — lead to an externally visible failure.
- *Fault*: An underlying cause of a potential error. Faults can be internal or external, and examples include, but are not limited to, hardware faults (e.g., physical deterioration), human-made faults (e.g., a missing interaction, forgotten by a system's maintainer), development faults (e.g., erroneous design of an internal component), and many more. For an external fault to cause an error within a system, a previously existing internal fault (a *vulnerability*) has to exist in the system.

Note that not all faults have to lead to errors, and not all errors lead to failures. The goal of FT mechanisms is of course to tolerate as best as possible any

occurring faults, so that the system can transparently mask errors and never experiences a failure (or, in other words, FT tries to equip a system with the means to ignore occurring errors so that clients of the system never notice a failure).

Further, the way a system behaves in case a failure occurs can be described by differentiating between *failure models*:

FAIL-STOP The simplest form of failure behavior; a failed system in this case fails permanently, i.e., it *crashes*, and can be detected as being in a crashed state by other systems. Also called *crash-stop*.

FAIL-SILENT A system crashes, i.e., stops delivering its service, but other systems may have trouble detecting this failure and cannot access the crashed system's last state.

CRASH-RECOVERY A system may intermittently fail to deliver messages required for service provision, but can recover from this failure.

BYZANTINE A failed system may deliver arbitrarily erroneous results — within the bounds of computational feasibility —, including ones with malicious intent designed to harm other systems, or showing any of the symptoms of the previous failure models.

This brief terminological introduction shall suffice for the purposes of this thesis. For further reading, see for example [75], [15], or [10].

For an alternative definition of distributed systems, maybe a little less humorous and more apt than the previous one, we could follow van Steen and Tanenbaum's view:

> "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system." [108]

Client-Server Architecture

This is a very broad definition, hence whenever we talk about distributed systems in the context of this thesis, we generally mean individual physical or virtual machines connected via a local or wide area network, which collectively aim to provide a service to ephemerally connected clients, i.e., employing a classical client-server architecture.

Several approaches to introduce FT exist for this generic architecture. In the following sections, we will briefly introduce the main concepts using an example application, to give context on FT mechanisms and slowly transition to the main FT mechanism this thesis is aiming to improve.

2.2 TIME-BASED, DATA-BASED, AND OPERATION-BASED FT

Let us assume we are trying to build an FT distributed system of the sort defined above, i.e., a collection of network-connected machines, which accept queries from clients wanting to make use of the provided service.

To start small, a single machine with appropriate software providing service functionality shall serve as version v_0 of our (as yet undistributed) system. In

this case, any fault in the machine whatsoever will lead to system failure, as a single machine without any further FT measures will have no way to mask occurring errors to clients. Even worse, when system v_0 fails, it will likely fail for good, since we have no way of restoring it to a correct state — internal faults may have led to irreversible corruption of required data. Hence, v_0 is a *fail-stop* (or possibly even a *fail-silent*) system.

From Fail-Stop to Crash-Recovery

Time-Based FT

For a version v_1, we can therefore employ *time-based* FT methods, which save a system's internal state required for service restoration to any kind of persistent medium (which is in itself another fault-tolerant system, but we shall ignore this digression for now). For example, periodic checkpointing of the system's application data to a hard disk which can survive power outages, or event sourcing-based application design which continuously streams events to a network storage both constitute time-based FT methods. The latter would even turn our system v_1 into a proper distributed system. With v_1 we have elevated our system to the *crash-recovery* failure model, as after a crash we can now restore the system's state and resume service[1]. Note that the primary mechanism which granted us FT was the addition of a level of redundancy, in this case in the temporal domain. We will encounter this paradigm multiple times in the next few sections, since redundancy of any form is the foundational enabling concept of FT. This is of course also a very general overview, skipping over many of the details required to actually make time-based FT mechanisms work properly, but shall suffice for our current discussion of techniques.

Avoiding Service Interruptions

Data-Based FT

A problem we still have with v_1 is that whenever the system experiences a failure—regardless of the actual underlying *fault* → *error* → *failure* chain—, all current clients of the system will be affected by this failure. In other words, service is interrupted for a period of time while the system is restored to an operational state. It would be beneficial if we had a way to reduce the impact of failures, i.e., continue service delivery for at least a percentage of clients. For example, let us assume without loss of generality that our system is meant to provide clients with simple access to a set of data[2]. If we were to partition the data, i.e., separate it into multiple distinct subsets, and then distribute these subsets to their own machines, any failure of a single machine would only affect clients trying to access the subset of data that was provided by this particular machine. However, this would not be a good choice if our goal is to avoid service interruptions, as those clients trying to access data on a crashed machine still experience a full service outage. Alternatively, we could replicate the entire dataset to multiple machines, which could transparently tolerate the failure of one or more of these, but only if we pay close attention to how write

1 This restoration can happen either manually, or even automatically by some external system
2 like e.g., a REST-like system would [94]

access to any data within our dataset are distributed among the replicas. Note how in the case where we partitioned the data onto multiple machines without replication, we were missing redundancy and did not gain any FT, whereas in the latter case the replication of data constituted redundancy in the data- or storage-domain.

If our service is truly meant to only provide read- and write-access to a set of data, a step up from the simple replication of data would be the usage of error correction codes such as erasure coding [110]. In erasure coding, data is not simply partitioned or distributed fully, but is first separated into n data *chunks*. Now, a particular code and parameter k can be chosen that signifies how many missing chunks of data we can tolerate and still be able to reconstruct the original dataset. Thanks to the mathematical principles behind error codes, which we will not dive into in this thesis, choosing a proper algorithm and n would yield (n, k)-code, where in addition to k data chunks, only k additional chunks need to be determined by the erasure code and saved to storage to be able to restore the full dataset. This works when less than k chunks fail, and it works regardless of which chunks out of all $n + k$ chunks get lost, and would be just one example of a *data-based* FT mechanism. This new version v_2 would allow us to completely mask the failure of a certain number k out of n total machines within our system, so clients would not have to endure any service interruption as long as only a maximum of k machines fail simultaneously[3].

It is important to note here that even though we make it sound easy to add these measures to a system, in real environments, these are in and of themselves rather complicated problems, which require careful engineering in order for a system to properly function and fulfill its FT guarantees. For example, simply "adding erasure coding" also requires that all writes be properly ordered on all shards, lest we run the risk of garbling our redundancy data. However, for the sake of this discussion, where our goal is a readily understandable and digestible description of FT mechanisms, we sometimes take the liberty of omitting implementation details of mechanisms which are not required to be fully understood for the purposes of this thesis.

What About More Complex Systems?

Operation-Based FT

The level of FT we have achieved with v_2 is already rather impressive, but still leaves some things to be desired. If, for example, the system needs to do more than just provide access to a set of data (e.g., perform a series of complicated calculations on the entirety of our dataset before a single data point can be updated or read), it would be difficult or even impossible to get sharding to work. To provide k-fault-tolerance for a more complex system, operation-based FT mechanisms can be used. These add redundancy at the level of operations, i.e., each operation in the system is executed on multiple machines.

To achieve this, instead of sharding or erasure coding the data, we could try to model our entire system as a deterministic finite state machine (DFSM) and replicate this entire DFSM onto multiple machines. If we initialize all ma-

3 A system with this property is said to be *k-fault-tolerant*.

chines with the same initial state and could find a way to distribute all client queries to these machines in exactly the same total order, all of them would deterministically (hence the name) execute exactly the same operations, reaching the same result for each client query. In other words, if we feed client queries into all of our DFSM sequentially and in the same order, every machine will individually execute the exact same steps as all the other machines, and all of them will reach the same outcome. As a first, very simple solution for distributing queries to all clients in the same order, we could employ a sequencer. This component would receive all queries from all clients and sequentialize them before forwarding them to the DFSM. The idea of replicating state machines comes with a high resource cost, as we need multiple machines capable of executing the entire service plus a component for ordering requests. The immediate benefit of this approach is that since the DFSM are independent of each other, it would suffice for just one machine to fully execute a client query in order for this query to be answered successfully[4].

We would now *almost* be able to provide k-fault-tolerance for any service we can model as a DFSM. The last issue lies in the sequencer component we introduced to ensure the same total order of client requests at each replica. This component itself is currently not protected by any of the FT mechanisms we introduced so far, and it's quite clear that it would be impossible to harden it without employing some kind of distribution. We then immediately arrive at the conundrum that now the sequencer itself would have to be replicated, requiring a total order of incoming requests, and so on. In other words, the sequencer constitutes a single point of failure and ruins the entire premise of trying to heave the system into a new class of FT. Nevertheless, let us call this version of our system v_3, and we shall now see how we can improve it further.

Consensus Algorithms

What we require is an atomic distribution of requests (i.e., messages) in a total order to multiple participants, which is aptly named *atomic multicast*. In order to solve the problem of atomically multicasting messages, machines can try to agree on the first message they should receive, before agreeing on the second message, and so on, until they have agreed on all messages that should be received. This effectively establishes an order. The process of agreeing on values is called *consensus*, and it has been shown that atomic multicast is solved if consensus is solved [27].

Hence, current systems employ complex distributed consensus algorithms to vote on an agreed upon order of requests, thereby fully distributing the process of ordering requests and removing the remaining single point of failure in system v_3. These algorithms, originating from the first versions of the Paxos algorithm [58], have been supplying their own community of researchers with a steady stream of optimization problems for more than 30 years now, and have been developed to a point where it is difficult to reason about them without specialized tools [24]. For the purposes of this thesis, it suffices to introduce the main concepts behind (and requirements of) popular consensus algorithms.

4 This is only valid if we assume a crash-stop model for individual machines in our system, and if clients accept the first answer they receive as correct. It is also a little unusual to think about replicated state machines in this way. More on this follows in the next subsection.

In the most general sense, all consensus algorithms are based on similar ideas. Participants will go through multiple voting rounds or phases, exchanging messages containing votes and meta-information about globally agreed upon variables, until a certain outcome has been reached by a defined *quorum* of participating nodes. Typically, majority quorums are used, i.e., if a majority of nodes agrees on an outcome, this outcome is *agreed* and will henceforth never change. We can intuitively reason about the exact number of nodes required to guarantee our pursued k-fault-tolerance as follows: In a fail-stop or crash-recovery-level system, the smallest possible majority required to reach agreement over a value is two, since one machine cannot be a majority compared to anything else. If we need at least two machines to agree on a value, and machines can fail-stop or temporarily crash, we need at least three machines to tolerate a possible failure of one of them (so that two are still left to vote on a value). Therefore, to generalize this, we need at least $2k + 1$ machines for the crash-recovery failure model, where k is the maximum number of machines that can fail before the entire system fails[5].

State Machine Replication

The approach of replicating deterministic finite state machines (DFSM) and using a distributed consensus algorithm to agree on a total order of incoming requests is commonly known as *State Machine Replication*, and has been an active area of research for more than three decades ([59], [72], [16]).

This thesis contributes to a subfield of this area of research, so while we have been laying some important foundations so far, we shall continue our brief introduction to further narrow our focus and pinpoint where exactly this thesis improves on the state of the art. For future reference however, this current system, being capable of true k-fault-tolerance, will be our fourth version, v_4.

The Highest Rung of the Ladder

Systems v_2, v_3, and v_4 have allowed us to transparently mask many faults in our system and increased the complexity of systems we can harden thusly. However, they did not yet manage to climb to the highest rung of the failure model ladder and claim *Byzantine* FT. As a reminder, a system failing in a Byzantine way could exhibit arbitrarily erroneous behavior to clients, including the sending of wrong results. We can assume for a moment a system v_4 with exactly three state machines, and a fault occurs which corrupts some internal part in one of the machines in a way that this machine sends out different consensus messages to the other two machines[6]. It is then entirely possible these other machines can become confused about the order of messages they are voting on, and no majority quorum can be reached.

Byzantine Fault Tolerance

5 Slightly confusingly, this parameter k is nowadays commonly an f, even though we call it k-fault-tolerance, and despite the fact that historically seminal papers used m [60]; so from here on out we say $2f+1$ instead of $2k+1$ (or similar). We just need to remember that k signifies how many machines (or subsystems) can fail in our current failure model before the FT mechanism breaks and the system fails.

6 This could be the result of a software development error or an active attack by a malicious party. The nature of the fault is irrelevant to this discussion, as long as it occurs.

In other words, it can be shown that in order to tolerate Byzantine faults, a quorum size of at least $2f + 1$ is required; so if f machines can fail in a Byzantine manner, we need at least $f + 2f + 1 = 3f + 1$ total machines to tolerate this [60]. There are additional requirements and pitfalls to making SMR systems Byzantine fault-tolerant when actually implementing them, but for now, we are content to know that for a final system v_5, we require at least $3f + 1$ machines. With this enormous overhead we buy the highest level of FT for our system—even a Byzantine fault in f subsystems cannot stop it from delivering its service to clients. We have thus finally managed to transparently mask arbitrarily complicated faults in our system, albeit at a rather high resource cost.

2.3 SUMMARY

In the last sections, we journeyed from a single, easily failing system to a complex system of subsystems with assumptions about failure models and sophisticated measures against faults of almost any kind, all to transparently mask internal faults to the outside world and continue service delivery. This goal, first motivated in the very beginning of this chapter, is an important one to reach, especially for critical systems. In addition, for all future discussions in this thesis, we shall assume the following[7]: The systems we are trying to optimize are state machine-replicated and seek to be Byzantine fault-tolerant, hence they are of distributed nature and employ multiple networked, communicating, and cooperating machines. Each machine is supplied with the same client queries in the same total order, executing a DFSM representing our service.

While we have now introduced the ideas and motivations behind such an architecture, we have not yet detailed how determinism can be achieved, especially when trying to optimize for performance. The next chapter will detail solutions to this problem.

7 A proper system model will be introduced in Section 3.4.

DETERMINISTIC SCHEDULING

3

For our previous journey from system v_0 to v_5, we at some point started to assume deterministically executing finite state machines in order to work our way up to Byzantine fault tolerance. These, as a reminder, require mainly two basic assumptions: (i) Their input is totally ordered, which — as we have seen — is usually achieved by employing consensus algorithms, and (ii) they execute their provided service deterministically, i.e., for a given initial state and execution (initiated by a client request) they all reach the same result[1]. While we have talked about the former requirement in the previous chapter, the latter has yet to be explored.

3.1 BACKGROUND

Achieving deterministic execution might seem trivial at first, since at their core, our current digital computers are already highly deterministic machines. For example, one can have them add the same two numbers together several billion times a second, and barring any hardware faults or disturbances due to cosmic rays[2], they will always reach the same result.

Multi-Threading

However, even though science has blessed us with nearly 50 years of biennially doubling transistor counts in microchips —a fact that has been astutely observed by the rather well-known *Moore's Law*— alongside proportional performance and efficiency gains, the recent years have seen a slowing or even stagnation of raw performance gains in single-threaded workloads [106], as chip designers are fighting the so-called *Power Wall* [89]. Thus, since the mid-2000s, most of the new transistors gained by improved manufacturing processes have been used to add more and more cores to CPUs. In order to fully make use of these multicore CPUs, workloads had to be increasingly parallelized. Unfortunately, since in parallel programs multiple processes are cooperating to solve a problem, synchronization and timing become critical, and programs might not execute deterministically anymore, i.e., multiple executions of the same parallelized program, given the same inputs, might not yield the same results.

Take for example two processes `A` and `B`, which execute a simple program that updates a shared variable. Without any synchronization measures, the order in which process `A` and `B` are allowed to access the variable is arbitrarily decided at runtime by the operating system's scheduler, depending on ephemeral factors like e.g., other processes in the system or wall clock time. If the operations carried out by processes `A` and `B` are not commutative, then multiple executions (e.g., on replicated state machines) can result in differing states af-

1 No matter which machine (or how often a single machine) executes this request, when starting from the same state.

2 Faults like these fall into a category called "soft errors".

ter the executions finish, breaking the determinism required for State Machine Replication as introduced in Section 2.2.

To prevent this, while at the same time keeping the performance advantages of parallelism, we have to find a way to make indeterministic executions of parallelized programs deterministic.

Deterministic Multithreading

This is where *Deterministic Multithreading* enters the scene. Specifically, we focus on a technique called deterministic scheduling to achieve deterministic multithreading As the name implies, with deterministic scheduling the two processes A and B would always execute at least those of their actions that affect determinism in the same order, guaranteeing the same result over different executions.

In the following sections, we will introduce how deterministic schedulers work by showing related work and established deterministic scheduling algorithms. Afterwards, we will have compiled the basic knowledge required for presenting the first contribution of this thesis in Chapters 6 and 7: A new and flexible deterministic scheduling solution, which can unlock further optimization potential in SMR systems.

3.2 DETERMINISM FUNDAMENTALS AND TERMINOLOGY

A brief summary of the most important terminology will help with the following discussions of scheduling algorithms.

Determinism

First, let us properly define *determinism* for the purposes of our research, in the most general way possible: A system shall at any moment in time consist of its current state, which shall encompass all of its internal variables required for full provision of the service it was designed for, and a set of possible transitions from that state. The transition that is chosen to get to the next state (i.e., configuration of variables) depends only on the input received while at the previous state. The next state thereby depends only on the previous state and received input. It follows that if such a system starts at the same initial state and receives the same inputs in the same order, it will always reach exactly the same resulting state.

Note that the system can include non-deterministic state and operations, but their status is irrelevant to the provision of the system's service. In other words, when setting up an exact replica of the system with the goal of providing the same service, only the parts that have to be copied over to achieve this are said to be deterministic, while non-deterministic state is everything that can be randomized without changing the system's behavior as observed from the outside. The following section will specify less abstractly what the system model for our research considers states and inputs.

Deterministic Scheduling

The internal modifications that happen in a system to transition from a given state when receiving input can either be applied sequentially or concurrently. In a sequential system, all updates to internal state variables happen in a defined order by default. However, in a concurrent system, we assume that the order of modifications are governed by external random influences. We have to make sure these updates also happen in a defined order, lest we risk

differences between two instances of the same system applying the same input to the same state, thereby violating our definition of determinism. We shall call the mechanism ensuring this order a *deterministic scheduler*.

Piecewise Determinism

If a system contains indeterministic events, but it is possible to identify all of these events and control for all pertaining variables in such a way that the re-execution of such events would happen in exactly the same way, the system is said to be *piecewise deterministic* (PWD) [39]. In other words, if we manage to find all possible influences that govern the outcome of each indeterministic event in a system, its execution can be made deterministic, and the system is PWD. An example of this would be a system with multiple threads that randomly interleave in different executions, but we take control of these interleavings by way of introducing a control component such as a deterministic scheduler. Iff the system is PWD, this component, i.e., scheduler, can make sure the executions of such a system are always deterministic.

Full Determinism

Considering we aim to optimize state machine-replicated systems, another useful definition is that of *full determinism* [34]: Systems which can be made deterministic without requiring any further communication between individual instances after the input to the systems has been totally ordered, are fully deterministic. In other words, a system is fully deterministic if a scheduler can ensure determinism while only relying on totally ordered input, and no other information from other replicated instances (such as, for example, previous execution logs obtained from a leader) is required.

Weak vs. Strong Determinism

Lastly, we classify a system as being *weakly deterministic*, if its determinism depends on correct interleaving of accesses to certain primitives, like mutexes [68]. By this, we mean to say that the system has to make use of special primitives—and do so *correctly*—for any operation that could endanger determinism, so as to ensure that the scheduler can fulfill its main goal. A common example of such primitives are locks, which can be acquired and released by threads modifying shared data. This is usually done in order to prevent data race conditions, but as will be shown in Section 6.1.2, we can also achieve determinism by properly ordering access to these primitives—as long as the application is weakly deterministic, i.e., correctly using these primitives. By specifying that a system has to utilize e.g., mutexes *correctly*, we mean to say that regardless of musings about determinism, an application has no inherent race conditions because of logical implementation errors. An example of incorrect usage would be an application with an implementation bug that forgets to protect a shared data resource with a mutex, allowing multiple threads to modify the data at the same time, inviting race conditions and all their inherent troubles. Such a system could not be considered weakly deterministic, as this source of indeterminism could not be removed by a deterministic scheduler which bases its operation on the manipulation of e.g., mutexes. In contrast to weak determinism, *strongly deterministic* systems take measures to transform any regular execution into a deterministic one, for example by ensuring that all memory accesses, not only those protected by mutexes, are deterministically ordered [68]. These systems can do so by, e.g., modifying code

at compile time, or by injecting layers between runtimes and the OS, which can intercept possibly indeterministic system calls.

With these terms defined, let us investigate prior work surrounding these terms, to get a feeling for the current state-of-the-art regarding deterministic multithreading, especially in the context of state machine replication. Finally, at the end of this chapter, we will specify our system model, on which the remaining research presented in this thesis will be based.

3.3 PREVIOUS WORK

Several prior publications have devised deterministic scheduling solutions aimed at introducing determinism to replicated state machines. We will look at a selection of these previous works, to pinpoint both similarities and differences between them and our following contributions. In a later chapter (cf. Section 7.3), we will also revisit some of these publications in greater detail, when we discuss how our novel scheduling solution, i.e., our first major research contribution, can imitate their behavior.

Deterministic Scheduling Algorithms

MTRDS

One of the very first deterministic scheduling solutions for replicated systems, *Multithreaded Deterministic Scheduling* (MTRDS), was presented in 2000 [49]. It is based on a transactional model, wherein replicas can abort transactions, and supports long-running replica threads for multiple client requests per transaction. This is in contrast to the singular RPC calls we assume in our system model. The remaining model is very similar to ours as defined below (Section 3.4).

With MTRDS, once a thread for handling a client-interaction is created, it can either be *blocked*, e.g., when waiting on a mutex, or be *ready* to run. At a high level, MTRDS works by carefully separating system-level and client-level requests and defining the order in which the deterministic scheduler may transfer control (i.e., continue execution) to any *ready* thread. Control is alternated between the deterministic scheduler and the next *ready* thread chosen to run by the scheduler. The order in which *ready* threads are chosen is governed only by the order of the single queue they arrive at (called the *external* queue). The scheduler deterministically maintains multiple *internal* queues, which it populates by taking requests out of the *external* queue, to manage multiple services running on a replica. Threads are then first taken from the internal queues whenever there is at least one such queue that is not empty. Translated to the previously introduced system model used in this thesis, only one such internal queue would exist. Note that MTRDS toggles control between the scheduler thread and the actual request threads, but only one thread is running at a time. It does therefore not achieve actual parallelism.

LSA

A different approach is taken by an algorithm called *loose synchronization algorithm* (LSA), published in 2002 by Basile et al. [18]. LSA does not explicitly preclude replicas from communicating after the total ordering of messages via

consensus has been achieved[3]. In LSA, a single *leader*-replica executes client requests and records the order of mutex acquisitions between threads. All other replicas are *follower*-replicas and receive the recorded mutex acquisition order via messages sent from the leader. They then assign mutexes to threads in exactly the same order. Therefore, to achieve determinism, LSA requires additional communication overhead between replicas even after a total order of requests has been established, foregoing full determinism. In comparison to MTRDS, however, LSA can achieve true parallelism.

PDS

Shortly thereafter, the authors of LSA proposed an improvement over LSA called *Preemptive Deterministic Scheduling Algorithm* (PDS) [17]. It introduces the concept of *rounds* to deterministic scheduling, which will be motivated and explained in greater detail in Section 6.1.1. Nonetheless, a brief introduction will be given here, to explain how PDS works.

Scheduling Rounds

A scheduling round is a deterministically chosen collection of threads that are allowed to acquire mutexes in a specific, pre-determined order. All threads trying to acquire mutexes without being part of a round are blocked. These threads then have to wait until the round is over, and they get the chance to be part of the next scheduling round. In the publication, PDS is separated into PDS-1 and PDS-2. In PDS-1, threads can only acquire one mutex per round, while in PDS-2 they are allowed to acquire up to two mutexes in a single round. The authors of [17] also posit (in a footnote) that allowing threads to acquire more than two mutexes per round would lead to race conditions and would have to be further studied. However, as publications like MAT and Kendo (cf., the next few paragraphs), or our own scheduling solution (cf., Chapter 7) show, this is not the case. In comparison to LSA, PDS improves efficiency by eliminating the need for inter-replica communication, but may achieve slightly worse performance because of round-based blocking of threads.

SAT vs. MAT

In 2006, Reiser et al. introduced terminology to distinguish between the models MTRDS and LSA/PDS implied regarding the number of threads actually executing in parallel: MTRDS is a so-called *SingleActiveThread* (SAT) solution, while both LSA and PDS can be called *MultipleActiveThreads* (MAT) algorithms. Correspondingly, the newly presented algorithm by Reiser et al. was named *Aspectix DEterministic Thread Scheduler — Multiple Active Threads* (ADETS-MAT) [71]. In ADETS-MAT, threads are separated into *primary* and *secondary* threads. Primary threads are allowed to perform mutex acquisitions (and other operations affecting determinism), but only one primary thread may execute at any given time. Secondary threads may run in parallel to the currently running primary thread, as long as they do not attempt to perform actions only permitted to primary threads. In comparison to PDS, there are certain types of replicated services which can benefit from ADETS-MAT's scheduling, while others may experience better performance under PDS [38]. An advantage of ADETS-MAT over PDS is the former's support for nested thread invocations and its ability to handle `wait/signal` operations as introduced by Hoare in 1974 [96].

ADETS-MAT

Kendo, published in 2009 by Olszewski et al. [68] takes a different approach

Kendo

3 This is the previously mentioned case where full determinism doesn't apply.

altogether, and introduces a deterministic logical clock for each thread, e.g., by using deterministic hardware events like certain CPU instruction counters, which, together with unique thread-IDs are used for determining an overall mutex locking order. However, while Kendo seems very applicable to use cases where multithreaded programs are run multiple times (e.g., for debugging purposes), it does not quite become clear how certain problems with Kendo would be solved when distributing it over several machines, even though the authors mention the possibility of applying Kendo to an SMR setting. Most importantly, the authors do not explain how thread creation in the case of new threads for incoming requests could practically work without either stalling the system, introducing indeterminism, or relying on uncertain knowledge. This will be further touched upon in Section 7.3.3.

Platforms Utilizing Deterministic Scheduling

The following publications are integrated solutions providing deterministic execution environments, as opposed to the previously presented conceptual schedulers.

CoreDet The "*CO*mpiler and *R*untime *E*nvironment that executes multithreaded C and C++ programs *DET*erministically" by Bergan et al. [23] aims to provide strongly deterministic execution using a compiler-based approach. Their environment adds additional synchronization code to make an application deterministic, e.g., for debugging scenarios. This means it works on a level closer to hardware than the approaches described above, and is capable of providing determinism even if an application does not explicitly obey high-level APIs as in weakly deterministic systems.

DThreads *DThreads* was heavily inspired by CoreDet. Its deterministic scheduling part is therefore not significantly different [61]. DThreads exchanges the standard POSIX `pthreads` library with a `dthreads` version, intercepting synchronization operations and enforcing strong determinism, similar to CoreDET. However, in contrast to CoreDET, DThreads claims to be more robust in terms of the actually achieved deterministic executions when compared to previous approaches, since their schedules only depend on the synchronization operations. In this regard, it is similar to UDS, albeit on another system level. We can simulate the behaviors of CoreDET and DThreads with our upcoming scheduling solution, which will be detailed in Section 7.3.4 and Section 7.3.5.

Parrot *Parrot* is also based on intercepting `pthreads` synchronization calls. One of its central claims, however, is that purely focusing on determinism is not sufficient for reliably testing multithreaded programs, as determinism says nothing about the stability of deterministic schedules in the face of slightly changed input [32]. Instead, the authors propose *StableMT*, which significantly reduces the number of possible schedules of a deterministic program, allowing developers to check each execution possibility for correctness. To this end, Parrot provides a new contract to developers, where by default programs are scheduled in a simple manner —similar to CoreDET and DThreads— but with the possibility of adding *performance hints* to programs, with which advanced de-

velopers can achieve better performance. However, their arguments, while likely true for debugging and development purposes, are relatively moot for the use case of SMR: Inputs to a program replicated to multiple state machines are identical between replicas anyway, and further, actual executions *have* to be deterministic without compromise, rendering stability relatively meaningless in this context. Nonetheless, the basic mode of Parrot can again be simulated by our scheduling solution (cf. Section 7.3.6).

CRANE is an SMR framework to equip arbitrary applications with transparent crash-fault tolerance. It was published by the authors of Parrot, and is based on the same scheduler Parrot had presented earlier. Hence, it uses the same deterministic round-robin scheduling as in Parrot's non-performance-hinted versions, albeit with an additional "time-bubble" mechanism, as a workaround to prevent indeterminism in special cases (when the arrival times of ordered threads show large wall-clock differences between different replicas). For more details, readers are referred to the CRANE publication, specifically section 2.2 [31]. *CRANE*

3.4 SYSTEM MODEL

With this related work and the basic terminology and foundational knowledge from the previous two chapters in our minds, we can define a system model for our future discussions. This model was originally implicitly adopted during the beginning of our research efforts on the upcoming scheduling algorithm as our first contribution, since this solution will be conceptually share similarities with related work such as PDS or MAT. The following explicit definition of this system model will clear up any potential ambiguities that exist between the models of the just presented related publications.

Our system will comprise a set of multithreaded processes running on different machines (also called replicas), so exactly one of the processes is running on each machine. Each process is responsible for execution of the service the system is designed to provide. Machines are connected via a network, with the capability to communicate and cooperate. The service is modeled as a synchronous remote procedure call (RPC) interface, i.e., clients can call a single method provided by the service while blocking and waiting for a single reply, before they can initiate another call[4].

Client requests are provided to machines in the same total order, established by a group consensus as described in the previous chapter. For each incoming client request, the processes on each machine all start one thread responsible for handling this request, which also sends a reply to the client after completing. All code for handling requests running within the threads has to be *deterministic* as defined in the beginning of this chapter, i.e., it can not introduce or

4 Since the service is replicated over multiple machines, clients will actually have to send requests to multiple replicas and be prepared to receive and compare multiple replies to see whether a quorum has been reached. This detail is irrelevant for the purposes of the provided service itself, however, as these tasks can be fulfilled by a supporting platform providing SMR primitives, so any service can be replicated easily.

depend on randomness[5]. Each process and its threads required for execution of the replicated state machine are exactly identical between nodes, but the surrounding environment in each machine may differ from other machines (e.g., in the operating system or hardware capabilities). Therefore, any *deterministic scheduler* needs to be able to fulfill its purpose (of making executions deterministic and identical between all machines) without relying on its machine's environment. We also assume our system to be *fully deterministic*, i.e., there is no further communication between replicas after the consensus has ordered requests and the threads for handling these requests are spawned. Note that for some of the following discussions of related work this last assumption does not necessarily hold.

Further, all access to shared data in a node is mediated by mutexes, meaning they protect access to variables that are to be shared between threads. We assume that the application (i.e., the process executing the replicated state machine) is implemented correctly, i.e., is *weakly deterministic*.

3.5 SUMMARY

In this chapter, we first motivated the introduction of multithreading to current systems, before detailing a series of terms for properly describing characteristics of deterministically multithreaded distributed systems. A series of related work was presented afterwards, which introduced concepts and systems that previously implemented deterministic multithreading using various strategies. Finally, we defined the system model our research in the remainder of this thesis will be based on. For now, we are left with the penultimate chapter of this first part of our work, which will introduce a last compilation of background information on optimization approaches. Afterwards, we specify our research goals and formulate the problems this thesis aims to solve, before transitioning to our contributions in Parts ii and iv.

5 For example, by depending on wall clock time, random functions, or other factors that might differ between replicas

OPTIMIZATION

4

The last part of our thesis title that remains to be properly introduced is the very first of its words: *Optimization*. Broadly speaking, optimization generally refers to the process of finding an optimal, i.e., best, solution from a larger set of possible solutions. Mathematically speaking, optimization methods attempt to find a value within a set of given possible values for which an *objective function* is always smaller or larger than for all other values of the set [90].

4.1 CHOOSING AN APPROACH

Large Field of Research

However, this seemingly simply definition hides a concept that covers multiple disciplines of research and is entirely too big a term to fully do its complexity justice within a single background chapter such as this one. One simply has to consider the fact that optimization approaches can be subcategorized in such a myriad of ways, that, no matter which perspective one chooses to look at this field, its multitudinous facets have the tendency to overwhelm by their sheer number.

Categorization by Problem

To give just a few examples of this, we could begin by dividing optimization problems into those that provably contain one global best solution—without yet knowing anything at all about how this optimum could be found—, or those that may contain multiple optimal solutions with respect to locally confined boundaries within the set of all possible solutions. The former are called problems with convex objective functions, while the latter problems are, not surprisingly, called nonconvex [90]. Just determining whether particular problems are convex or not is the subject of entire research fields.

Categorization by Method

We could also try to categorize optimization not by the problems it is trying to tackle, but by the methods it employs. These could, for example, be distinguished by whether they can provably find an optimal solution out of all possible solutions, or whether they are only approximating it. Among the latter methods, a further subcategorization can tell us whether these methods yield mathematically provable upper limits to the distance between the approximated solution to the truly optimal one (called approximation algorithms), or whether an approach simply finds a *good* solution without any hard evidence about *how* good this solution is compared to the overall best one (many heuristics fall into this last category). For each of these categories, a veritable deluge of techniques rears their heads and promises various degrees of accuracy, speed, ease-of-use, and maturity ([40], [102], [28], [76]).

Categorization by Shape of Data

A wholly different categorization attempt could subdivide optimization approaches into whether they find the best solution to a problem that is described by discrete or continuous data. In other words, this categorization would distinguish between methods that iterate over a discretized space of variables, and methods that attempt to find optimal values for continuous functions.

Categorization by Knowledge

Optimization can yet again completely differently be categorized into on-

line and offline approaches, where offline approaches are those that have full knowledge of the behavior of a system, including future states after input is applied, whereas online optimization has to deal with only having an inaccurate picture. Put differently, online optimization attempts to inch towards its goal by deciding new parameters based on periodical and partial or even inaccurate feedback from the system, while offline methods have access to a full data set about a system's behavior or a fully accurate model of the system in terms of its reactions to optimizations ([81], [86]).

Categorization by Number of Goals

We could also categorize techniques by their ability to optimize a single objective function, in contrast to those that allow for the optimization of multiple objectives at once. The latter category alone is still large enough to warrant whole survey papers laying out its further subcategorization ([73], [65]).

Simply put, the mere attempt to categorize the field of optimization has the potential to induce headaches. Nonetheless, we have to, of course, provide the proper context to situate the research presented in this thesis within the larger area of possible optimization approaches. To this end, this chapter will use core assumptions of our system model, as defined in the previous chapter, coupled with circumstances during our PhD, such as available resources and previous knowledge, to narrow down possible optimization methods.

Optimization Goals and Metrics

Definition of Optimization Goals

To narrow down the possibilities for choosing valid optimization approaches, we first define what exactly it is that we want to optimize. In order to optimize a real-world system, we need to define one or more proper *goal(s)* (i.e., the objective function) we want to optimize for, and a *metric*—or multiple ones—which allow us to track our progress towards achieving the chosen optimization goals. Additionally, we should have a clear understanding of the *parameters* we can manipulate in order to influence the metrics and move towards our optimization goals.

A non-exhaustive list of some common optimization goals for real-world systems could include:

- System Output / Performance
  - Throughput
  - Latencies
  - Success and error rates
  - System uptime / Reliability / Availability
- Efficiency
  - CPU utilization
  - Memory pressure
  - Hard disk utilization
  - Queue lengths within protocols or hardware

  - Network / Bandwidth usage
  - Electricity usage / Operation costs
- Domain-specific goals like
  - Sales closed
  - Entities manufactured
  - Events generated

Definition of Metrics

The exact definition of metrics required to measure these goals vary greatly from goal to goal, and are dependent on specific cases. For example, in order to measure request success and error rates, we have to define exactly what constitutes a *successful* request as opposed to an erroneous one. Maybe errors can be thrown during the execution of a request, but with proper error handling the client still receives a valid answer. In this case, a metric measuring error rates has to decide whether it includes these occurred errors or whether it can ignore them because the overall request still completed successfully. For this thesis, we defined the main optimization goals to be *performance* and *resource efficiency*, the latter specifically regarding the operation costs of SMR systems. The choices fell on these goals after discussions within our research group and comparison of previous works in this area of research. It was envisioned that optimizing these goals could help with the adoption of SMR as a technique, which in turn would increase the reliability of systems used in the wild—a common desired outcome of our research we have already expressed in the very first pages of this thesis.

Regarding metrics that could track the effectiveness of employed optimization methods regarding performance, a common approach in SMR systems research is to measure request throughput rates and end-to-end latencies, where we would like to maximize the former and minimize the latter. As for a simple metric that shows cost optimization, we simply look at the daily fees that running a single replica would cost when deployed in a common cloud environment.

Definition of Parameters

Finally, while the exact set of parameters we could manipulate within our prototype systems became clearer during the course of our research, a few core properties of the envisioned system were known from the outset. Our soon-to-be-presented novel deterministic scheduling solution had preliminarily been proposed when we started our PhD, so we knew some of its parameters already. On top of that, the underlying library we were to use for our implementations was known to be BFT-SMaRt [25] from very early on, since apart from this library, no other suitable SMR libraries for research purposes existed. Building our own SMR library from scratch was quickly ruled out as an option (cf., [36] for a good up-to-date summary of the reasons why). Hence, we knew the parameters this library offered, too. Altogether, this anticipated collection of all parameters of our system setup consisted of tens of variables, both discrete and continuous.

Rule Inference

The First Approach

With these first facts about goals, metrics and parameters known, a few initial decisions regarding the suitability of optimization approaches were possible. First and foremost, it was clear that our entire approach would have to be primarily based around implementations and measurements of prototype systems, since creating sound mathematical models of our entire distributed system setup seemed impractical, if not outright impossible with our resources. Previous research efforts within our group showed that the mere attempt to formalize a single Byzantine consensus protocol using state-of-the-art tools already represents a huge undertaking, requiring many simplifications and restrictions for a model to work at all—not to mention the formalization of a whole SMR setup.

Therefore, a first approach we decided to pursue was rigorous simplification and manual inference of rules and relationships between system parameters and behaviors. By this we mean that we planned to look at only certain subsets of our system, e.g., the deterministic scheduling by itself, and to use models and simulations to get an intuitive understanding of this subsystem's behavior under different circumstances and various configurations.

Based on this understanding then, we intended to create simple proof-of-concept algorithms focused only on optimizing those parameters we had built an intuition for, in order to evaluate their effects within the entire system setup. This approach later led to our first successes in regard to self-optimizing SMR systems, which will be discussed in Chapter 14.

Reinforcement Learning

The Second Approach

For true self-optimization of the entire system as described above, with a multitude of parameters and goals, we were primarily looking towards online optimization methods, i.e., methods that have the ability to interact with complex live systems. This subcategory of optimization is also often called *optimal control*. Theoretically, these optimal control problems could still be solved using more classical methods such as dynamic programming (cf., [70], for example), but the aforementioned lack of a solid mathematical model of our environment finally led to researching recent advances in online learning. This field, a discipline of machine learning, has in recent years made great strides in tackling difficult optimal control problems [45]. Online learning approaches are particularly well suited to optimizing systems which are not mathematically modeled or formalized, but that exist in the real world and can be interacted with. One popular and especially well-suited approach to optimizing these complex, unmodeled live systems is Reinforcement Learning (RL) [88]. While the idea of RL itself is not that young ([50]), its recent advances, for example in robotics control ([95], [51]), in beating humans at complicated games ([66], [74], [79]), or for optimization tasks in real-world systems ([67]) were primarily achieved by combining it with Deep Learning, i.e., the use of deeply layered neural networks. More background information and related work on this ap-

proach, called Deep Reinforcement Learning ([100]), will be presented later, when we talk about how we attempted to utilize RL for the optimization of our systems (cf., Chapter 15).

4.2 SUMMARY

With this penultimate chapter in the foundational part of our thesis, we briefly introduced the multidisciplinary field of optimization and drew attention to its wide variety of research areas. We motivated the optimization goals that were to lead us through our research, and followed this by giving insight into our rationale behind choosing mainly two optimization approaches. The first of these approaches, rule inference and the manual development of algorithms exploiting these rules, would play an important role in the first years of the PhD, where we designed and implemented the prototype of our self-optimizing SMR system. Finally, we teased Reinforcement Learning as our second approach, which will be further presented towards the end of this thesis.

PROBLEM STATEMENT

5

The previous three chapters have introduced background knowledge, terminology, and related work to define the context of our research and help with understanding the contributions of this work. We will now conclude this foundational first part of the thesis by outlining our initial research goals when we began our PhD, alongside a few updated goals that developed over the years as we grew into the topic and discovered potential avenues of further research. The goals are formulated alongside short problem statements pertaining to the respective goal, so as to both motivate our approaches and show the initial conditions we encountered when starting our research efforts.

5.1 RESEARCH GOALS

Main Thesis Goal

From the outset, our primary research goal for our PhD, when condensed to its core, read:

MAIN GOAL Utilizing deterministic multithreading, investigate approaches that can enable state machine-replicated fault-tolerant systems to self-optimize themselves during runtime, while requiring minimal input from developers or operators. Research the most promising approaches, using—where possible—prototype implementations, measurements, evaluations, and comparisons to similar existing solutions.

This overarching goal was then further subdivided into individual research goals that were either developed during the initial weeks of the PhD, or evolved over the following months, in collaboration with fellow researchers and cooperation partners within our DFG-funded research project.

5.1.1 *Runtime-Configurable Deterministic Scheduling*

First Research Goal

In previous work, it has been shown that the performance of multithreaded systems based on deterministic scheduling can vary greatly depending on the combinations of particular scheduling solutions and application execution characteristics [38]. This problem could be mitigated by inserting a runtime-reconfigurable scheduling solution into these deterministically multithreaded systems, by enabling adaptation to various application profiles and to other changes in the surrounding execution environment. One sub-goal therefore was to first *contribute to the finalization of research* on such a scheduling algorithm, which was already underway when we joined this project. Further sub-goals then had in mind the *implementation, testing, and evaluation of this scheduling algorithm*, to bring it from the realm of ideas to the physical world.

5.1.2 *Improved Resource-Efficiency of SMR Systems*

Second Research Goal

With deterministic multithreading, state machines can utilize modern multi-core hardware better. However, this added performance comes at a cost: Machines have to be provisioned for maximum achievable throughput lest they risk poor user experiences during high load phases, which costs money and needlessly takes up resources. The goal was to *develop an architecture capable of automatically scaling hardware resources* of these machines *during runtime*, to save costs and free hardware for dynamic re-allocation *within cloud contexts*.

5.1.3 *Improved Performance of SMR Systems*

Third Research Goal

Utilizing not only deterministic multithreading, but our novel and reconfigurable version thereof, potentially significant performance improvements were thought to be possible for SMR systems that face highly variable load or other changes in their environment. Our goal in this regard was to *research ways in which the dynamicity of our scheduler* could be applied *to realize* these envisioned *performance gains*, and *build a prototype system* with which the effectiveness of our approaches could be *measured*. During this research, three sub-goals emerged, which can be summarized as follows:

Subgoal 3.1

First, to have a basis on which reconfiguration decisions could be formed, a deterministic metric was required that would work even in Byzantine settings. We were to *develop, implement, and evaluate a technique for providing this decision basis*, and had the further goal of *integrating it with our prototype* for evaluating SMR system performance.

Subgoal 3.2

Second, a straightforward *proof-of-concept mechanism was to be invented* which clearly *demonstrates the validity of the research approach*, i.e., the idea of utilizing reconfigurable deterministic multithreading to improve system performance during runtime. In this vein, our goal was to look at experiences we had gathered so far in the previous months of research, and to *identify a way that would quickly lead to a successful demonstration* of our approach.

Subgoal 3.3

Finally, our last goal was to *realize the originally envisioned, fully self-optimizing system*, by way of first *researching the most promising approaches*. Then we were to *choose a suitable method, implement it within our system prototype*, and finally *measure its effects on performance* during runtime under a variety of different load and application profiles.

5.2 SUMMARY & OUTLINE

This presentation of research goals concludes the first part of our thesis.

While the introduction already laid out a roadmap of the thesis (cf., Section 1.2), this following outline is meant for a quick re-orientation, now that we have finished laying the foundations and provided an enhanced understanding of our research topics and goals.

Outline

The very next part will introduce and explain in detail our runtime-reconfigurable deterministic scheduling solution, called *UDS*, which completes our first re-

search goal and provides us with an elementary tool for all further research efforts. In Part III, we then explore how exactly deterministic multithreading, enabled for example by schedulers such as UDS, can allow for resource efficiency optimizations in SMR systems. This part will also demonstrate evaluations based on a prototype implementation and thereby complete our second research goal. Finally, Part IV will present our research into using UDS' reconfigurability to improve the performance of SMR systems during runtime. We first develop and evaluate *ByTI*, our solution for deterministic measurements of certain performance metrics in Byzantine settings. Then we implement and benchmark a first iteration of a self-optimizing SMR system, based on rules we inferred from our experience gained through using UDS. As our last contribution, the research and implementation efforts we put into a neural network-based Reinforcement Learning approach are laid out. Altogether, this part completes our final three research goals. To conclude and provide an outlook, Part V concisely summarizes the contributions of this thesis, provides a collection of future work we identified in these areas of research, and closes with an outlook and rather brief personal opinions on some matters surrounding this research.

II

UNIFIED DETERMINISTIC SCHEDULING

CORE IDEAS

6

Deterministic scheduling, e.g., for replicated state machines or debugging multithreaded programs, has been a topic of research since at least the early 2000s. Previous work in this area has already been presented in Section 3.1. However, as Domaschka et al. have shown in 2008, the runtime performance of systems employing such a scheduling solution highly depends on the particular locking patterns of the application that is being executed [38]. Switching to a different scheduler optimized for a deployed application could therefore significantly improve performance, as long as that application does not change in its execution behavior, or a new one is deployed to the system. One idea to mitigate this problem depends on a feature that all previously published works lack: Reconfiguring the scheduler during runtime of the system, in order to adapt to changing system parameters. These can include the current load put on the system by clients, the distribution of the type of requests being sent to the system or even reconfiguration of the SMR cluster to include new or exclude old nodes. In other words, existing deterministic schedulers are either not configurable at all (MTRDS, LSA, MAT, Kendo), or offer only very little configurability (e.g., by choosing between PDS-1 and PDS-2 at system startup). This makes them unsuitable to our research goal of improving the performance of deterministically multithreaded systems.

6.1 FUNDAMENTAL PRINCIPLES AND NOTATIONS

To tackle this problem, our research group set out to develop a novel scheduling solution, to potentially unlock substantial optimization potential. We christened this scheduling solution the *Unified Deterministic Scheduler* (UDS), and it was designed to provide both significantly more opportunities for configuration than previous algorithms and also to have the capability to be reconfigured during runtime of the system, all while preserving determinism.

The following sections will detail the underlying ideas of UDS and provide a thorough description of its development, its exact operation, and the functionality and opportunities it unlocks when being used in favor of previous deterministic scheduling solutions.

6.1.1 *Scheduling Rounds*

Motivating Round-Based Scheduling

UDS is a round-based deterministic scheduler. While this terminology was first introduced in Section 3.3, we provide a more in-depth introduction, specific to UDS, here. Let us first motivate the concept of scheduling rounds. As discussed before, all replicated state-machines are guaranteed to receive the same requests in the same total order, and each request will be handled by a single thread spawned in each replica. Each thread can be given a unique and identical ID between all replicas, simply by assigning ascending numbers in

the order of arrival. In order to fulfill request processing, these threads will occasionally require access to shared data. According to our system model, shared data should be guarded by mutexes to prevent data races. When observing the same order of mutex acquisitions among all threads in a piecewise and weakly deterministic system (see Section 3.1), the system will behave deterministically [5]. From here on out, we will call attempted mutex acquisitions, or more generally all actions that could affect determinism, *critical actions*. Guaranteeing the same order of critical actions across several replicas will ensure their determinism, as long as they are not implemented incorrectly (e.g., by deliberately generating randomness).

Sequential vs. Parallel Execution

A sequential scheduler is therefore trivially deterministic, e.g., by executing each thread in ascending order of IDs, and letting each thread complete its execution before starting the next. Since the order of threads is identical between replicas, the order of critical actions taken by sequentially scheduled threads will also be deterministic without further efforts. When trying to parallelize execution, however, we are presented with a problem: In order to achieve determinism, a scheduler has to decide on a certain order of critical actions, based on currently available information. In systems that are fully deterministic to comply with our system model (cf., Section 3.4), this means that the scheduler needs to decide this order without communicating with other replicas.

Handling Input

Looking at the information we have available, it might seem at first glance that since we know the order of requests, we can design a scheduler that optimally schedules critical actions with relative ease. However, while the order of incoming requests is guaranteed by the system's consensus solution, the exact timing of message reception is not, meaning that between two different state machines, one can receive a request much earlier than the other (when measuring arrival using an abstract, absolute wall-clock time). All we can reasonably assume is that messages are received in a *timely manner*, meaning that the system will *eventually* deliver all messages to all replicas—within sensible boundaries (a concept known as *Liveness*). Any algorithm trying to decide on parallel scheduling of critical actions can therefore never know deterministically which threads are currently available, or, in other words, can not know whether the thread list it sees in its local replica is the same as other schedulers see in other replicas. Without further help, schedulers therefore can not know which threads to include in their scheduling decisions, because new input may arrive at any time.

Additionally, if threads are permitted to spawn their own sub-threads, these difficulties are exacerbated: We do not currently have a way to deterministically insert them into the list of ordered threads, since at specific points in time these lists could be of different lengths at different replicas. Put differently, without any additional help or a new idea that solves this problem, in a *live* system, i.e., a system which continuously receives input, it is impossible for a deterministic scheduler to know *when* exactly to decide on a schedule, or when to include new threads into its scheduling decisions. In systems that merely require deterministic scheduling for debugging purposes, i.e., where the execution of the same sequence of threads is to be repeated multiple times, we

know the total order of all threads across the entire execution without having to deal with additional incoming input. For such systems designing a scheduler that at least works would be relatively easy[1], since we can take the entire list of threads and impose a fixed order of critical actions across all of them before starting execution.

Introducing Rounds

When dealing with continuous input, we need to somehow introduce deterministic points in (logical) time, at which our envisioned parallel scheduling algorithm can—with certainty—plan ahead and assign an order of critical actions to a known number of threads. Our solution to this problem is to take the next n threads, where n is a small and known number, and assign critical actions to these n threads. Since n can be counted up deterministically in the identical total order of all threads, we can be sure that all schedulers will always include the same subset of n threads for their next decision[2]. Such a deterministic subset of n threads in combination with the particular assigned order of critical actions is called a *scheduling round*, as first introduced by [17]. After all n threads of a round have either terminated or "used up" all of their assigned critical actions, the round ends and a new round is started, for which the scheduler can now deterministically pick the next n threads from the list of remaining threads (which may or may not include threads that have been part of a previous round before). Rounds themselves are numbered for reference, usually starting with r_0 for the first round and counting up for each subsequent scheduling round. If we assume that a system can have multiple instances of a deterministic scheduler, then these schedulers are named by unique IDs. The tuple of scheduler ID and round number uniquely identifies any scheduling round in a system.

Primary Threads

Scheduling rounds break down the problem of having a list of unknown length of threads during runtime, into smaller, deterministically known subsets of threads which can be safely reasoned about. From here on out, we will adopt the name *primary* (threads) for all threads which are currently part of a scheduling round, and generally denote the number of *primaries* (i.e., the size of a scheduling round) with the letter n. For example, if we let our scheduler choose four threads for a scheduling round, we say that $n = 4$ and there are currently four primaries. All other threads which are not part of the current round have no special name. These other threads may freely start and run in parallel, but only up to the point where they want to perform their first critical action. At this point, they are blocked and have to wait until they become primary. Once they have become primary, they may be allowed by the scheduler to perform their critical action(s), the details of which will be explained in more detail in the following Section 6.1.2.

Waiting for Required Input

One of the core implications of the idea of introducing scheduling rounds is so important we prefer to specifically lay it out here for the sake of completeness: Should any of the chosen n primaries for a scheduling round not yet

1 But even in this case, designing a highly performant scheduler which generalizes well across different programs might still be difficult, of course.

2 We will later also show a solution to deterministically assign IDs to subthreads spawned by other threads during runtime

be present at a replica, this replica has to wait until all these n requests have been received, before permitting those threads to perform critical actions (in some algorithm-specific predetermined order). It is this waiting which makes or breaks our deterministic scheduling. Optimizations to this scheme are possible under certain circumstances, and will be introduced in Section 8.3.

A Simple Scheduling Example

For example, if we let $n = 2$, each replica will start with threads t_0 and t_1 as primaries in their very first scheduling round. If a replica has not yet received either thread, the deterministic scheduler in this replica will block *any* thread trying to perform a critical action (while others can run freely in parallel), wait until *both* t_0 and t_1 have been delivered at this replica, and then allow only those two threads to take critical actions in a predetermined order.

6.1.2 *Steps*

Let us also briefly look at how exactly critical actions can be assigned to threads within scheduling rounds. Whenever a new scheduling round is started, the scheduler has a known number of n primaries. Each of these has become primary because it requested to perform a critical action at some prior point in time. After deciding the set of primaries for the current scheduling round, the next decision our scheduler has to make is the order and number of critical actions our primaries are allowed to perform within our round.

Steps and Total Orders

During a thread's lifetime, it can switch from being blocked and waiting to being granted a critical action and resuming multiple times before finally terminating, resembling a somewhat philosophical journey through our system. Therefore, we call the concept of a primary being granted a critical action a *step*, and say a primary is *consuming a step* when it is unblocked by the scheduling algorithm to perform its requested critical action. We also sometimes simply call the overall order of steps within a round the *total order*, abbreviated as *TO*, which can be somewhat confusing when compared to the total order of all incoming requests in a replica; however, this generally works well within the context of talking about schedulers and scheduling decisions.

Notation

To better distinguish the two, we will start calling the ordered list of all threads in the system Θ or simply `allThreads` as a variable in code.
The primaries, i.e., the currently chosen list of threads permitted to perform critical actions in the current scheduling round, are either simply called *primaries*, or denoted by Θ_{prim}. When denoting threads with their IDs, we will use t_{id}, where id is an incrementing number representing the thread's position in the consensus-dictated total order of all incoming requests.
Similarly, to distinguish several mutexes in a system, we will note them down as m_{id}, where id is again simply a unique number per mutex.
To denote the total order of steps within rounds, we introduce the following notation: Squared brackets including a comma-separated list of numbers, where each entry in the list denotes a step for the thread with the corresponding position in Θ_{prim}. Steps can then be consumed by the corresponding threads in the order they appear in this list.

Notation Example

To illustrate our notation with an example: If we are in the 42nd schedul-

ing round of a scheduler instance, rounds currently have $n = 3$ threads, the threads chosen for the next round are t_{18}, t_{19}, and t_{21}, and we design our scheduler so that it permits each thread to perform exactly one critical action per round, in ascending order of their thread-IDs, we say *Scheduling round #42 has three primaries, Θ_{prim} = [t_{18}, t_{19}, t_{21}], and the total order is [0, 1, 2]*. If we were to grant each primary two steps in a round-robin fashion, we might say *the total order is [0, 1, 2, 0, 1, 2]*.

Total vs. Causal Ordering of Actions

Lastly, let us have a look at total and causal orders of critical actions. If we observe closer how critical actions such as acquisitioning mutexes are performed, then we notice that in non-deterministic systems, threads first *request* this action, before they can actually *perform* the action. This is because when trying to acquire a mutex, threads may not immediately get it in case the mutex is currently held by another thread. Only once the mutex is free can the requesting thread actually be granted access, thereby performing its previously requested action. With a deterministic scheduler, we add a third step—the assignment—in between requesting and performing the action. An assignment happens at a specific, deterministic point in (logical) time within a scheduling round, and ensures that from the viewpoint of a mutex, the assigned order will be observed when threads actually acquire it. This can be achieved by a per-mutex wait queue in which we enqueue threads that have requested and been assigned this mutex, but can not yet acquire it because it is held by another thread. As we have shown previously ([5]), it is sufficient to observe a total order of assignments to ensure determinism in our system model. The total order in which threads perform actions across multiple mutexes can slightly differ between replicas (always guaranteeing at least the same causal ordering of actions, however), but since the assignments were totally ordered, from the viewpoint of a single mutex the order of acquiring threads will be total.

Ordering Example

We can construct a simple example to illustrate this: Let us assume we have three primaries, Θ_{prim} = [t_1, t_2, t_3] as well as two mutexes m_1 and m_2. In order to fulfill their client requests, t_1 and t_2 both need to acquire m_1 during their execution, while t_3 requires m_2. For all replicas, we assume that our deterministic scheduling solution constructs a scheduling round with these three threads as primaries, and a total order of steps of the form `[0, 1, 2]`. We assume that in a replica `A`, all three threads are present and waiting, because all have requested their respective mutexes when the scheduler starts the scheduling round. The scheduler will, according to the total order, allow t_1 to consume its step, which will immediately unblock t_1 and let it fully acquire m_1. We can imagine the total order now to look like this: `[1, 2]`. Next, the scheduler in replica `A` will let t_2 consume its step, which will attempt to assign m_1 to t_2. Unfortunately, t_1 is still holding m_1, so t_2 is put into the per-mutex wait queue of m_1, to signify that it will be guaranteed to be the very next thread that may acquire this mutex as soon as t_1 releases it. The total order now has only the step for t_3 left:`[2]`. t_3 consumes its step, and since m_2 is not currently held by anyone, it can immediately acquire it. After an unspecified amount of time, t_1 finishes executing within the critical section guarded by m_1, releases this mutex, and since t_2 is found in its wait queue,

t_2 is immediately unblocked and made the owner of the mutex. Now let us assume that in another replica B, the same initial operations are performed: t_1 consumes its step and immediately acquires m_1. Now, however, since B just so happens to have considerably more potent hardware than A, t_1 manages to finish the few instructions within the critical section guarded by m_1 in parallel to the scheduler doing its scheduling. Therefore, when t_2 is unblocked, m_1 has, by chance, already been released by t_1, so t_2 can immediately acquire m_1. Afterwards, t_3 acquires m_2 as usual.

In this example, the global total order of mutex acquisitions differs between replicas `A` and `B`: In `A`, the mutexes are acquired in the order $m_1 \rightarrow m_2 \rightarrow m_1$, while in `B`, the total order of acquisitions is $m_1 \rightarrow m_1 \rightarrow m_2$. However, the causal ordering of mutex acquisitions is preserved, and, from the viewpoint of the individual mutexes as well as from the viewpoint of the threads acquiring them, the order of acquisitions is identical between replicas. If the application is implemented correctly (i.e., the two mutexes m_1 and m_2 are guarding different shared data), the two executions will be deterministic, i.e., produce the same resulting state.

Note how our envisioned deterministic scheduling solution does not need to control exact timing of thread execution. Simply ensuring the order of critical action assignments guarantees deterministic execution within our system model.

6.2 CONCERNING PERFORMANCE

At this point, we might want to make time for a short reminder of what we are building up to, i.e., why we are doing all of this in the first place: Our current goal is to introduce parallelism to state-machine replicated systems, in order to increase their performance during runtime. Generally speaking, to utilize current multiprocessor hardware best, we would like to maximize parallelism, while ensuring deterministic execution. Hence, in this section, we will take a brief look at the problems our currently envisioned scheduling solution might face when maximizing parallelism, so as to sensitize ourselves to the aspects we should focus on when optimizing performance. In other words, we need to know which problems our scheduler might face or even introduce into the system, in order to minimize their overall impact on performance.

6.2.1 *Round-Filling Delays*

Maximizing Primaries

It seems quite obvious at first glance that in order to increase parallelism, we would like to increase the size of our rounds as much as possible. Allowing more primaries into each scheduling round increases the likelihood that among these many primaries, multiple different types of requests are represented, with each type requiring a different set of mutexes. Generally speaking, the more diverse the set of mutexes that can be assigned and ultimately acquired during a scheduling round, the larger the chance we achieve high parallelism during the execution within that round.

Waiting for Input and Stalling

However, , as we increase the number of primaries, we increase the chance we have to wait until the required number of threads have arrived at our replica, by virtue of clients making requests to our system. This is most easily explained by a simple example: Let us imagine we only have clients that are sending synchronous requests to our SMR-system, meaning that the next request will only be sent once an answer to the previous request has been received. In that case, at any point in time, the maximum number of requests that can be in the system is equal to the number of clients that are currently connected and actively sending requests. If we design our scheduler to maximize parallelism by setting the number of primaries high, then as soon as we set the number of primaries higher than the number of currently active clients, the system will stall indefinitely and can not proceed without further input. An easy example would be one currently active client and a system with $n \geq 2$ primaries per round. The first request by this client will be delivered to our replicas, a thread will be spawned in each replica to fulfill the request, and as long as this thread does not request any critical actions, it can start executing. However, as soon as a critical action is required, our scheduler will block the thread and attempt to start a new scheduling round. As we have learned in Section 6.1.1, however, we can not start the round until at least n threads have arrived in the system, which will never happen because there are less than n clients and no new requests are generated until the old requests are answered. Hence, we are stalled indefinitely if we do not introduce further mechanisms like artificial request creation, or until enough new clients connect and send requests to our system. Even in cases where clients send asynchronous requests (i.e., they send out new requests without first waiting for replies to old ones), the system can be frequently delayed depending on the rate with which clients are sending out requests and the size of the scheduling rounds we choose in our scheduling solution.

This problem is a fundamental drawback of all round-based deterministic schedulers which allow for $n > 1$ primaries, and we will demonstrate and visualize this effect in Section 8.1.1. We shall keep these delays caused by filling scheduling rounds, or *Round-Filling Delays* (*RFDs* for short), in mind as we look for solutions that can mitigate this problem.

6.2.2 *Unbalanced Execution Times*

A scheduling round ends as soon as all primaries are either terminated or blocked on a critical operation that is no longer part of the current round; e.g., when the scheduler allows one step for each primary, but threads need more than one critical action to complete their execution.

Delayed Round Endings

However, if a primary executes some long-running operation while all other primaries in the current round are already terminated or waiting for their next critical operation, the round ending will be delayed. This in turn delays the start of the next round, where currently blocked threads could proceed, thus reducing concurrency overall. We call this effect *Unbalanced Execution Times*, or *UEBs* for short. UEBs can have a significant effect on performance, since

they can reduce parallelism considerably. We demonstrate and visualize this effect in Section 8.1.2.

Delayed Critical Actions

This effect is not only observable at the end of rounds, but also when the scheduler assigns multiple critical actions to each primary in a single round: A thread may already be waiting for its next critical operation, whereas others who come before in the total order are still executing and have not yet consumed their steps, thereby delaying the first thread. Hence, UEBs can increase the latency experienced by clients, as their requests have to wait for other long-running threads.

6.2.3 *Lock Congestion*

One obvious cause for limited concurrency is when primaries have to wait for a mutex because another thread has not yet released it. In this case the achieved parallelism of the round is reduced—in the worst case to almost sequential execution. However, this problem is mostly inherent to the application's logic and cannot easily be solved by the execution environment.

6.3 DYNAMICITY

With a strong understanding of our notation, of how a round-based deterministic scheduling solution could operate, and of the most common problems a scheduler faces or introduces when trying to maximize parallelism, we can introduce the central idea that distinguishes our scheduling solution from previous work and enables us to employ effective countermeasures to some of these problems.

Fixed Configuration Schedulers

To introduce this idea, let us first take a closer look at some commonalities between the previously presented related works in the field of deterministic scheduling (cf. Section 3.3). At first glance, they look rather diverse in their approaches and results. However, when examining them more closely, we can come to the following realization: Once a system using one of these approaches has been initialized, the configuration of its deterministic scheduler, e.g., the number of primaries per round or the number and order of steps within rounds, among other parameters, is fixed, and stays that way during the entire runtime of the system.

Application-Specific Performance

We stress this fact because as Domaschka et al. have shown in 2008, the previously presented deterministic scheduling strategies perform very differently depending on the *application profile* of the program being executed, i.e., the type of service being provided by the system [38]. By *application profile*, we mean the distinct patterns of critical actions and calculations threads need to perform in order to execute the service, and the collection of different such patterns for various tasks in the application. As an example, if the service provides a simple CRUD[3]-based data storage system, clients will use these four different request types to interact with the system, and each thread handling

3 Create, Read, Update, and Delete

a request of a certain type will have different patterns of locking and unlocking mutexes in between calculations or memory accesses. We will revisit the claims of application-specific performance behavior in a later chapter, with benchmarks suited to our needs, but for understanding the core ideas of our envisioned deterministic scheduling solution it is sufficient to remember that fixed configuration scheduling can only achieve optimal performance for a very narrow set of applications—namely, the ones it has been tuned for.

Dynamicity

When presenting RFDs and UEBs like this, followed by the thoughts about fixed-configuration scheduling solutions, it seems rather plausible that a certain level of *dynamicity*, i.e., reconfigurability of the scheduler *during runtime*, might help with mitigating these problems. If our scheduler had the ability to change the number of primaries of scheduling rounds, it could react to sudden changes in the rate of incoming requests in order to avoid RFDs. An ability to cleverly select the next primaries from Θ and the capability to change the total order of steps might be able to mitigate UEBs. We should therefore investigate possibilities to add dynamicity to deterministic scheduling, and research its potential effects on system performance.

7

In this chapter, we are going to present our *Unified Deterministic Scheduling* Algorithm (UDS), which improves on related work in several ways and allows for finely-tuned performance adjustments thanks to its ability to reconfigure itself during runtime. We will use pseudocode and the previously introduced notation to detail the ideas and inner workings of our scheduler, and afterwards compare it to existing scheduling solutions. Finally, in Chapter 8 we will lay out several strategies we pursued when trying to create implementations of the algorithm, as well as evaluation results obtained while vetting it for correctness and to discover potential performance improvements.

Some passages of this chapter are partially based on our published paper on UDS [5], and it is important to note here that the algorithm itself was not developed solely by the author of this thesis, but in teamwork by the authors of said paper. The main contributions to this topic by the thesis author are the implementation and evaluation efforts detailed later, alongside several bug fixes and some optimizations to the UDS algorithm. The algorithm as presented in the following sections already includes several bug fixes compared to the originally published version from 2016. The differences will be detailed in Chapter 8.

7.1 THE CORE UDS ALGORITHM

Scheduler State

As specified in our system model, incoming requests arrive in the same order in each replica, and this order is used to identify all requests. On arrival of a new request, a new thread is started and appended at the end of the totally ordered list of all threads Θ. This new thread will take care of processing only this single request. Our system contains the primary threads Θ_{prim} and a set of mutexes, each with its own queue of threads waiting for this particular mutex to become available. Primaries are the only threads permitted to perform critical actions.

The state S of our scheduler is defined by:

```
1 Θ := [] # ordered list of threads
2 Θ_prim := [] # ordered list of primaries
3 round := 0  # round number (0 = no round yet)
4 highestThreadNo := -1  # threads already seen
5 progress := false  # did we make progress this round
```

Of course, as previously motivated, UDS is a round-based scheduler. At the beginning of each round `r`, Θ_{prim} needs to be deterministically defined. To achieve this, UDS uses the first `n(r)` threads from Θ fulfilling a deterministic predicate `prim(r,`Θ`)`, which allows us to filter out inappropriate threads for `r`. In the easiest case and if not specified otherwise, for all further discussions the predicate is simply always true. Other definitions are discussed in Sec-

tion 7.2. If there are less than `n(r)` threads in the system, the round cannot start until enough new threads are created due to new requests being delivered. Should UDS be used in debugging scenarios, `n(r)` has to be limited by the number of all available threads.

Starting New Rounds

To start a new round, the following function is used. Note that all operations following in this section are assumed to run mutually exclusive, i.e., only one thread is actively executing them at any time, thereby protecting state S from corruption due to e.g., illegal concurrent access. This can be achieved in actual implementations by encapsulating all following scheduler code with its own `schedulerLock`, which has to be acquired by threads wanting to execute these methods. Threads relinquish control of the `schedulerLock` by using `wait/signal` primitives to block themselves and obey the order defined by UDS' scheduling methods, which is what the calls to `waitFor()` signify in the following code listings. Waiting threads can be then be woken up at these points by other threads signalling them using the given conditions in the pertaining initial `waitFor()` call of the original thread.

```
6  startRound() := {  # called to start a new round
7     r := ++round
8     Θ_prim := []
9     reconfig(r, progress)  # reconfigure for next round
10    progress := false
11    for(n := 0, i := 0; true; i++) {
12       waitFor(exists Θ[i])    # delay (!)
13       if(!Θ[i].terminated && prim(r,Θ[i])) {
14          Θ[i].primary := true
15          Θ_prim.append(Θ[i])
16          if(++n ≥ n(r)) # found enough threads
17             break
18       }
19    }
20    if(highestThreadNo < i) {
21       highestThreadNo := i
22    }
23 }
```

Within the loop in `startRound()`, the currently executing thread first waits until enough threads are present in Θ, so the list of primaries can be filled. Only once enough threads are present in the system can the round truly start.

Defining Total Orders

For each round, a total order has to be defined, which specifies the order in which threads may perform critical actions they request. This order may change for each round, but once defined for a round, it stays fixed for this round[1]. `steps(r,`θ_i`)` is used to denote the definition of the number and order of possible critical operations thread θ_i may take within round `r`. This number can differ from thread to thread, but is again immutable once defined for a round. For example, a total order of the form `[0,0,1,0,1]` defines

1 We will extend this requirement later on to accommodate for explicit thread creation

both the order in which critical actions can be performed, and the number of steps per thread, which is three for Θ_{prim}`[0]` and two for Θ_{prim}`[1]`. `highestThreadNo` is computed here to be used for explicit thread creation, as will be explained soon.

Waiting for Turns

When a primary thread wants to perform a critical action, it has to wait for its turn first. To be more precise, this waiting is achieved via one of UDS' central mechanisms, `waitForTurn`, which blocks a thread

- when other primaries are prior within the total order of the current round,
- when the current thread has already exhausted its steps of the current total order, or
- before any critical action if it is not yet primary.

As this operation is part of the configuration space of UDS, we postpone its possible definitions to Section 7.2. The operation sets a variable `finished` per thread to `true` if this thread has reached the round's end.

Acquiring and Releasing Mutexes

Mutex acquisition can either mean immediate locking in case the mutex is free, or assignment of the thread to the per-mutex wait queue, in case the mutex is not available at the moment. Unlocking releases the mutex or passes it directly to the next thread within the queue if present.

```
24 Mutex::lock(m) := {  # lock mutex m
25    waitForTurn()  # obey the total order
26    θ := currentThread
27    if(m.owner ≠ nil) { # mutex occupied
28       m.enqueue(θ)
29       θ.enqueued := m
30       checkForEndOfRound()
31       waitFor(m.owner == θ || θ.enqueued == nil)
32       if(m.owner ≠ θ)
33          lock(m)  # repeat
34    } else {  # mutex free
35       m.owner := θ
36    }
37    progress := true
38 }
```

```
39 Mutex::unlock(m) := {  # unlock mutex m
40    next := m.dequeueFirst()
41    m.owner := next
42    if(next ≠ nil) {
43       next.enqueued := nil  # signal
44    }
45 }
```

Note that if the application's high-level mutex API is used correctly (a requirement in our system model), a `lock()` can be followed by either another

`lock()`, or has to be followed by the pertaining `unlock()`, the latter of which is therefore also naturally only ever executed within the current round, i.e., by primaries.

Thread Termination

In order to terminate, a thread also has to become primary first. Otherwise, termination could introduce indeterminism as it could happen either while the thread is primary or while it is non-primary, depending on the progress of this thread in different replicas. This could, for example, change the deterministic selection of threads for the next round between replicas (which naturally always excludes terminated threads). If a thread terminates, it is marked accordingly and its future turns in the total order need to be skipped by removing potentially remaining steps of this thread (cf. Line 54).

```
46 Thread::terminate := {  # self-termination
47    if(round == 0) {
48       startRound()  # bootstrap
49    }
50    progress := true
51    θ := currentThread
52    waitFor(θ.primary)
53    θ.terminated := true
54    removeFromOrder(θ)
55    checkForEndOfRound()
56    θ.stop()
57 }
```

Removing a thread from the current total order of steps is simple, as long as we remember to signal the next thread in line to wake up and consume its step.

```
58 removeFromOrder(θ) := {
59    totalOrder.removeAll(Θ_prim.getIndexOf(θ))
60    if(totalOrder.length > 0) {
61       Θ_prim[totalOrder[0]].waitingForTurn := false
62    }
63 }
```

Explicit Thread Creation

Since explicit thread creation also modifies Θ, it is relevant to determinism and necessarily a critical operation. The newly created thread can immediately start executing, but may not yet acquire mutexes. As Θ is totally ordered, the new thread has to somehow be enqueued within this order in a deterministic fashion. This is where we can use the previously computed `highestThreadNo` from Line 21, which signifies the highest index of the thread in Θ which is guaranteed to be deterministically known by all replicas. Since all threads up to Θ`[highestThreadNo]` are deterministically known, we can define the configurable deterministic function `threadPosition()` to insert a newly created thread. A further design decision is whether we let the newly created thread immediately run as primary or not. The method `newAsPrimary()` can decide this. Both functions' possible configurations are discussed

in Section 7.2. For now, we can simply assume that new threads are inserted behind the last current primary, and that they do not immediately gain primary status themselves.

```
64 createThread := {  # explicit thread creation
65    waitForTurn()  # obey the total order
66    θ_new := new thread
67    Θ.insertAt(θ_new, threadPosition(round))
68    highestThreadNo++
69    θ_new.primary := newAsPrimary()
70 }
```

Ending a Round

Once there are no runnable threads left in Θ_{prim}, the current round ends. A primary thread is considered not runnable if

- it is suspended because it has already consumed its allotted number of critical actions within the total order of `r`,
- it can not acquire its desired mutex because that is still locked by another thread, or
- it has already terminated.

Once a round ends, some primaries may still be enqueued in per-mutex wait queues. Enqueued primaries are simply dequeued and have to attempt to acquire their mutexes again (cf. Line 33). `checkForEndOfRound()` tests for the all of these conditions, and in case a round has ended, performs the required clean-up operations before starting a new round.

```
71 checkForEndOfRound := {  # detect end of a round
72    if(|Θ_prim| < n(r)
73       return  # not at end of round
74    foreach(θ ∈ Θ_prim) {
75       if( θ.enqueued == nil &&
76          !θ.terminated &&
77          !θ.finished &&
78          !θ.waitingForTurn ) {
79          return  # not at end of round
80       }
81    }
82    # end of round detected
83    foreach(θ ∈ Θ_prim) {
84       θ.primary := false
85       θ.finished := false
86       θ.waitingForTurn := false
87       if(θ.enqueued ≠ nil) {
88          θ.enqueued.remove(θ)
89          θ.enqueued := nil
90       }
91    }
92    Θ_prim := []
```

```
93    startRound()
94 }
```

When beginning a new round, we include the opportunity for deterministic reconfiguration with another configurable function `reconfig()` Line 9. Within this function, new configurations for `n(r+1)` and `prim(r+1, θ)` can be defined for `r+1`. The same applies for the definition of the total order, `steps(r+1, θ)`.

Deadlocks & Livelocks

Note that if the application is deadlock-free, UDS will not introduce any new deadlocks[2], because UDS simply follows a schedule any other regular scheduler could have followed as well by chance. However, it is possible for UDS to introduce livelocks, e.g., when repeatedly selecting primaries which immediately terminate their round because they can not currently acquire their mutexes. To avoid this, `prim()` and/or `reconfig()` can be modified, e.g., to increase `n(r+1)` if previous primaries did not make progress (hence the initial introduction of the pertaining variable in Line 5).

7.2 UDS DESIGN SPACE

UDS embraces the idea of reconfigurability during runtime. In the previous subsection, we have detailed the basic algorithm, while hinting at several ways to adapt UDS to one's own needs.

The design space of UDS consists mainly of the different possible implementations of `prim(r, θ)`, `n(r)`, and `waitForTurn()`, which defines the total order, as well as `steps(r, θ)`, `threadPosition()`, and `newAsPrimary()`.

In a minimal version of `prim(r, θ)`, it simply always returns true:

```
95 prim(r,θ) := { # define primaries
96   return true
97 }
```

However, especially when `n(r)=1`, it could make sense to select primaries that are able to immediately acquire their desired mutex, e.g., because it is currently available. Otherwise, the round would be over immediately, and we have wasted scheduling time for nothing.

Primaries and Performance

We have already introduced the idea that the number of primaries per round can be highly influential on the performance of the system. Usually, the larger `n(r)`, the more threads can run concurrently and hopefully in parallel on a multicore system. Having too many primaries, however, increases the risk of round-filling delays. In the worst case, RFDs can cause a system to stall completely at Line 12. The mitigation of this problem will be a core topic of a later part of this thesis (cf. Part IV).

In extreme cases, UEBs can also cause a system to stall indefinitely. If, for example, a request thread would spawn a new long-running thread, e.g., act-

2 See also the brief discussion on this in Section 8.2.

ing as a daemon, this thread would get inserted into Θ and could at some point be selected as a primary if `prim()` allows it. However, if this daemon thread is meant to only execute in the background without ever performing a critical action, once a round with this thread starts it would never end. A way to mitigate this problem is to force long-running or daemon-threads to periodically `yield()` as follows, permitting the current round to end.

```
98  yield() := { # no-op for scheduling
99     waitForTurn()   # obey the total order
100 }
```

Batching and UDS

The mode of operation of the atomic broadcast protocol responsible for ordering requests can be another influence on the design of UDS. In many practical systems nowadays ordering is decided on in batches, i.e., multiple ordered requests are delivered to the execution pipeline at once whenever an ordering decision has been agreed upon. If the scheduler knows the size of a delivered batch and has deterministically seen the first request of a batch, it knows the rest of the requests of that batch and can deduce that all other replicas will have seen the entire batch, too. This would allow for `n(r)` to be temporarily increased without incurring RFDs.

Let us finally review the creation of total orders of steps for new rounds. We will be giving a few examples of common possible configurations, but note that the possibilities to create configurations tailored to specific applications are manifold. First, data structures for the total order and number of primaries are required.

```
101 totalOrder := []
102 N := 0
```

Next, the following stub of `reconfig()` serves as a demonstration of how different configurations of UDS could be implemented to imitate PDS or MAT.

```
103 reconfig(r, prog) := {  # reconfigure for a new round
104    N := 2  # as an example
105    totalOrder := [0,1]  # PDS-1
106    totalOrder := [0, 1, 0, 1]  # PDS-2
107
108    N : = 1  # as an example
109    totalOrder := [0, 0, 0, 0, ...]  # MAT
110
111    if(prog == false) {
112       # increase N compared to last round
113       # or make sure other threads are used
114       # by reconfiguring prim()
115    }
116 }
```

Since we simply set N here, a trivial version of `n(r)` could be:

```
117 n(r) := {  # number of primaries
118    return N;
119 }
```

WaitForTurn()

A core missing piece of the puzzle is the functionality within `waitForTurn()`, which forces threads to obey the designated total order and acts as the main mechanism guaranteeing determinism. The following listing shows a possible implementation which enforces total orders by allowing threads that have become primary to remove steps from the current total order one after another:

```
120 waitForTurn() := {  # enforce total order
121    if(round == 0) {
122       progress := true
123       startRound()  # bootstrap
124    }
125    θ := currentThread
126    while(true) {
127       waitFor(θ.primary)
128       i := Θ_prim.getIndexOf(θ);
129       if(!totalOrder.contains(i)) {
130          # no more steps
131          θ.finished := true
132          checkForEndOfRound()
133          waitFor(θ.finished = false)
134       } elsif(θ == Θ_prim[totalOrder[0]]) {
135          # next in the order
136          totalOrder.removeFirst();
137          if( totalOrder.length > 0 ) {
138             # Wake up next
139             Θ_prim[totalOrder[0]].waitingForTurn := false
140          }
141          break
142       } else {
143          # waiting for being first in the order
144          θ.waitingForTurn := true
145          checkForEndOfRound()
146          waitFor(θ.waitingForTurn == false)
147       }
148    }
149 }
```

Note that these examples for `reconfig()` and `n()` are merely demonstrations, in this case of configurations that would cause UDS to mimic PDS-1, PDS-2, or MAT. However, the number and total order of steps can be finely tuned to specific needs of the application being replicated. In other words, if threads executing requests of a particular application happen to always adhere to a fixed pattern of locking and unlocking the same mutexes, it would be a good idea to adapt `reconfig()` to provide the exact number of steps

needed for a thread to finish within one round, instead of keeping threads in the system for multiple rounds. In addition, if, in this example, multiple types of requests were to exist, where one request type requires far longer execution times than others, it might be a good idea to make sure these requests end up in the same rounds so as to avoid UEBs. Further, UDS can be tuned to optimize for different goals, i.e., response latency, throughput, execution resource efficiency, etc.

Placing Explicitly Created Threads

For explicit thread creation and its configuration options `threadPosition()` and `newAsPrimary()`, the following would be a simple example for inserting the new thread as non-primary into Θ, just behind the last thread deterministically seen by the scheduler:

```
150 threadPosition := { # where to place new thread
151   return highestThreadNo
152 }
```

```
153 newAsPrimary := { # new thread as primary
154   return false
155 }
```

Additional options would be to place new threads behind the last primary of the current round, or to put it in front of all threads, depending on the desired execution characteristics of the system. For example, in systems with live request processing for clients, i.e., with continuous input during runtime, it might be preferable to process old requests firsts.

Primary Status of Explicitly Created Threads

When deciding whether to immediately promote created threads to primary status, it is important to consider the configuration of `reconfig()`, i.e., the method generating new total orders of steps. If generated total orders can also include steps for new threads, they can be made primary immediately. Otherwise, this would not have any benefit. Note that the functions `threadPosition()` and `newAsPrimary()` may or may not depend on round numbers. They could even deterministically change in a sense that they behave differently for every other thread creation.

Most of the reconfiguration decisions made by the scheduler need to be deterministic. Since all threads but one (the last one closing the round) are suspended at the beginning of a new round, this thread can safely access the complete scheduler state S using e.g., `reconfig()` (cf. Line 9). Hence, whenever we would like to dynamically adapt UDS to changing system conditions during runtime, as in upcoming chapters, we can safely use `reconfig()` to do so.

7.3 COMPARISON TO STATIC SCHEDULERS

Mimicking Previous Work

As we have seen, UDS is significantly more flexible than previous scheduling solutions. We will round off our introduction of UDS by briefly comparing it

to the related work to show potential weak points in UDS besides its considerable strengths. One of the strong points of UDS is of course its configurability, which we demonstrate in the following comparisons by specifying UDS configurations that imitate the behavior of the previously published scheduling solutions found in related work Section 3.3.

7.3.1 *MAT*

Emulating MAT

In MAT [71], the number of primaries always equals one: `n(r)`:= 1. This primary is allowed to take an arbitrary number of steps, and thus the round lasts as long as this thread can run. The total order is trivial, giving the single thread an unlimited number of steps. `prim(r,` θ`)` excludes threads currently waiting for an unavailable mutex. With this configuration, UDS emulates MAT, i.e., the causal order of mutex locking would be identical.

However, in contrast to UDS, MAT was designed from the beginning to work with reentrant locks, nested invocations and condition variables, i.e., `wait` and `notify` operations in Java. While reentrant locks could easily be implemented by wrapping `lock()` and `unlock()` and maintaining lock counters, UDS in its presented form can not currently deal with nested invocations and condition variables. As the integration of responses to these invocations would require coordination between replicas, we would leave the domain of fully deterministic systems with nested invocations. We have therefore not continued development of UDS in this regard, although some preliminary work required for making UDS fit for nested invocations has been completed and will have to be presented in future work.

In the Aspectix Deterministic Scheduler Suite (ADETS) developed by Hauck et al. in the years prior to the beginning of this thesis, MAT was included in a version capable of thread creation during runtime, which it was incapable of supporting in its original version. ADETS-MAT allows thread creation by inserting the new thread as non-primary after the position of the last known thread in Θ, which resembles the code in Lines 150 ff. [38].

7.3.2 *PDS-1 and PDS-2*

Emulating PDS

PDS introduced the notion of scheduling rounds used in UDS. It has a configurable, but fixed number of primaries for each round once the system starts, and the predicate `prim()` will simply always return true.

PDS comes in two flavors named *PDS-1* and *PDS-2* [17], later renamed to *simple PDS* and *PDS* [19][3]. The difference between PDS-1 and PDS-2 is the number of critical actions each thread may perform per round, so it is identical to UDS' total order of steps created each round. PDS-2 uses a round-robin scheme when assigning two critical actions to each thread. This is easily emu-

3 We will mostly continue using *PDS-1* and *PDS-2* in this thesis due to the nice overlap of these names with our concept of steps.

lated by UDS; in fact, in the code shown above, we have included the respective orders required to make UDS behave like PDS.

With PDS, threads can explicitly create new threads, which is a critical operation. New threads start as non-primary, and are inserted into the list of all threads directly after the last primary.

Note that in their first paper on PDS, the authors claimed that more than two steps would potentially lead to race conditions. With UDS we show that an arbitrary number of steps is possible.

7.3.3 *Kendo*

Kendo as a Special Case

Kendo [68] resides between the application and the operating system, and has been designed to ease the testing and debugging of concurrent stand-alone applications. It therefore does not support implicit thread creation for incoming requests during runtime, i.e., it can not support interactive or continuous input, as mentioned before.

In Kendo, $\Theta_{prim} := \Theta$ and `prim()` is defined as in Line 95, so simply put, all threads are primary. In addition, the application runs in a single round which never ends, except when it bugs itself into a deadlock or terminates execution. Each thread maintains an individual logical clock counting arbitrary but deterministic events. In an implementation, this could be realized by using certain CPU instruction counters, or at the very least by counting mutex acquisitions. Mutex acquisition order is based on these logical clocks in the sense that acquisitions have to occur with increasing clock values, where ties are broken by thread IDs. Threads may perform critical actions when all acquisitions with a lower clock count have happened, which in practice means a thread has to wait until it has the lowest clock count among all threads. Therefore, `waitForTurn()` is very different to the one presented earlier:

```
156 waitForTurnKendo() := {  # Enforce total order
157     if(round == 0)
158         startRound()  # Bootstrap
159     θ := currentThread
160     waitFor(θ.clock = minOfAllClocks())
161 }
```

Note that in addition to this, multiple other places in the algorithm would have be changed to accommodate logical clock counters.

If we recall a paragraph from Section 6.1.2 on total vs. causal ordering, UDS does not necessarily enforce total ordering of mutex acquisitions across replicas. Rather, mutex *assignments* are totally ordered, which results in a causal ordering of the actual acquisitions. Kendo does not adhere to this idea. Any thread wanting to `lock()` a mutex has to wait for the corresponding `unlock()` of this mutex, all while keeping its logical clock up to date. Then, it has to compete for the mutex by using its clock value, which is both resource-

consuming and the reason why UDS can only imitate Kendo, but will not always lead to the exact same causal ordering of lock acquisitions.

Kendo & SMR

We have hinted earlier that Kendo does not support replicated state machine systems, as well as explicit thread creation during runtime. Using our developed notation and sense for how deterministic scheduling works, we can now explain this in greater detail. Since all threads in Kendo are primary, any newly created thread, regardless of whether it is created explicitly (by a thread running a request) or implicitly (in response to a newly delivered request) is also part of Θ_{prim}, and it has to be inserted into Kendo's total order. Since Kendo has no notion of scheduling rounds, there are no easily discernible deterministic points in time in which we could decide on the exact location in the total order, i.e., the clock value of the new thread. Additionally, if we simply assume that input to Kendo is delivered in a deterministic way, the thread handling this input by starting new request threads has to have the lowest clock value in order to perform this critical operation. In other words, in order to accept input, the receiving thread inside the system blocks all other threads from making progress when it has the lowest clock value while waiting for input. We could try to mitigate this problem by adding a large fixed number to the clock value of the input thread whenever new input arrives, but we can never be sure how large this value should be when we want to be certain that other threads don't starve, while also not preventing the system from being capable of accepting new input.

It is, however, possible to create a configuration *similar* to Kendo, in which a round would end after a previously defined number of critical actions or when all threads have terminated. At this point, new threads could be added to the system.

7.3.4 *CoreDet*

Emulating CoreDET's Scheduler

The CoreDet authors discuss several strategies for achieving determinism: DMP-O, DMP-B and DMP-BP (*Deterministic shared memory multiprocessing* with *O*wnership tracking, Store *B*uffering, and *P*artial *B*uffering, respectively[4]). For each of their three implemented parallelism approaches, different critical actions are defined down to single read and write operations to memory or thread creation and mutex locking. Each strategy employs the same deterministic scheduler which lets threads run in parallel for a quantum of instructions as long as no communication between threads is attempted, followed by a serial phase in which all threads' operations are ordered sequentially. Both phases together constitute one scheduling round. CoreDet deterministically defines the length of its parallel phase by counting instructions utilizing their modified compiler, and each thread is allowed to either run for exactly the allotted number of instructions or until an instruction is reached that needs to communicate with other threads. Afterwards, the serial phase starts, in

4 The paper also mentions DMP-TM (*T*ransactional *M*emory), but does not expand on it due to several difficulties with implementing the approach, but UDS cannot be easily compared to transactional memory systems anyway

which all threads work on their critical operations sequentially, before the next round starts with its parallel phase. The authors of CoreDet tested two versions of their scheduler, modifying the length of the serial phase. In the first version, only the very next mutex release in each thread is completed before the next round starts, while in the second version each thread runs until the entirety of its remaining instruction budget (its quantum) that has not been used in the parallel phase is used up.

Provided we let UDS handle the same critical actions as defined by the respective strategy, and assuming that threads always reach a critical action within their allotted number of instructions, we can imitate this scheduler using a UDS configuration with the following parameters: $\Theta_{prim} := \Theta$ and `prim()` is defined as in Line 95. In other words, while UDS can not count instructions by utilizing compile-time code inserts, we can assume that scheduled threads will either perform a critical action at some point and that the parallel phase then looks very similar to CoreDet's PDS-1-like scheduling, or that threads simply run in parallel and finish without attempting a critical action anyway. Since threads can only run in parallel until their first critical operation, and afterwards all critical operations are sequentialized, this is the same outcome as UDS with a round-robin total order with ascending thread IDs, giving each thread exactly one critical action per round. Explicit thread creation is a critical action and new threads are appended to the thread list as non-primaries.

Note that CoreDet is an example of a platform with completely different critical operations than we used in our system model so far, which may include even multiple instructions and sub-operations. Additionally, it is noteworthy that CoreDet needs to treat an `unlock()` as critical operation so that a mutex will not be passed to another thread inside a single round, which can introduce indeterminism without per-mutex wait-queues as introduced by UDS.

7.3.5 *DThreads*

Emulating DThread's Scheduler

DThreads similarly switches between parallel and serial phases per scheduling round, and the UDS configuration for DThreads is the same as for CoreDet. Communication between threads (i.e., making visible the changes performed during the parallel phase) may only occur during the serial phase, strictly controlled by a token guaranteeing a globally observable total order. This is equivalent to UDS' total order with steps for each thread, where each thread gets one step per round. Critical actions in DThreads are mutex `lock()` and `unlock()`, as well as thread creation and termination.

7.3.6 *Parrot*

Emulating Parrot's Scheduler

As described in Section 3.3, Parrot is intended primarily for development and debugging scenarios and argues that purely focusing on determinism is not desirable for testing multithreaded software. Therefore, it gives developers the opportunity to add performance hints to their code, which optimize execution

in a trade-off against determinism. UDS can not deterministically emulate executions utilizing these performance hints, but the basic deterministic executions enforced by Parrot for unoptimized code is simply achieved by configuring UDS for round-robin scheduling akin to PDS-1, CoreDet or DThreads. Newly created threads are inserted at the end of the current primaries.

7.3.7 *Comparison*

In this section, we have shown UDS to be capable of emulating or imitating numerous scheduling algorithms and execution platforms for fully, weakly deterministic systems. Table 7.1 summarizes the important configuration parameters for each discussed algorithm at a glance.

UDS Configuration Space

The configuration space of UDS is much larger than the space of any of the existing algorithms as apparent by the different possible implementations of `prim(r, θ)`, `n(r)`, `waitForTurn()`, `steps(r, θ)`, `threadPosition`, and `newAsPrimary()`. Additionally, CoreDet and DThreads cannot benefit from the distinction between total and causal ordering we explained in Section 6.1.2, as they have to consider an `unlock()` as critical operation because of missing per-mutex wait-queues. On the other hand, these platforms can achieve strong determinism, whereas UDS is primarily focused on weak determinism.

Most importantly, however, UDS allows for deterministic adaptation to current system conditions during runtime, since its configuration can be changed from one round to the next, whereas all previous algorithms use the same static configuration after startup.

	Emulated Scheduler					
Parameter	MAT	PDS	Kendo	CoreDet	DThreads	Parrot
`n(r)`	1	N	All Threads	All Threads	All Threads	All Threads
`prim(r, θ)`	θ's next Mutex avail.	True	True	True	True	True
No. of Rounds	∞	∞	1	∞	∞	∞
`steps(r, θ)`	∞	1 or 2	∞	1	1	1
Total Order	N.A.		Logical Clock and Thread ID			
Thread Creation	Non-Primary		Primary	Non-Primary		Primary
Thread Position	`highestThreadNo`	Behind Primaries	End of Primaries	Behind Primaries		End of Primaries

Table 7.1. UDS Configurations required to emulate existing scheduling algorithms or deterministic execution platforms

8 IMPLEMENTATION AND EVALUATION

After the definition and publishing of the core UDS algorithm in pseudocode, our next contribution was the undertaking of actually implementing and testing UDS. Given the huge configuration space of UDS and potentially hidden edge-cases and logical bugs in the algorithm itself, we opted to first simulate UDS using an event-based simulation framework. This allowed us to easily see schedules produced by UDS under different configurations, to get a feeling for UDS' mode of operation when implemented, and to identify possible logical errors. Afterwards, we implemented UDS in Java and tested it against simulated workloads, which further deepened our understanding of the algorithm and allowed us to further correct its behavior in certain scenarios. Finally, we embedded our UDS implementation into a modern SMR framework to perform end-to-end tests, using UDS for one of its main intended use cases: Deterministic runtime-reconfiguration for performance and efficiency optimizations of SMR systems. The following sections briefly describe how each of the steps mentioned above were completed, while presenting any results obtained during this phase of our research.

8.1 EVENT-BASED SIMULATION

Motivation

Whenever the temporal characteristics of a complex system have to be analyzed under varying conditions and configurations, an event-based simulation can be a helpful tool to understand the system's inner workings. We implemented UDS in the event-based simulation tool DESMO-J[1] with the help of two student assistants.

Goals and Features

The primary goal of implementing this simulated system was to find out how UDS behaves—i.e., which schedules it produces and whether it is bug-free and stable—under different configurations and load scenarios (i.e., request distributions), and to gauge the effects of different UDS schedules on system performance as measured by request latencies and overall throughput. Given that the goal was to get a clearer picture of how the UDS algorithm behaves in a real system, the simulation was set up to imitate the scheduling of threads on real CPU cores, using a simple preempting time-slice based system scheduler assigning processes to free CPU cores in a round-robin fashion, on top of which UDS can deterministically assign critical actions to threads using predetermined total orders. Additionally, accompanying Python scripts using matplotlib[2] were developed to visualize the simulation results in several plots of different types (e.g., time series plots showing individual request lifecycles, latencies per request, CPU core load per time-slice, among others). We named this simulation framework with its collection evaluation and visualization scripts *UDS-SIM*.

1 http://desmoj.sourceforge.net/home.html

2 https://matplotlib.org/stable/api/pyplot_summary.html

Capabilities

Runtime dynamicity, i.e., reconfiguration of UDS as discussed in Section 7.2, was not yet a part of UDS-SIM. However, reconfigurations between several distinct, shorter simulation runs while keeping the same overall system setup still allowed us to examine the impact a dynamically reconfigurable scheduler can have. In order to observe the effects different UDS configurations have under various application profiles, UDS-SIM was given the capability to generate multiple different requests with respective pre-specified sequences of locking and unlocking of mutexes supplied via configuration files. Additionally, it is possible to specify the probabilities and distributions of the different requests being generated, altogether providing a powerful and comprehensive way to simulate a wide variety of application behaviors and observe UDS' scheduling decisions under numerous conditions.

In the following paragraphs, we show the most relevant insights gained from UDS-SIM, in respect to some of the research questions stated in the beginning of the thesis. For each example, a short selection of a few of the important simulation configuration parameters required to achieve the presented results will be shown. The full configuration file format with explanatory comments on each available configuration option, as well as class diagrams of the simulator, can be found in the appendix (cf., Part VI).

8.1.1 *Demonstrating Round-filling Delays*

RFDs

Thanks to UDS-SIM, it is easy to directly show the impact RFDs can have on request latencies in case the system currently experiences low load, e.g., because not many clients are currently connected or actively sending requests.

The following configuration is just one example where RFDs can be clearly observed[3]:

```
# high primaries and low request rate → RFDs
# set up UDS; steps are irrelevant for this demo
NUM_PRIMARIES = 8
TOTAL_ORDER_SCHEDULING = ROUND_ROBIN

# request generator supplying 1 request every 20ms
NUM_REQUEST_AT_START = 1
GENERATOR_TYPE = NORMAL
GENERATOR_DISTRIBUTION = UNIFORM
GENERATOR_DISTRIBUTION_PARAMETERS = 20.0; 20.0; -1.0
GENERATOR_BATCH = 1

# simple request; quickly locks and unlocks a mutex,
# before performing a short calculation
```

3 Please note that due to the simulation project's early roots from the first months of our research, there is an unfortunate terminological overlap between *steps* as understood by UDS and *steps* as used in the simulator for specifying request actions. In UDS, a step is a conceptual token that is consumed by a primary from the current round's total order whenever a critical action is taken. In the simulator, a request step can also be a non-critical action like performing a calculation without accessing mutexes.

```
15 GENERALIZED_REQUEST = REQUEST_A
16 REQUEST_A_PROBABILITY = 1.0
17 REQUEST_A_VISUAL_NAME = ReqA
18 REQUEST_A_STEPS = 2
19 REQUEST_A_LOCK = 1, -1; 0.0
20 REQUEST_A_DISTRIBUTION = UNIFORM; UNIFORM
21 REQUEST_A_PARAMETERS = 4.0, 4.0; 12.0, 12.0
```

Running this simulation and visualizing its logged traces yields a time-series plot as seen in Figure 8.1 (where only the first few threads are shown for improved clarity of the figure), and a plot of latency per thread as shown in Figure 8.2.

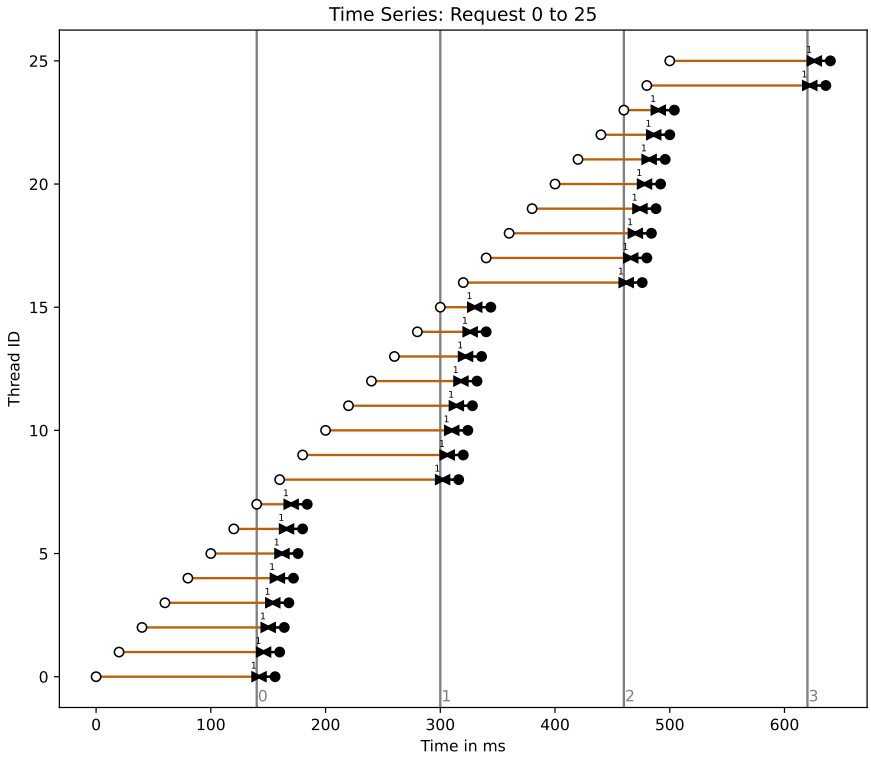

Figure 8.1. Time Series plot of thread-lifecycle for demonstrating RFDs, with UDS configured to `n()=8`. White circles depict the initial start of a thread, a red line signifies a thread is blocked and waiting, a right-pointing triangle marks the locking of a mutex (with the number above it showing the pertaining mutex ID), a left-pointing triangle stands for a mutex unlock, black lines show that a thread is executing, and black circles signify thread termination. The large vertical gray lines show round boundaries, with round numbers shown towards the bottom of the graph.

Looking at Figure 8.1, we can see how a round `r` can only start once `n(r)=8` requests have arrived in the system. For example, round 0 starts at precisely the moment 8th request is received, blocking the execution of all previously arrived threads for significant amounts of time. The same is true for subsequent rounds.

Impact of RFDs

It is noteworthy, however, that RFDs impact latency variability and not

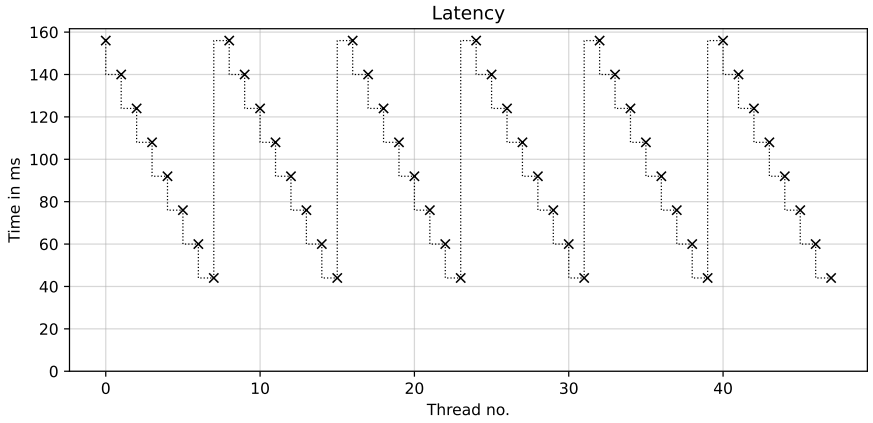

Figure 8.2. Latency per thread, i.e., time between thread start and thread termination, in a RFD-setting with UDS configured to `n()=8`, demonstrating the significant impact of RFDs on latency variability. The dotted line is only included to help with easier recognition of the request pattern.

throughput, since they occur only in low-load scenarios. In a user-facing system, for example, clients would notice highly variable latencies depending on their luck, i.e., which exact position their request happens to occupy in a round. This can clearly be seen in Figure 8.2.

Mitigation Strategies against RFDs

Simply changing the UDS configuration to `n()=1` mitigates RFDs completely. With all remaining configuration options identical to the previous simulation, the results are vastly different, as shown in Figure 8.3 and Figure 8.4.

In both figures for the configuration with 1 primary, we can see that the outcome is now optimal for this specific case, i.e., under this low-load scenario. All requests experience the same shortest possible latency given the specific request profile used in this simulation. Throughput stays the same, limited by the configured generation rate of the request generator.

8.1.2 *Demonstrating Unbalanced Execution Times*

UEBs

Recall that UEBs occur when there are multiple types of requests in an application, and the differences between required execution times to fulfill these requests differs significantly. Then, schedules which lump together short- and long-running threads into a single round will negatively affect performance due to prolonging rounds unnecessarily. However, reasoning about this on paper is different from actually seeing it happening in a real, albeit simulated, system. It is easy to demonstrate UEBs in UDS-SIM, and Figure 8.5 shows the results of a run with the following simulation configuration:

```
# requests of different lengths and 2+ primaries → UEBs
# 2 primaries, 1 step; suffices to demonstrate UEBs
NUM_PRIMARIES = 2
NUM_ROUNDSTEPS = 1
TOTAL_ORDER_SCHEDULING = ROUND_ROBIN

```

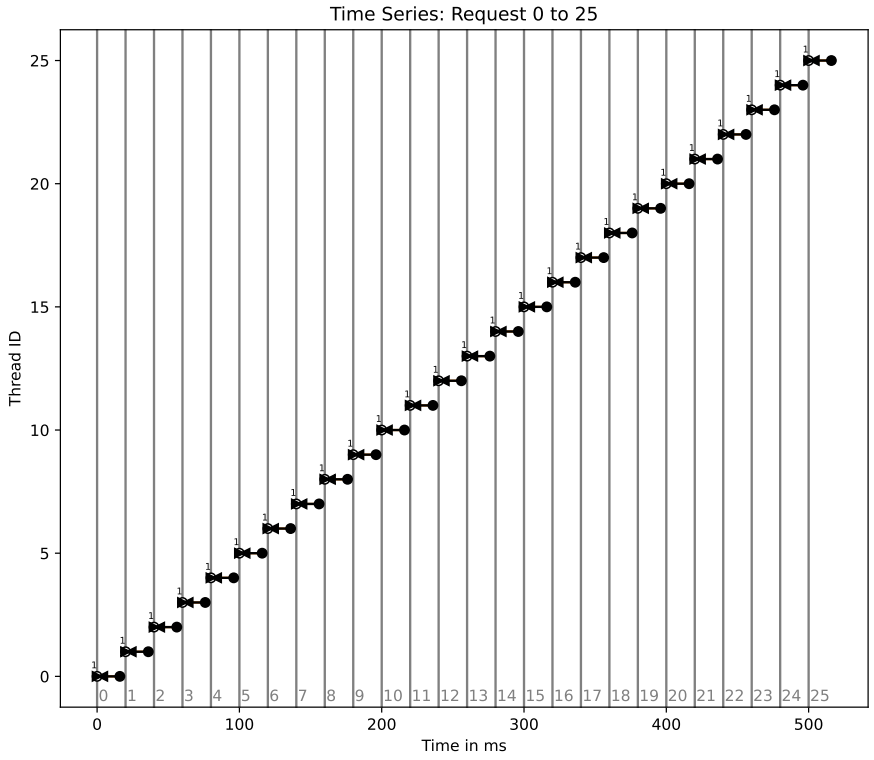

Figure 8.3. Time Series plot of thread-lifecycle for demonstrating mitigation against RFDs by configuring UDS to `n()=1`. Legend identical to Figure 8.1

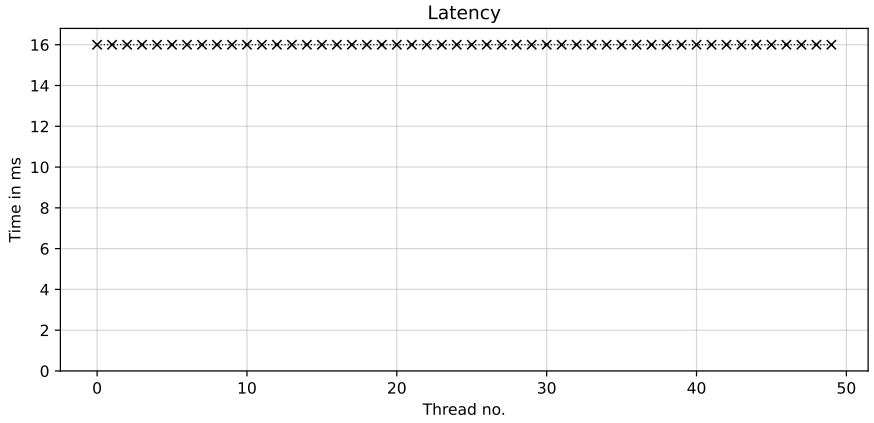

Figure 8.4. Latency per thread after mitigating RFD by configuring UDS to `n()=1`. Legend identical to Figure 8.2

```
# generate a batch of 2 requests every 15ms
NUM_REQUEST_AT_START = 2
GENERATOR_TYPE = NORMAL
GENERATOR_DISTRIBUTION = UNIFORM
GENERATOR_DISTRIBUTION_PARAMETERS = 15.0; 15.0; -1.0
GENERATOR_BATCH = 2

# define 2 requests
GENERALIZED_REQUEST = REQUEST_A; REQUEST_B

# short request A; quickly locks and unlocks a mutex,
# before performing a short calculation
REQUEST_A_PROBABILITY = 0.75
REQUEST_A_VISUAL_NAME = ReqA
REQUEST_A_STEPS = 2
REQUEST_A_LOCK = 1, -1; 0.0
REQUEST_A_DISTRIBUTION = UNIFORM; UNIFORM
REQUEST_A_PARAMETERS = 4.0, 4.0; 9.0, 9.0

# request B: lock, unlock, longer calculation
REQUEST_B_PROBABILITY = 0.25
REQUEST_B_VISUAL_NAME = ReqB
REQUEST_B_STEPS = 2
REQUEST_B_LOCK = 2, -2; 0.0
REQUEST_B_DISTRIBUTION = UNIFORM; UNIFORM
REQUEST_B_PARAMETERS = 4.0, 4.0; 20.0, 20.0
```

Analyzing UEBs

We can immediately see unused CPU time before the beginning of round `r2`, where both new threads have to wait until the long-running thread `t2` finishes. A more intelligent schedule in this case could —theoretically— reconfigure UDS to `n(1)=1` and modify `prim(1,`θ`)` so that it looks through θ and prefers to pick shorter-running threads when primaries are configured to 1. In that case, only `t3` would be picked for `r1`, which would finish over 10ms earlier than without reconfiguration. At the point in time when this `r1` ends, the two new threads `t4` and `t5` would have arrived in the system, so that θ `= [t2,t4,t5]`. Since we now have two threads of type `B` in the system (`t2` and `t4`), we could reconfigure again to `n(2)=2`, and select both long-running threads for `r2`. Afterwards, we would have enough threads of type `A` in the system to safely increase primaries to 4 or even 5, parallelizing all of these executions.

Mitigation Difficulties for UEBs

However, it is of utmost importance to note that all the above is far more difficult to determine and decide than the mitigation strategy for RFDs, which can work by simply setting `n()=1` when a low-load phase is detected somehow. Looking at a simulation trace afterwards and speculating about optimal schedules with the power of hindsight and global knowledge of the system is of course completely unfair when compared with the knowledge an optimization agent has at its disposal during runtime. Solving the hard problem of *deterministically* detecting and reacting to certain conditions in the system to

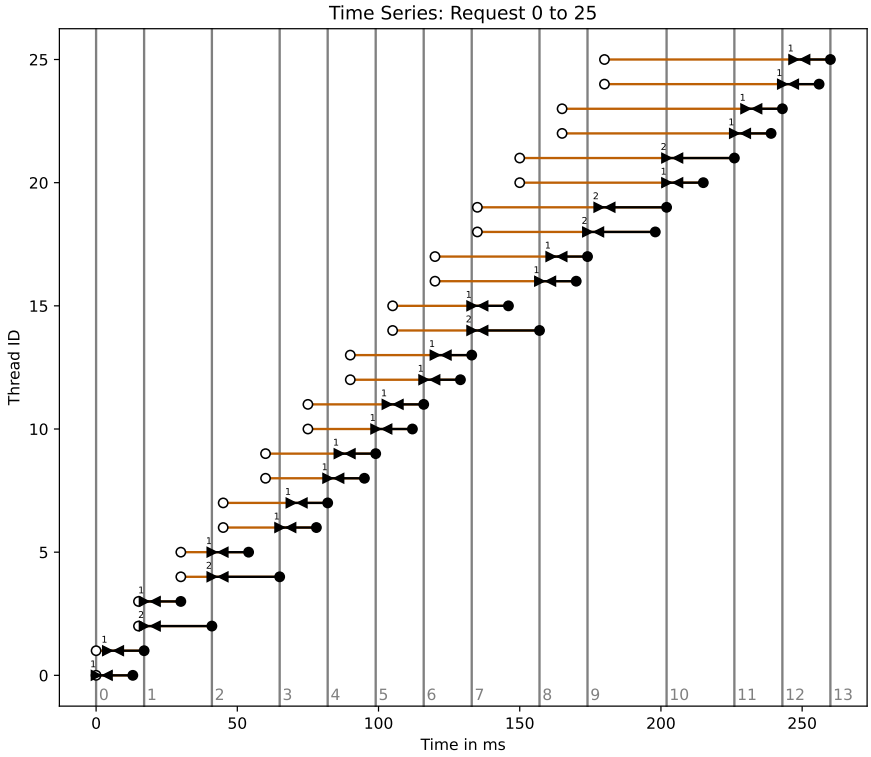

Figure 8.5. A demonstration of how UEBs can affect a system negatively if schedules aren't optimized perfectly. Legend identical to Figure 8.1

intelligently reconfigure UDS is one of the primary goals of this thesis and will be further discussed in upcoming chapters.

For now, however, the simulations we were able to run and visualize using UDS-SIM helped us understand how UDS produces schedules under different configurations, and gave us inklings of how future strategies of optimizing these schedules could look.

8.1.3 *Investigating Performance Effects*

Motivation

These carefully crafted demonstrations of micro-optimizations in very specific environments, where requests always take exactly the same time and incoming request rates are uniform, only show a part of the picture, however. In addition to understanding how UDS affects systems on a micro-level, we wanted to know how UDS' configuration parameters generally affect overall system performance, specifically throughput and request latency distribution, when measured over longer periods of time and with more realistic request distributions and load variations.

For this purpose, we included automatic thread generation strategies with UDS-SIM, which attempt to find the maximum supported throughput before latency spikes, given a UDS configuration and predefined request profiles. In this subsection, a short selection of simulation runs, demonstrating the pro-

found effects a simple UDS parameter can have on the performance of systems, will be shown.

Primaries as a Primary Driver of Performance

Among all the adjustable parameters in the UDS design space, the number of primaries per round may be the one with the biggest singular impact on performance, especially in high load scenarios and on systems with many CPU cores. This is conceptually rather easy to understand, since more primaries means an overall greater chance for parallelism within a round. The drawbacks of setting primaries too high when there is not enough load on a system were already shown in previous sections (cf. RFDs), but so far no empiric justification for high primary counts under high load were presented. Consider therefore the following simulation setup, where we specify a system with 8 CPU cores to have some headroom for parallelization, and show what happens to maximum throughput when scaling primaries from 1 through 20.

```
# change between runs, starting from 1 through to 20
NUM_PRIMARIES = 1
# 8 cores to allow for nicely parallelized execution
NUM_PROCESSORS = 8

TOTAL_ORDER_SCHEDULING = ROUND_ROBIN

# automate thread generation; other params irrelevant
GENERATOR_TYPE = MAXTESTING3

# three types of request
GENERALIZED_REQUEST = REQUEST_A; REQUEST_B; REQUEST_C

# Request A
REQUEST_A_PROBABILITY = 0.4
REQUEST_A_VISUAL_NAME = ReqA
REQUEST_A_STEPS = 2
REQUEST_A_LOCK = 1, -1; 0.0
REQUEST_A_DISTRIBUTION = NORMAL; NORMAL
REQUEST_A_PARAMETERS = 5.0, 2.0; 10.0, 3.0

# Request B
REQUEST_B_PROBABILITY = 0.3
REQUEST_B_VISUAL_NAME = ReqB
REQUEST_B_STEPS = 2
REQUEST_B_LOCK = 0.0; 2, -2
REQUEST_B_DISTRIBUTION = NORMAL; NORMAL
REQUEST_B_PARAMETERS = 10.0, 2.0; 3.0, 2.0

# Request C
REQUEST_C_PROBABILITY = 0.3
REQUEST_C_VISUAL_NAME = ReqC
REQUEST_C_STEPS = 3
REQUEST_C_LOCK = 3, -3; 0.0; 1, -1
REQUEST_C_DISTRIBUTION = NORMAL; NORMAL; NORMAL
REQUEST_C_PARAMETERS = 10.0, 4.0; 5.0, 2.0; 4.0, 1.0
```

Three request types are specified, to simulate what the profile of a small application exposing different actions to clients might look like. Each request type has at least one critical action (locking a distinct mutex), and takes different amounts of time to simulate calculations in between. `REQUEST_C` additionally has a second critical action where it locks the mutex `REQUEST_A` requires at the beginning of its executions to interweave requests a little more than without any mutually required locks.

Simulation Setup

The request generator is set up to use one of the automatic strategies of finding a viable maximum throughput. It does this by progressively raising the throughput across a number of simulation episodes, where in each episode several thousand threads are generated, while measuring the averages and standard deviations of observed request latencies per episode. Once request latencies begin spiking upwards, it terminates the attempts to raise throughput, and looking back through previous episodes tries to find an episode where throughput was high while latencies had not yet spiked. In detail, it does this by calculating an internal score for each episode, using the simple formula, where *throughput* is the number of requests per second calculated over the entire episode, and *latencies* are all individual request latencies logged during the episode[4]:

$$score = \left(\frac{1}{\texttt{mean}(latencies)} + \frac{1}{\texttt{sd}(latencies)} \right) * throughput^2 \qquad (8.1)$$

Starting UDS-SIM with this configuration will run an automatic process of finding the maximum supported throughput rate in requests per second for the given simulation parameters. One such run (for 8 primaries, i.e., with `NUM_PRIMARIES = 8`) is shown in Figure 8.6.

Every second episode (for better readability) is labeled with its ID. The error bars show the standard deviation of the latency for this episode, and we can see that at a certain point, after raising throughput enough, latencies spike quite dramatically. The chosen maximum sustainable throughput rate is marked in red, while the other episodes of this simulation run are not specially colored.

Results

After running 20 of these simulation runs, incrementing `NUM_PRIMARIES` from 1 through 20 from run to run, the resulting data can be plotted in a summary graph as seen in Figure 8.7.

Takeaways

The important takeaways from these plots, as well as from several other similar simulations with different settings —which are not shown here— are i) scaling primaries up and down can significantly affect system throughput given the right circumstances, ii) setting primaries very high incurs large deviations in the average request completion latencies, and iii) a generally sound sweet spot for `n()`, both providing high throughput and acceptable latency variation during high system loads, can usually be expected at or around the number of CPU cores the system has available. For example, in this setting, `n()=8`,

4 This formula is not special in any way, and was simply landed on after some experimentation. As long as the formula is identical between simulation runs with different primary configurations, however, the exact formula does not matter all that much for the purposes of this demonstration

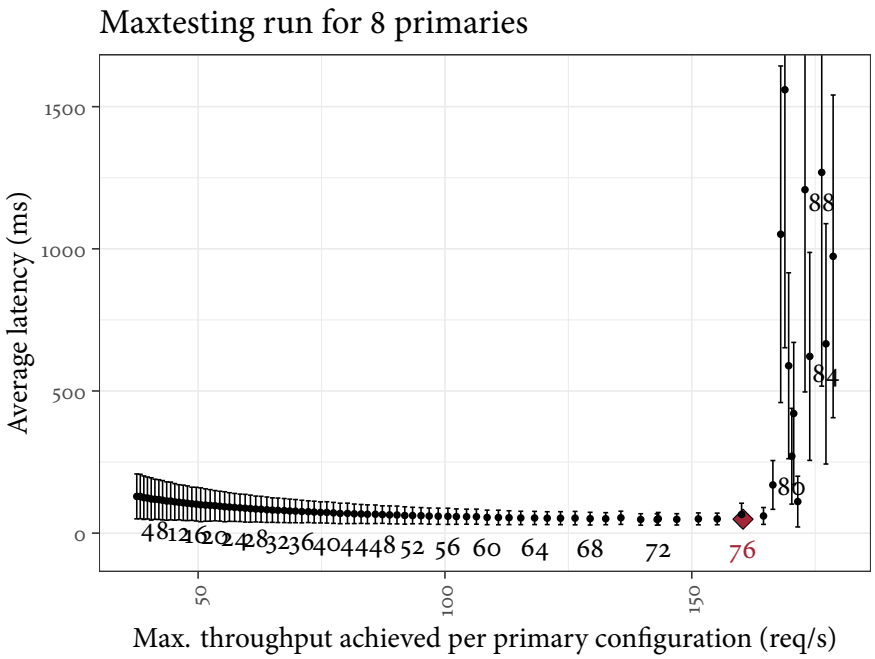

Figure 8.6. Single simulation run with `GENERATOR_TYPE` set to `MAXTESTING3` and `NUM_PRIMARIES = 8`. The plot shows the achieved throughputs in req/s for every episode, plotted against the average request latency during that episode, with error bars showing the standard deviation of latency around these averages. The upper ends of the error bars for several episodes towards the end of the run have been truncated for better readability of the plot. Similarly, only every 4th episode is labelled for improved readability. The episode with the maximum score as calculated by Equation (8.1) is marked in red (#76).

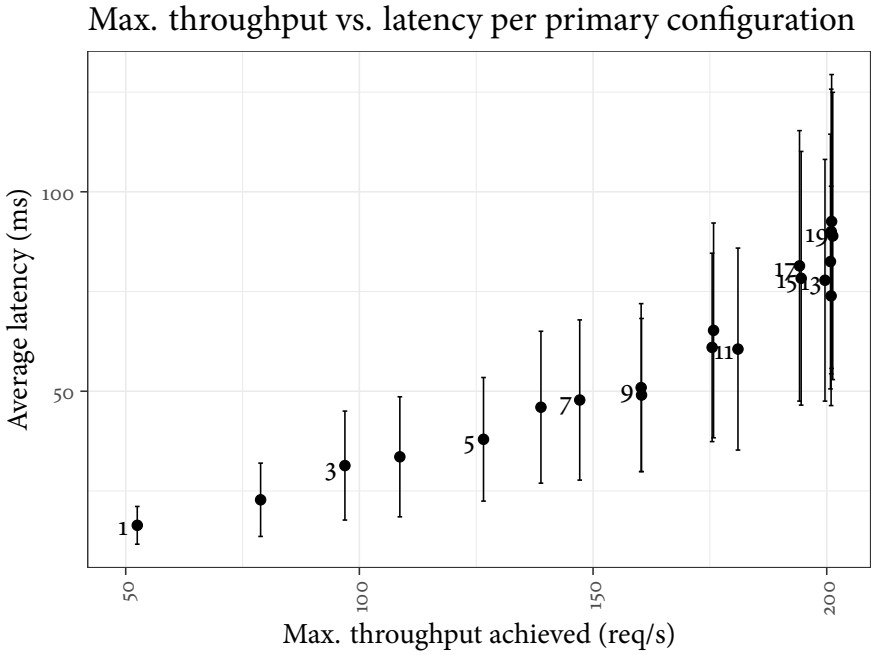

Figure 8.7. The summary of all 20 simulation runs, showing the throughput and average latency achieved during the best episode per each simulation run (i.e., the episodes with the best score as calculated by Equation (8.1)). Error bars show the latency standard deviation around the average for each episode.

i.e., the same number as there are CPU cores in the simulated system, provides throughput numbers in the upper quartile of all observed values, while keeping latencies manageable at an average of around 50ms, which is roughly similar to the median of observed latencies and approximately 2.6 to 3.3 times the length of a generated request's minimal execution time (as dictated by the simulation configuration).

Discussion

Optimal sweet spots for performance are of course highly dependent on the actual purpose of a system. Some interactive systems prefer predictable, low latencies and can sacrifice throughput, while batch-processing systems might prefer the highest achievable throughput without worrying about latencies. Regardless of a system's specific performance goals, these experiments have shown that just changing a single UDS parameter can have tangible effects on a system's behavior.

Classifying Parameter Impact

Similar to these previously shown runs with primaries, a multitude of simulations were performed in addition to these few shown here. Additionally, during this time in our research, efforts to implement UDS in a real SMR-system were underway and first results of real-world benchmark runs started to accumulate. These efforts will be expanded on in the following chapters. With this gathered data and experience, roughly gauging the possible impacts off different system and UDS parameters on general behavior became possible. Putting these results from UDS-SIM experiments and first real benchmarks together yielded first estimates of some parameter effects on the most general system metrics. These estimates are summarized in Table 8.1.

	Impact on Metric		
Parameter	Throughput	Latency	Parallelism
`n(r)`	medium	high	high
`steps(r, θ)`	little	medium	medium
Total Order	little	medium	high
# of CPU Cores	high	little	high
Application Profile	medium	high	high

Table 8.1. Rough estimates of how much the most important UDS parameters and system variables affect general metrics of system behavior, partially based on experiences won by running UDS-SIM experiments.

These results are to be taken with a grain of salt, however, as they are non-quantitative, and were merely meant to guide us in our further research effort, e.g., to decide which parameters to focus on first when utilizing UDS' dynamicity to improve system adaptability and performance. They are provided here merely to supply readers of this thesis with a similar intuition for how UDS can generally affect systems.

8.2 ALGORITHM BUG FIXES

While implementing UDS for UDS-SIM and in an actual SMR-setting, during many trials, errors, and focused bug hunting sessions, a few substantial bugs originally published in the algorithm in our 2016 paper [5] were found and subsequently fixed. As part of the proper documentation of our research, as well as a short *lessons-learned* for demonstrating the pitfalls one may encounter when designing a scheduling algorithm for deterministic multithreading, we briefly list these bugs and how they were fixed in this section.

UDS Versioning

For referencing and keeping track of different versions of UDS, we started numbering its versions as it developed over the years. Our first published version from 2016 had the version number 1.1, as it was an iteration of an earlier internal version 1.0[5]. The following subsections will list the subsequent versions of UDS alongside the fixes we made in each version, until the current version of UDS, 1.2.3, as it was presented in this thesis, is reached.

8.2.1 *Overzealously Ending Rounds*

Problem Description

One of the first major bugs we fixed early on was located in `checkForEndOfRound()`, in the list of conditions to determine whether a round has ended. In our original version, we only checked whether a thread is enqueued in a mutex-queue, is terminated or has used all of its steps in the current round, as shown in the following short snippet:

```
74 foreach(θ ∈ Θ_prim) {
75    if( θ.enqueued == nil &&
76       !θ.terminated &&
77       !θ.finished) {
78       return  # not at end of round
79    }
80 }
81 # end of round detected
```

Mitigation

However, it seems quite obvious in hindsight that additionally a check for whether a thread is currently waiting for its turn is required as well, as seen below or in the current version of the algorithm in Line 78. Consider Line 144 in the current version of `waitForTurn()`, where threads are blocked if they are primary, but can not yet consume a step because they are not yet first in line in the current round's total order. This, of course, is quite a common occurrence, and meant that without the fix as shown below, rounds frequently ended even though threads that could have made progress in the current round were still available. These two lines, modifying the `waitingForTurn`-condition for a thread, were the first necessary modification for this fix. Secondly, the respective check was added to `checkForEndOfRound()`:

5 Between 1.0 and 1.1, mainly the structure and presentation of the algorithm changed, without significant modifications to the logic of the algorithm itself.

```
74 foreach(θ ∈ Θ_prim) {
75    if( θ.enqueued == nil &&
76       !θ.terminated &&
77       !θ.finished &&
78       !θ.waitingForTurn ) {
79       return  # not at end of round
80    }
81 }
82 # end of round detected
```

These changes brought UDS to version 1.2.

8.2.2 *A Ghastly Deadlock*

Problem Description

The next bug was a little more subtle [6] and devious. To understand it, first consider the following version of `waitForTurn()` as found in UDS up until and including version 1.2:

```
120 waitForTurn() := {      # enforce total order
121    if( round = 0 ) startRound()  # bootstrap
122    θ := currentThread
123    waitFor( θ.primary ) # wait until primary
124    i := Θ_prim.getIndexOf(θ);
125    if( !totalOrder.contains(i) ) {
126       # No more steps
127       θ.finished := true
128       checkForEndOfRound()
129       waitFor( θ.finished = false )
130       waitForTurn()  # Repeat
131    } else { # wait until first in total order
132       θ.waitingForTurn := true
133       waitFor( θ = Θ_prim[totalOrder[0]] )
134       θ.waitingForTurn := false
135       totalOrder.removeFirst();
136    }
137 }
```

In this version, a thread checking whether it can perform a critical action waits until it becomes primary, then checks whether it has any steps left in the total order (Line 125), and if it does, waits until it is the first in the total order (Line 133) before finally continuing with its action. This works fine for many possible total orders and application profiles. However, assume we have a total order in a round-robin fashion, with at least two primaries `t0` and `t1`, and at least two steps per primary, e.g., `[0, 1, 0, 1, 0, 1]`. Assume further that the first primary `t0` executes a request which demands the subsequent nested locking of three mutexes `m0`, `m1`, and `m2` before unlocking them in re-

6 To give proper credit, it was initially noticed by Philipp Butz, a student assistant tasked with implementing UDS in both UDS-SIM and in a later master's thesis.

verse order and terminating. The second primary `t1` simply requires mutex `m0` before unlocking it and terminating. When `t0` has consumed its first step and performed its critical action (= locking mutex `m0`, which it was granted immediately as `m0` had no owner at the time), it woke up `t1` in this process (cf. Line 135), which had been waiting for its step to be at the head of the total order. Therefore, `t1` now proceeds with its own attempt to lock `m0`. This puts it into `m0`'s wait-queue and `t1` gets blocked while waiting for this mutex to become free (Line 31). The total order in this example now reads `[0, 1, 0, 1]`.

After this, `t0` wants to proceed with its next critical action, which is to lock `m1`. It re-enters `waitForTurn()`, and since once again it has a step at the head of the total order, it can proceed with locking `m1`, which again succeeds. Now, the total order reads `[1, 0, 1]`. `t1` is still blocked while waiting for `m0`, since this mutex is being held by `t0`. `t0` now wants to perform its third critical action, which is to lock `m2`, so it re-enters `waitForTurn()`, sees that it has more steps left in the current round, and then gets blocked at Line 133 while waiting for its next step to rise to the head of the total order. At this point, both primaries are blocked in a classic deadlock introduced by the scheduling algorithm, waiting forever for conditions that will never occur.

Mitigation

To fix this problem, `waitForTurn()` was rewritten to separate the two cases of i) a thread being first in the total order and ii) a thread waiting for it to become first. Before a thread may wait for the latter, however, it performs a `checkForEndOfRound()` after setting its own status to θ`.waitingForTurn = true`, but before actually blocking. This way, the end of a round in which no further actions can be taken is detected and prevented.

```
120 waitForTurn() := {      # enforce total order
121    if( round = 0 ) startRound()  # bootstrap
122    θ := currentThread
123    while(true) {
124       waitFor( θ.primary )
125       i := Θ_prim.getIndexOf(θ);
126       if( !totalOrder.contains(i) ) {
127          # no more steps
128          θ.finished := true
129          checkForEndOfRound()
130          waitFor( θ.finished = false )
131       } elsif( θ = Θ_prim[totalOrder[0]] ) {
132          # next in the order
133          totalOrder.removeFirst();
134          if( totalOrder.length > 0 ) {
135             # Wake up next
136             Θ_prim[totalOrder[0]].waitingForTurn := false
137          }
138          break
139       } else {
140          # waiting for being first in the order
141          θ.waitingForTurn := true
142          checkForEndOfRound()
```

```
143             waitFor( θ.waitingForTurn = false )
144         }
145     }
146 }
```

8.2.3 *Fixing a Deadlock, Creating a Livelock*

Problem Description

However, this was not yet the end of this bug. After these modifications to UDS, it was quickly noticed that it was still possible for the primaries of the next round to request the same currently blocked mutexes, which would enqueue them again, and depending on the total order of the next round also recreate the problem of the previous round, immediately ending the next round. This process can repeat forever, effectively livelocking the system, preventing it from making progress.

Mitigation Strategies

A possible fix for this bug introduced the new state variable `progress`, which recorded whether any significant progress in the current round was made since it started. If a round ends while this variable is still false, a livelock can then in theory be prevented by changing scheduler parameters, such as increasing the number of primaries for subsequent rounds or mutating the total order. The exact locations where `progress` was introduced are not explicitly listed here and can simply be found in the current version of the algorithm as presented in Section 7.1.

Deadlocks & Livelocks, Again

It is questionable, however, whether the introduction of the `progress` variable is a permanent fix for the problem of livelocks. We have not yet conducted in-depth research on the possibility that dead-/livelocks may be a conceptual problem introduced by configurable deterministic scheduling, as opposed to a one-time bug that we managed to fix. In other words, since UDS and other deterministic schedulers introduce complicated patterns of wait-Conditions unrelated to the execution of actual application logic, it is theoretically possible that among the myriad of combinations of possible application profiles and number of primaries and total orders, a special case exists, which results in dead-/livelocks in deterministically scheduled systems, regardless of the actual implementation of the scheduler. This could be worth investigating, e.g., by utilizing model-checking tools for parallel systems like the TLA^+ ecosystem [57], but was not within the scope of this thesis.

After the fixes for these two related deadlock and livelock bugs, UDS was now at version 1.2.1.

Shortly thereafter, a minor bug with the newly introduced progress handling was found, which is too small to explicitly list here. It also was quickly fixed, and the UDS version bumped to 1.2.2.

8.2.4 *Prematurely Ending Rounds, Again*

Problem Description

Another small but interesting bug (by way of remaining undetected despite the

earlier scrutiny of `checkForEndOfRound()`), affecting round endings once more, was found a short while later.

`checkForEndOfRound()` in UDS 1.2.2 looked like this:

```
71 checkForEndOfRound := {     # detect end of a round
72    foreach( θ ∈ Θ_prim ) {
73       if( θ.enqueued = nil &&
74       !θ.terminated &&
75       !θ.finished &&
76       !θ.waitingForTurn ) {
77          return          # not at end of round
78       }
79    }
80    # end of round detected
81    ... # perform round ending stuff
```

The problem occurs when this function is called while there are not yet all the primaries for the current round present in the system, which can happen in numerous ways while a round is starting and threads are being added to Θ_{prim}. In case all the primaries that have been added to the round so far are waiting on some condition, a round ending could be detected before all `n(r)` primaries for this round are present in the system.

Mitigation

The fix was consists of adding a short check at the start of `checkForEndOfRound()`:

```
71 checkForEndOfRound := {     # detect end of a round
72    if( |Θ_prim| < n(r) )
73       return     # not at end of round
74    foreach( θ ∈ Θ_prim ) {
75       if( θ.enqueued = nil &&
76       !θ.terminated &&
77       !θ.finished &&
78       !θ.waitingForTurn ) {
79          return          # not at end of round
80       }
81    }
82    # end of round detected
83    ... # perform round ending stuff
```

8.2.5 *Bootstrapping UDS*

Problem Description

The last bug fix bringing UDS to its current version 1.2.3 was located in `Thread::terminate()`. In UDS 1.2.2, it read as follows:

```
46 Thread::terminate := {      # self-termination
47    progress := true
48    θ := currentThread
49    waitFor( θ.primary )
```

```
50    θ.terminated := true
51    removeFromOrder( θ );
52    checkForEndOfRound()
53    θ.stop()
54 }
```

If a system has only ever seen threads which use no critical action, threads will run into `Thread::terminate`, but UDS has never been bootstrapped with a first round. Therefore, no thread exists which determines primaries, meaning that as long as no thread requests a critical action and thereby starts the first round, all other threads will forever be blocked during termination, on `waitFor(θ.primary)`.

Mitigation

An easy fix is to simply let the very first thread that tries to terminate itself bootstrap UDS:

```
46 Thread::terminate := {      # self-termination
47    if( round = 0 ) {
48       startRound()        # bootstrap
49    }
50    progress := true
51    θ := currentThread
52    waitFor( θ.primary )
53    θ.terminated := true
54    removeFromOrder( θ );
55    checkForEndOfRound()
56    θ.stop()
57 }
```

This concludes the journey through the lessons learned while developing UDS in the earlier phases of our research.

8.3 JUMP-STARTING ROUNDS

Optimization for Starting Rounds

A potential optimization opportunity was realized when we took a closer look at how threads consume steps for performing critical actions. In earlier paragraphs, we stated that a round could only start once all primaries for this round are present in the system. This is not entirely wrong, but a bit too strict in its formulation. Consider the fact that threads arrive in a well-defined total order, delivered by the GCS of the system, meaning that if we have filled a round partially, e.g., to half its primaries, and all threads are present in order, the first threads can actually start consuming their steps early. This holds true as long as the condition is met that all threads prior to the thread currently wanting to consume its step are present and have been added to the primaries in order.

We implemented this optimization in our version of UDS that was used for all future evaluations in this thesis, which will sometimes explain less noticeable effects of RFDs for certain request types where we would otherwise expect the full impact of RFDs.

8.4 CONCLUSION

In this chapter, we presented our contributions towards the first concrete implementations of the abstract UDS algorithm. First, we created a full-fledged event-based simulation of UDS, called UDS-SIM. This simulation framework made it possible for the first time to explore UDS in an experimental way. Thanks to a comprehensive array of configuration options, possibilities within UDS-SIM were vast and allowed us to gather first crucial data on UDS' inner workings.

The main results of these experiments were

- firstly, a significantly improved understanding of the impacts of UDS' configuration parameters on system metrics such as throughput or latency, and
- secondly, the discovery and fixing of some more or less severe logical bugs in the algorithm itself.

With this phase of research, we therefore laid valuable foundations for our subsequent research into the best ways to utilize UDS' dynamicity when implemented and embedded into an actual SMR system. The following chapters will dive into this research, showing how UDS can be used outside of simulations to attain tangible, valuable optimizations in real-world systems.

III

EFFICIENCY OPTIMIZATION

PROBLEM STATEMENT

This chapter completes the shift from reporting on foundational knowledge about UDS and deterministically scheduled fault-tolerant systems, to results obtained during our PhD while we researched how to optimize these systems.

9

9.1 RESOURCE EFFICIENCY AS A GOAL

So far, in previous chapters, the focus of our discussions lay nonchalantly on things like overall throughput or average request latencies, but of course there are a multitude of other goals a system be optimized for, as touched upon in Chapter 4.

Optimization Goal

Therefore, the first fundamental question that has to be answered when optimizing any system, is *what* exactly the optimization approach should improve. Good next questions would be how to select, design, and measure appropriate metrics pertaining to the chosen optimization goal(s), in order have any idea whether deployed optimization techniques actually improved the system.

The question of which optimization goals should be focused on highly depends on circumstances and requirements of one's particular situation, of course. During discussions with peers and inside our research group, it became apparent that for the next phase of our research, in addition to the UDS-related performance improvements already foreshadowed by the UDS-SIM results, there was also significant potential for optimizing resource-efficiency in certain SMR-based systems. Specifically, it was surmised that in certain scenarios like cloud-based SMR deployments, where vertical scaling of resources, e.g., CPU cores, is possible, scaling these resources during runtime coupled with parallelism using our deterministic scheduler could have beneficial impact on resource usage.

Picking this resource efficiency as a goal prompted a series of efforts aimed at improving the footprint of fault-tolerant systems based on the SMR paradigm, both in terms hardware resource usage, and consequently also regarding the incurred monetary costs of running such a system. Not only would UDS play a role in this endeavor, but other techniques come into play as well, which will briefly be introduced in the following sections. Similarly to the prior presentation of UDS itself, most of the results and approaches presented here have also been peer-reviewed and published in 2018 [3], further validating them.

9.1.1 *The Problem with Scaling SMR Systems*

Problem Statement

First, let us look at the actual problem we are trying to solve when optimizing SMR systems for resource efficiency. The main disadvantage SMR faces when compared to other fault tolerance approaches is the amount of hardware it requires in order to function. Recall that a typical SMR system able to tolerate

f faults requires at least $N = 2f + 1$ replicas for CFT, and $N = 3f + 1$ replicas for BFT, which can be reduced to $2f + 1$ after recent work in this field (cf., Section 10.1.2). But even assuming advanced techniques, a minimum of $2f + 1$ replicas have to be provided, which adds significantly to overall system cost. Of course this high initial cost buys the ability to tolerate Byzantine faults, which few other techniques can claim for themselves. Nevertheless, it would greatly help SMR's cause and adoption rate in the wild if further resource efficiency gains could be realized.

Scaling BFT SMR Systems

Consider therefore what happens when the replicated service is faced with highly variable load (e.g., a fluctuating number of client requests). In non-replicated systems, scaling in/out is a common technique, especially in cloud environments. Cloud providers usually provide special automated scaling mechanisms[1], which scales out instances during high stress phases, and will scale in if system load drops again. Unfortunately, such horizontal scaling techniques do not work for SMR systems, as every replica has to process each request. Hence, additional replicas will not increase but usually rather decrease throughput as the protocols for ordering incoming requests will create significantly more overhead for each additional replica. For example, most variants of the popular PBFT protocol [26] require $O(N^2)$ messages with N replicas.

Consequently, it is common practice to dimension CPU power, memory, and storage for replicated systems based on the expected worst-case peak load. However, with highly variable load, this approach frequently results in under-utilization of these provisioned resources, resulting in unnecessary costs and energy consumption. On the flip side, under-provisioning these machines will result in poor or insufficient service performance in peak load situations.

9.1.2 *Vertical Scaling to the Rescue*

FITCH

Another approach, in this area of research first explored—to the best of our knowledge—by Cogo et al. [29], is to instead vertically scale replicas, i.e., add or remove computing resources like CPU cores and memory, depending on the current system load. In the common Pay-as-you-go cloud model, where only utilized resources have to be paid, this approach can save significant costs. Compared to previous research, our solution aims at further refining this approach via the addition of deterministic multithreading, thanks to our UDS algorithm.

The main goal of this phase of our research can therefore be summarized as follows: Maximize throughput of an SMR system in high-load situations while also minimizing total costs, particularly during low-load periods, by way of vertically scaling replicas. Since we conducted this research under the mantle of a DFG project called OptSCORE, we christened the architecture developed for achieving this goal after this project.

1 see https://aws.amazon.com/ec2/autoscaling/ as an example

9.2 OPTSCORE APPROACH

Proposed Architecture

In the OptSCORE architecture, we deployed our first real implementation of UDS version 1.2.3. The significant effects UDS' dynamicity can have on system performance have already been teased, but a detailed analysis of scheduler reconfiguration to further enhance performance was not yet a part of this particular research effort, and will be presented in the final part of our thesis. Therefore, for all future discussions on resource efficiency in this part, we statically set UDS to use 4 primaries and 1 step per scheduling round, which provided a good middle ground between providing high performance during high system load, and minimal effects from RFDs during low load phases.

9.2.1 *Architecture*

Fig. 9.1 illustrates the components of our OptSCORE architecture and their interactions, located within each replica.

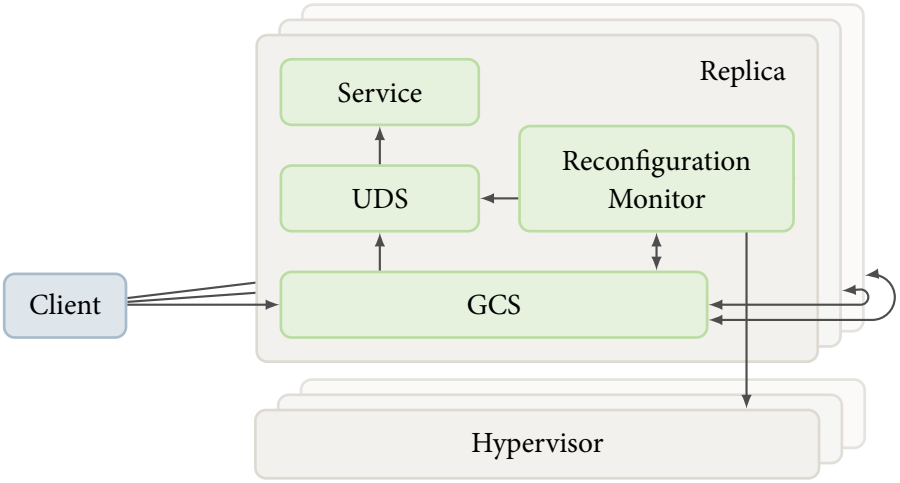

Figure 9.1. The envisioned OptSCORE architecture for vertically scaling SMR systems during runtime.

First, client requests arrive and will be totally ordered by the Group Communication System (GCS). These ordered requests are then handled by the deterministic scheduler (UDS) while being executed by the service implementation. Finally, responses are sent back to the clients.

Reconfiguration Process

During this process, the Reconfiguration Monitor (RM) observes several parameters like the CPU utilization of the host system and the utilization of queues inside the GCS. As soon as those parameters reach specific thresholds, the RM can either request additional computing resources from the hypervisor or release them. Note that as long as system resources do not affect determinism, which they generally shouldn't for normal hardware and applications, there is no need to specially handle reconfiguration decisions regarding these resources. This seemingly petty detail will come back to haunt us in later chapters, where we would perhaps like the RM to also mandate a reconfiguration of UDS, in order to always guarantee perfect utilization of the currently provi-

sioned resources. This requires determinism, both in detecting and measuring changes, as well as broadcasting or a similar means of distribution of reconfiguration decisions to all replicas. We will revisit this problem in Part IV, as this was not yet a part of the system setup for this phase of our research. Also note that scaling happens automatically due to decisions by the RM and does not have to be initiated by a user.

With this architecture, system resources can in theory scale dynamically with load and lay the groundwork for cost-efficient SMR installations. In the next chapter, we briefly discuss related work in this area and how we implemented our evaluation platform while working towards this goal, before we finally present and discuss our evaluation results on resource efficiency optimization.

VERTICALLY SCALING SMR SYSTEMS

After the problem statement and before we dive into our implementation and evaluation details, let us take a brief look at how exactly resources like CPU cores can be scaled during runtime, and at the related prior work regarding SMR efficiency optimizations.

10

10.1 BACKGROUND AND RELATED WORK

10.1.1 *System Support for Vertical Scaling*

Background on Vertical Scaling

Vertical scaling is the adaptation of resources either during runtime or offline, and can be achieved in multiple ways. For our purposes, we focus on vertical scaling in virtualized environments, i.e., scaling of resources using virtual machines (VMs). Theoretically, hot-plugging hardware would be possible nowadays, but is of course neither widely employed nor easily automatable. Scaling virtual resources can either be performed during runtime by the hypervisor managing the underlying hardware, or by preparing a new VM with more resources and migrating state from the current VM. In some controlled environments (i.e., private cloud setups), the first method could allow for live vertical scaling. However, current public cloud providers usually do not (yet) make the necessary functionality accessible to their customers, and thus support only the second method. The following paragraphs will primarily discuss details on the first approach, as it is the one we envisioned as part of our solution.

Vertical Scaling Support

Scaling resources during runtime in virtualized systems is currently not fully supported by all hypervisors and guest operating systems. The Linux kernel for instance has been able to plug and unplug CPUs at runtime, as well as to attach or detach RAM modules, for many years now. However, Linux-based hypervisors, like KVM or XEN, are to some extent not able to create new virtual processors and RAM modules after a virtual machine has been started. On the other hand, hypervisors can distinguish between available and assigned resources. By over provisioning the available resources from the beginning, the actual resources can be dynamically assigned to the respective guests. Adding and removing CPUs can be easily done by switching on and off virtual CPUs in the device file systems. While adding virtual RAM also causes no major difficulties, removing RAM sometimes requires collaborative techniques for giving back unused RAM, e.g., memory ballooning. Table 10.1 shows the minimum version of hypervisors supporting vertical scaling of CPUs and RAM as of 2022. The following discussions are the result of work within our research group, with J. Köstler in particular providing much of the original data gathering in 2018 [3], which was augmented by our own research while vetting this data for inclusion in this thesis.

	CPU		RAM	
Hypervisor	**Add**	**Remove**	**Add**	**Remove**
KVM/QEMU	QEMU 1.5	QEMU 2.7	QEMU 2.1	QEMU 2.4
XEN	XEN 2.0		XEN 3.0	
ESX/ESXi	ESX/ESXi 4.0		Not supported	
Hyper-V	Not supported		Hyper-V 2016	

Table 10.1. Hypervisor Support for Vertical Scaling.

In QEMU, the maximum assignable number of cores and amount of RAM must be specified before the machine boots. The same applies to the XEN, which supports those operations since its early versions. VMware's hypervisors integrated hot-plugging of virtual CPUs and hot-adding RAM resources with ESX/ESXi 4.0. However, as this feature is disabled by default, it needs to be activated explicitly for each machine. Hot-removal of RAM is still not supported, even in newer versions of the hypervisor. Microsoft's Hyper-V on the other hand supports both adding and removing of RAM resources in recent releases, but is still not able to hot-plug virtual CPUs as of 2022, even if there are a number of runtime options that allow to increase and decrease the utilization of the logical host CPUs.

Table 10.2 shows the support for dynamically adding and removing processing resources in guest operating systems. For x86-based CPUs, Linux systems integrated these features into the Linux kernel during the development of the Kernel version 2.6, whereas Microsoft added support for hot-plugging RAM into their server operating system family with Windows Server 2003 and for hot-plugging CPUs with Windows Server 2008.

	Scaling Support	
Guest OS	**CPU**	**RAM**
Linux	Kernel Version 2.6.13	Kernel Version 2.6.15
Windows	Windows Server 2008 Data Center Windows Server 2012 Standard Windows Server 2012 Data Center and above	Windows Server 2003 and above

Table 10.2. Guest Operating System Support for Vertical Scaling

If the requirements of the hypervisor and the guest operating system are met, vertical scaling can be implemented in private cloud environments.

Vertical Scaling Availability

Regarding publically available offerings, some cloud providers are already starting to offer vertical scaling services[1], even though major cloud providers

1 see for example https://docs.ionos.com/cloud/compute-engine/virtual-servers/virtual-servers, which allows the addition of cores and RAM during runtime, but no removal

like Amazon, Google or Microsoft do not. Thus, vertical scaling in public clouds results in convoluted replacement strategies of VMs during runtime, or entails downtimes, as old virtual machines are shut down and new resized ones reusing their virtual disks are started.

In summary, vertically scaling CPU cores and RAM during runtime is possible in private cloud environments, but not yet fully supported in popular public cloud offerings. However, we expect these features to become increasingly available in coming years.

10.1.2 *Related Work*

We only briefly list directly related research in this section, glossing over a larger body of research that works on, e.g., enabling multithreading in SMR systems, as this topic can be argued to only peripherally relate to the main goal of researching the effects of vertically scaling SMR systems. In a later chapter, we will make up for this omission by then introducing this body of work, at a position where its connection to our efforts is clearer (cf., Section 14.1).

Vertical scaling for SMR

Optimizing SMR systems with vertical scaling seems to have only been studied by very few other researchers so far. FITCH [29] also attempts to optimize an SMR system with vertical scaling (alongside other approaches). It achieves scaling by replacing replicas with new ones having altered configurations or using different hardware resources. However, the necessary reconfiguration and synchronization processes result in some non-negligible overhead. We are currently not aware of any other SMR system that allows vertical scaling of resources at runtime.

Other Approaches for Resource Efficiency

Orthogonal Approaches

There are numerous approaches trying to minimize the replication costs by reducing the number of replicas in the system. Through the separation of the agreement part of the protocol which orders the requests and the actual request execution, the number of replicas needed for the potentially more costly request execution can be lowered to $2f + 1$ [83]. With newer approaches, the number of replicas, including those needed for ordering requests, can generally be reduced to $2f + 1$. This is either achieved by trusted subsystems [22, 30, 52, 78] or by setting f replicas during normal case operation into a passive mode where they neither order nor execute any requests [35, 37]. This can save costs of up to a quarter of the total cost of operations.

Own Work

Yet another approach is to specifically optimize certain types of application for Byzantine SMR settings. We have collaborated on such an approach, called SmartStream, in [6], and other examples include [69], or [48].

It is important to note that our optimization approach is orthogonal to all of these mentioned approaches, since it targets each individual replica instead

of the entire group. Hence, these approaches could further benefit from our solution and vice versa. A combination of our approach with those from prior work is left to future research.

10.2 IMPLEMENTATION

Prototype Goals and Approach

To evaluate the approach sketched in the OptSCORE architecture, we implemented a prototype system to run evaluation tests in a consistent and reproducible way. Since this was meant as a first step for evaluating whether the approach works, we did not yet include automatic scaling in the test framework and instead explicitly scaled CPU cores during different test runs, according to pre-determined configuration files created for specifically testing various use cases (cf. the evaluation in Section 10.3).

For measurements, we chose to utilize micro-benchmarks, since i) it is a well-known and frequently used approach to validate early prototypes or special systems, and ii) to avoid occlusion of our solution's effects by possible effects introduced by (mis-)configuring a real-life distributed application. In the same vein, since our goal was to optimize the normal case, we do not include tests with replica failures and therefore also disabled checkpointing in the GCS.

As described in Section 10.1.2, vertical scaling of resources at runtime is difficult to achieve with current virtualization solutions. Therefore, we implemented a system which demonstrates our claims by scaling CPU cores using the Linux kernel, by editing the files found in `/sys/devices/system/cpu/`.

10.2.1 *Replication Framework*

BFT-SMaRt

Our replication framework and all future work is based on the popular Java-based BFT-SMaRt library [25], which conceptually is divided into two parts: The GCS part, which orders incoming requests and utilizes multiple cores reasonably well already, and the application logic part, which hands control over to the user of the library sequentially in a single delivery thread. The user is given batches of ordered requests, which are then simply executed one after another.

Based on previous extensions of BFT-SMaRt kindly provided by Eduardo E. P. Alchieri, we modified the second part of the library so that each ordered request is processed by an individual thread, to adhere to our system model as presented in Section 3.4. This means that, e.g., for a batch of 10 ordered requests, 10 independent threads would be started, where each thread is responsible for the processing of one request and for sending back a reply to the client.

10.2.2 *Deterministic Multithreading*

UDS Version and Integration

Since parallel request processing would introduce indeterminism, we implemented UDS version 1.2.3 in Java, then integrated it with BFT-SMaRt's request execution, such that UDS controls the individual request processing threads taken from a standard JDK cached thread pool. The number of *existing* threads can therefore fluctuate during runtime and is managed by the thread pool, while the number of *active* threads at any given time is controlled by UDS, which might wake threads up or put them to sleep in order to wait for their turn.

10.2.3 *Linux Core Scaling*

Scaling CPU Cores

By writing either `1` or `0` to `/sys/devices/system/cpu/cpu`N`/online` in Linux, CPU N can be taken on- or offline during runtime. Taking a core offline effectively makes it inaccessible to the OS scheduler, and processes, interrupts, and timers are migrated away from this core to remaining ones. Enabling a core makes it visible to the OS scheduler, so that threads can be assigned to the core again. This rather closely emulates what vertical core scaling of VMs via a hypervisor would look like, and is used for the following evaluations.

Since core (de-)activation is done by merely writing to files, we implemented this functionality in Java so that it can be controlled during runtime of an application from inside the JVM, gathering all parts necessary for vertical scaling tests in a single JVM environment.

10.3 EVALUATION AND RESULTS

Evaluation Procedure

To evaluate our main research questions we performed measurements on a BFT-SMaRt setup with $N = 4, f = 1$ in BFT mode. During the measurements, up to 8 different client machines connected to the cluster, each spawning several client instances, which were ramped up to server saturation over the course of test cases (cf. Subsection 10.3.1). The replica machines each sported 2 dual-socketed 16-core AMD EPYC 7281 CPUs, 128 GB DDR4 RAM, and were all connected via a single Gigabit switch.

The client machines consisted of a mix of physical quad-core Xeon E3-1220 machines and quad-core VMs running on Xeon E3-1230. Clients were connected to the replicas via the same Gigabit switch. Due to our chosen evaluation workloads and methodology, the Gigabit link never represented a bottleneck during the measurements. All machines were running Ubuntu 16.04 LTS with Linux Kernel v4.4 and Java version 1.8.0_162. We used our modified BFT-SMaRt version, which was forked from commit `6892ab38` of version `v1.1-beta`[2]. These modifications did not, however, include any changes to the entire GCS part of the library. We added modifications to a custom `ServiceReplica` implementation, receiving threads after the `Delivery-`

2 As found on GitHub at https://bft-smart.github.io/library/

`Thread` delivers batches of requests to the business logic. BFT-SMaRt was configured to use a batch size of 400 and in- and out-queues of size 500k each.

10.3.1 *Methodology*

Using our modified BFT-SMaRt library, we implemented an application with a distinct application profile, capable of handling 5 different types of requests (also called *workload* in the following discussion):

`NOOP`: Does no computation and immediately returns once delivered by the GCS. We used it to test the overhead of deterministic scheduling.

`C250`: Returns after a computation of $250\mu s$.

`C1000`: Returns after $1ms$ of computation.

L_{32}`250U`: Performs a computation for $250\mu s$, but guards this with one out of 32 randomly selected mutexes (locking before computation and unlocking afterwards). The maximum number of concurrently executing requests is thus 32. This imitates a workload requiring exclusive access to a fraction of shared data.

$L_1U_1C_{250}L_1U_1$: Also runs for $250\mu s$, but quickly locks and unlocks a single mutex before and after processing. This imitates accessing shared state before and after request processing, as it is often found in real applications.

Application Profiles

The first three application profiles are used to show scheduling overhead and general behavior of the system under normal uncontended load. The last two request types represent real applications, which could access shared state by guarding it through the use of locking. The L_{32}`250U` workload with its 32 locks (twice as many as the maximum number of cores tested and shown in the results) is used to show the effects of an application that is highly parallelizable, while the $L_1U_1C_{250}L_1U_1$ workload shows how much overhead is introduced by fast locking and unlocking of a single, highly contended lock.

Configuring Benchmarks

A configuration of a benchmark run on the evaluation system mainly consists of the parameters for i) the number of active CPU cores, ii) sequential vs. parallel execution and iii) one of the five workloads. We did not consider workloads with mixed request types (as expected in real-world applications) in order to cleanly separate the effects that are meant to be shown by each particular workload.

For each benchmarking configuration, we performed separate evaluation runs, which each start with a low number of clients and continuously ramp up to high load. Load is ramped up by continually adding new clients, where each client, once connected, constantly sends synchronous requests until the entire test has finished. Thus, at the end of a test run, hundreds of clients will be putting the replicas under stress. Increasing the number of clients took place in cycles In each cycle we spawned new client instances, then let the entire cluster stabilize for 15s, and finally started a measurement phase of 30s,

in which response times were logged at the clients for every request. From this data, we calculated overall throughput (req/s) and average response times, before continuing with the next cycle and adding new clients.

Plotting Results

Each run is then represented as one line in the following graphs, with each line consisting of multiple data points — one for each client count per run. The graphs plot throughput (x-axis) against response time (y-axis). As soon as the cluster starts to approach its throughput limit, throughput will no longer increase while response time spikes rapidly, due to queued requests piling up. The maximum sustainable peak throughput for a given configuration is where throughput is highest without latency yet spiking.

The following sections will present and discuss the results gathered from performing a total of over 600 of these benchmark runs over a course of several weeks.

10.3.2 *Single-threaded Execution*

Baseline Results

In our first experiment, we established a baseline by scaling the number of cores in our prototype system without multithreaded request execution. We then compare all five workloads (see Figure 10.1), to see how BFT-SMaRt and the evaluation application perform without any modifications. In the graphs, the legend shows the configuration of each evaluation run.

With the `NOOP` workload, adding cores shows increasing performance up to a certain point, which is mainly achieved due to the internal concurrency in the group communication system part of the library we did not modify. In our case, BFT-SMaRt saturates at 4 active CPU cores. Adding more cores after that does not lead to better performance, but instead causes throughput to slightly fall off again. This is due in part to some quirks of the architecture of AMD EPYC CPUs, especially the first-gen ones we used in our evaluation systems. EPYC CPUs consist of a Multi-Chip Module made up of so-called CCX, which in turn consist of 4 cores with their own respective L3 caches. Communication between CCX incurs a latency penalty, so thread migration between cores on different CCX noticeably impacts performance. For more details, see [91]. This effect could be counteracted to some extent by manually pinning threads to cores. We did not attempt this, because it would mean that our results only generalize to systems where DevOps Engineers responsible for running such a resource-optimized BFT installation perform this additional, highly system dependent and tedious manual step, whereas we wanted our results to generalize well to all kinds of systems.

All other workloads use requests that perform some work on the replicas. In the three cases with a 250μs workload (`C250`, $\texttt{L}_{32}\texttt{250U}$, $L_1U_1C_{250}L_1U_1$), the sequential execution of workloads leads to a performance increase if we scale to 4 cores, but does not see any improvement for more cores. This is because request execution was still forced to be sequential, so only one core at a time was used, which gets saturated at about 4000 req/s (i.e., the maximum number per second possible for this type of request thanks to the 250μs calculation in each request), while all other internal tasks, e.g., for request ordering inside

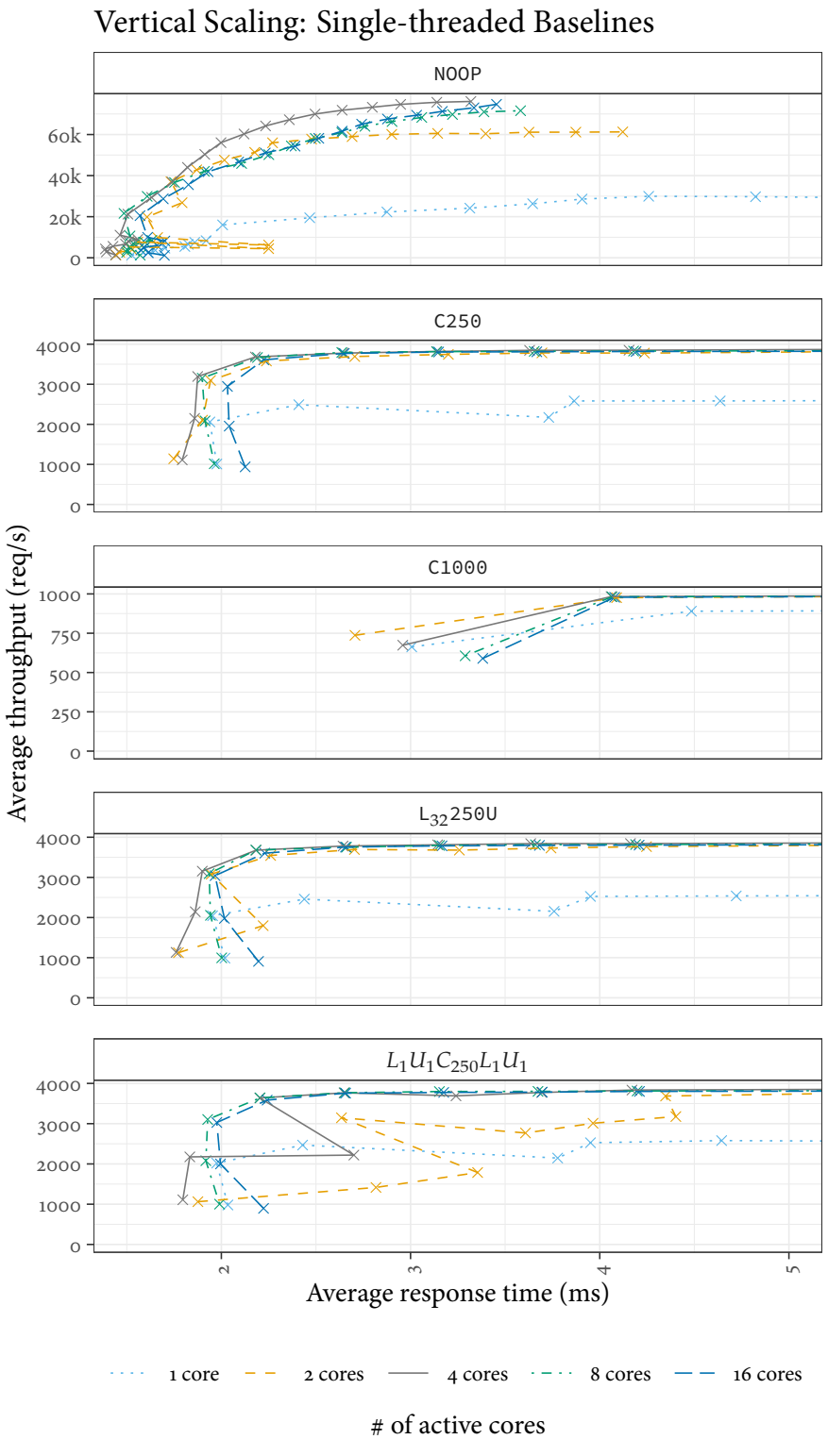

Figure 10.1. Performance of different workloads with vertical CPU scaling and single-threaded request execution

BFT-SMaRt, can benefit from the additional cores. For the `C1000` workload, the behavior looks similar, but even more pronounced.

Furthermore, we can observe that sometimes an increase in the number of clients does not increase throughput, but causes latency to spike a bit. Since this was repeatable, we assumed this to be a systematic occurrence within our system setup, especially with few active cores, but did not further investigate the effect, since it does not significantly affect our evaluation results and claims.

Lastly, we also performed measurements for 32 cores, but did not add the results of these runs to any of our graphs, as they happen to be always worse than 16 cores. This can also be explained by the architecture of the EPYC CPUs inside the replicas: Not only does communication between a CCX incur a performance penalty, but thread migration between two sockets exacerbates this effect. Since we did not use thread-pinning, the results of tests with more than 16 cores would add nothing of value to our discussion and only hinder the readability of plots.

Limited Effects in Single-Threaded Systems

The observed data so far serves to confirm results of prior work, i.e. that depending on the workload, vertical scaling of CPU cores has limited effect on traditional SMR systems which execute requests sequentially. Additionally, they established a baseline for comparisons in the following evaluations. Figure 10.2 and Figure 10.3 include such a reference as a line displaying the single-threaded performance with four cores, as the best overall best configuration for sequential execution (according to Figure 10.1).

10.3.3 *Deterministic Multithreaded Execution*

Benefits of Multithreading

Improved utilization of multiple cores can be achieved with deterministic multithreading, as discussed before. However, the introduction of a deterministic scheduler to a system inevitably leads to an unknown performance overhead. Therefore, we investigated the impact of this overhead for our UDS implementation in the next series of measurements. We used the `NOOP` workload and the compute-only workloads `C250` and `C1000` to compare the optimistically achievable upper bound (if no lock contention were present) to single-threaded execution.

The results shown in Figure 10.2 demonstrate that for `NOOP` requests (which do not benefit from parallel execution) the system suffers a performance loss of about 60%, depending on the number of cores. This loss is due to the inherent overhead of added thread scheduling and implementation details of the UDS scheduler; we have improved our UDS implementation since these benchmarks were run, so that today, these measurements would be shifted by a few percentage points. Nonetheless, it should be obvious that for systems in which requests cause only negligible system load, parallel execution with a deterministic scheduler is not useful.

Scheduling Overhead vs. Application Load

This picture changes drastically for other workloads. For both the `C250` and `C1000` workloads, we achieve substantial speedups of over four times the baseline measurements, with enough cores active. Also, when comparing the multithreaded run with 4 cores directly to the 4-core single-threaded baseline,

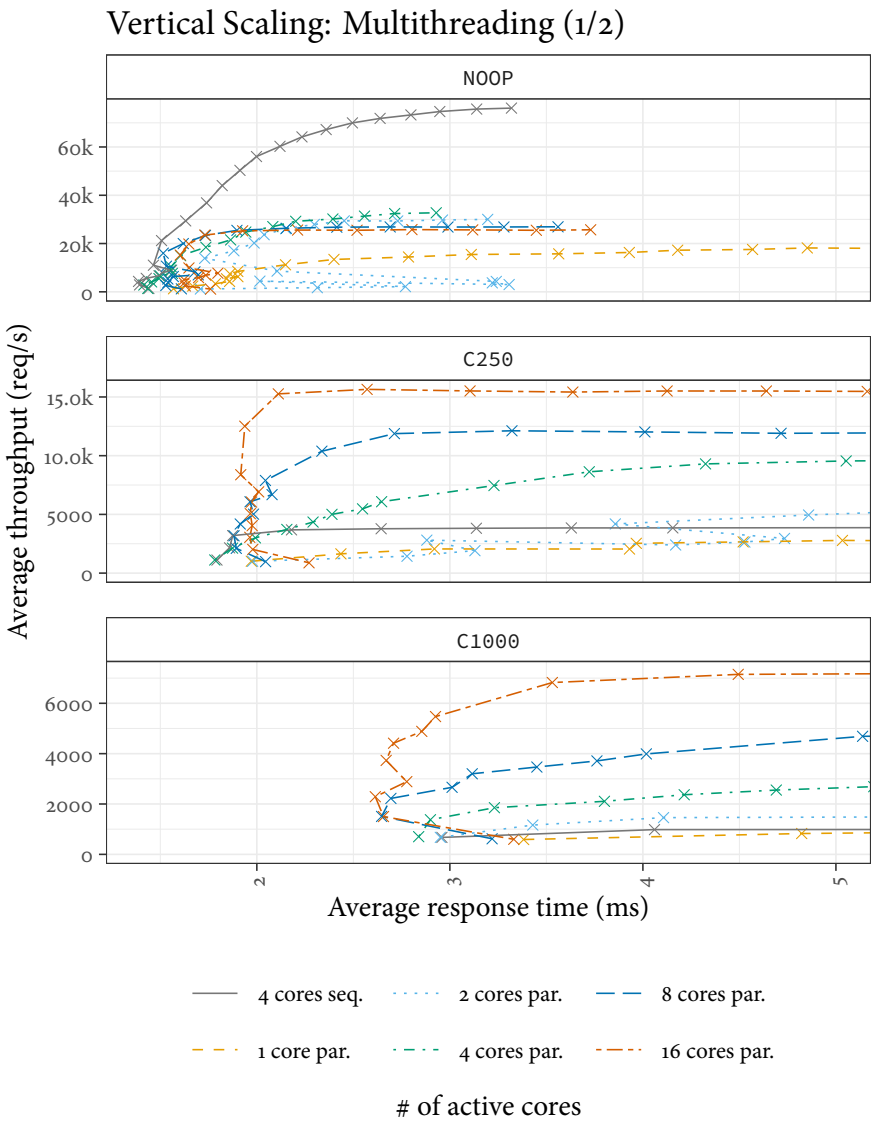

Figure 10.2. UDS overhead. The gray line (4 cores seq.) is a reference to compare to single-threaded execution.

speedups of over two times can be observed. These results therefore clearly validate the claim that the system scales well in terms of throughput if we increase the number of CPU cores, as long as requests perform some work on the replicas.

Therefore, provided an application demands sufficient execution resources in terms of CPU utilization, parallelization using deterministic scheduling can net great performance gains, even with the added scheduling overhead factored in. In other words, scheduling overhead can quickly become negligible with real workloads.

10.3.4 *Towards Realistic Workloads*

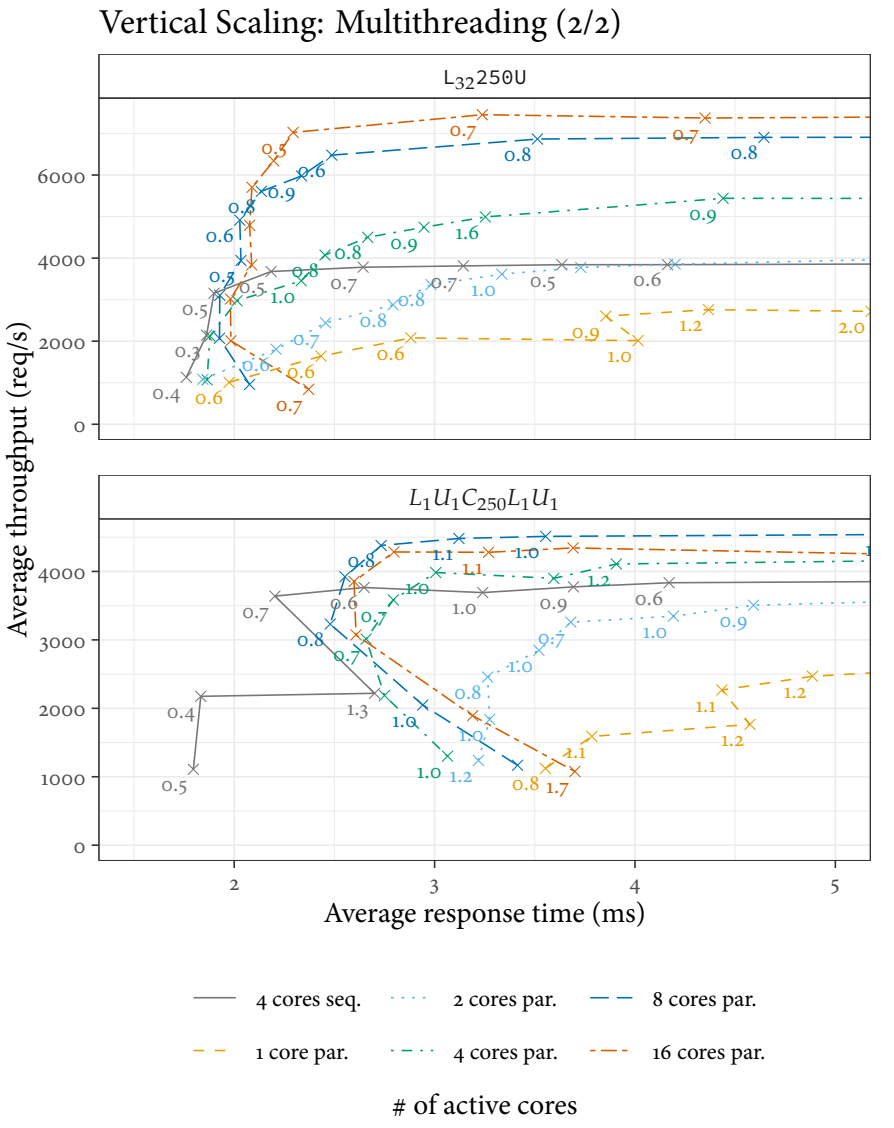

Figure 10.3. Automatic scaling for self-optimization, using realistic workloads accessing shared state. Small numbers represent SD of average request response time

Final Experiments

These results so far indicate that using vertical scaling and a multithreaded deterministic scheduler can be a valid approach, but the considered workloads produced no coordination overhead in the scheduler due to conflicting requests. In the final experiment we therefore considered more realistic use cases with workloads that use locking to protect state modifications.

The measurements in Fig 10.3 show that the benefits of having multiple cores are less than in the lock-free experiments in Figure 10.2, but that throughput still scales well with the number of cores. In this figure, since it is slightly larger and less crowded than the previous plots, we also included the standard deviations of the averaged request response times per load level as numbers next to each measurement, in order to give an even clearer picture of the system's stable behavior under different loads.

For the $\texttt{L}_{32}\texttt{250U}$ workload, the 2-core multithreaded configuration already achieves performance on par with the 4-core single-threaded measurements. Increasing the number of cores yields further significant improvements, with up to about 7440 req/s for 16 cores and multithreading compared to 3875 req/s with sequential execution.

The $L_1U_1C_{250}L_1U_1$ workload with the single global lock is an attempt to show an unfavorable case for UDS, because a total order of all lock operations must be guaranteed and the single available lock results in sequentialization of requests. Nevertheless, our system still achieves a peak throughput of around 4700 req/s with 8 cores (4103 req/s with 4 cores) compared to about 3840 req/s for the single-threaded case on 4 cores.

10.3.5 *Cost-Saving with Dynamic Vertical Scaling*

Based on these measurements and to underline the savings potential of our scaling approach we calculated the following cost estimates based on a typical daily usage scenario of an SMR service deployed to a future cloud provider with vertical scaling capabilities.

Cost-Savings Calculations

For this, we take our throughput measurements for workload $\texttt{L}_{32}\texttt{250U}$ (see Figure 10.3) as a basis and combine it with a typical daily load pattern shown in Figure 10.4 as the light gray area, as it could be experienced by a user-facing service. We calculated the maximum throughput for each full hour and determined the Amazon EC2 compute instance required to cope with this throughput if we tolerate a maximum average response time of 3ms (dark gray area in Figure 10.4). The available EC2 instances, their capabilities and costs, as well as the highest sustainable throughput per configuration (based on the $\texttt{L}_{32}\texttt{250U}$ measurements with an average response time of max 3ms) are shown in Table 10.3, with current pricing information taken from [84].

Assuming a very conservative reconfiguration interval of 60 minutes the daily costs of a dynamically scaled solution are $10.08 per replica, whereas the total costs of one replica without dynamic scaling (light orange area in Figure 10.4) would come in at $23.04. It is assumed without loss of generality that our solution introduces negligible reconfiguration overhead, which is therefore not further considered here.

Discussion

Even with this very coarse-grained reconfiguration interval, we achieve an impressive theoretical savings potential of over 56%, just in this example alone. This number will clearly vary with different workloads and daily load patterns, but could be further optimized significantly, e.g., by reducing the reconfiguration interval to minutes or even seconds.

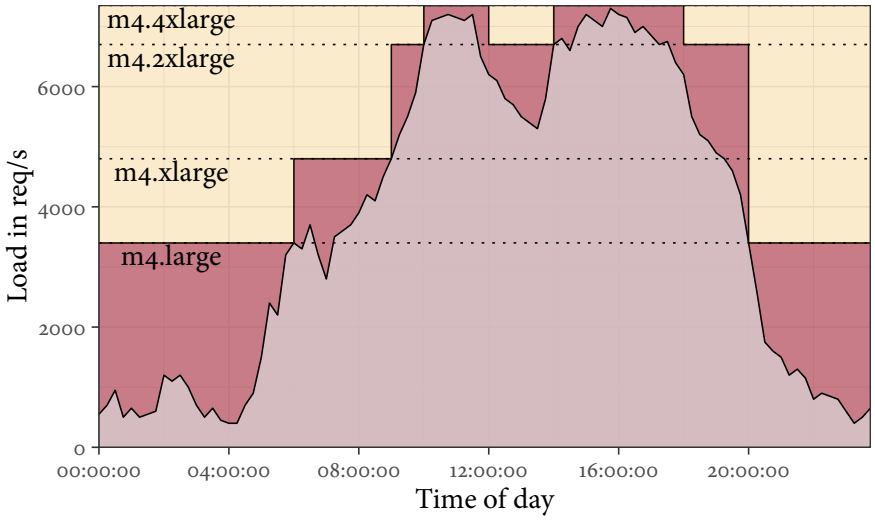

Figure 10.4. Workload Pattern (light gray) and the required Amazon EC2 instances to satisfy hourly peak load, scaled (red) vs. unscaled (light orange)

Instance Type	# of vCPUs	RAM	Hourly rate	Maximum theoretical throughput
m4.large	2	8	$0.12 per Hour	3400 req/s
m4.xlarge	4	16	$0.24 per Hour	4800 req/s
m4.2xlarge	8	32	$0.48 per Hour	6700 req/s
m4.4xlarge	16	64	$0.96 per Hour	7350 req/s

Table 10.3. Amazon EC2 Pricing (Frankfurt region), as of 2022.

SUMMARY

11

In this chapter, we introduced our contributions towards improving the hardware resource efficiency of state machine-replicated systems by way of vertical scaling, which can allow operators of such systems to substantially reduce their costs when running SMR services.

Contributions

To this end, our proposed OptSCORE architecture fills a gap in existing SMR systems by addressing the problem of expensive hardware resource costs in the face of highly variable system load. The main novel contributions of this part of our research are as follows:

- We provided a systematic analysis of approaches for vertical scaling and their applicability in public and private cloud infrastructures.
- We presented an architecture for generically supporting dynamic vertical scaling when combined with deterministically multithreaded service execution in SMR systems.
- We experimentally evaluated the OptSCORE techniques with micro-benchmarks modeled after typical patterns of application behavior.
- We presented an illustrative cost analysis modeled after common real world daily workloads to demonstrate the potential dramatic cost-savings our approach allows for, without compromising performance.

Future Work

Future work in this area would have to integrate the current OptSCORE prototype with an automatic scaling solution capable of making sensible scaling decisions during runtime. Therefore, an approach aimed at automatic self-optimization in this manner, albeit in a slightly different system setting, was our next main research goal and will be presented in the following part of this thesis.

IV

PERFORMANCE OPTIMIZATION

12 Motivation & Goals

With two of the main contributions of this thesis—our reconfigurable scheduler as well as resource efficiency optimization for SMR systems—out of the way, it is time to look at the remaining major contributions in our efforts towards improving the runtime-performance of SMR systems.

As a short reminder, the principal idea is to utilize UDS' capability to dynamically change configurations during runtime, so as to benefit from its preliminarily demonstrated effects on metrics like throughput and request latencies. Somehow intelligently reconfiguring UDS could therefore improve system performance in reaction to changes in system behavior. This part of the thesis will validate and further research these ideas. Changes in system behavior can be something as common as a sudden variation in system load due to fewer or less active clients (or the opposite), or they can be induced by, e.g., deployment of a new application, with a different application profile, to the SMR cluster. A deliberate reconfiguration of parameters in other parts of the SMR framework or simply a failure in some subcomponent which affects performance otherwise can be additional reasons for runtime-variability in an SMR deployment.

Detecting Changes

This idea of utilizing our reconfigurable scheduler seems rather simple on the face of it, yet comes with a few unexpected issues, some of which had the ungainly habit of quickly devolving into devilishly tricky problems to solve during the course of our PhD. In the description and evaluation of our previous optimization for resource efficiency, we have skirted one of the main issues of optimizing SMR systems: The question of how to actually detect changes in our system, how to do so *deterministically*, and how to distribute reconfiguration decisions of some optimizer component (e.g., the Reconfiguration Monitor of our OptSCORE Architecture from Section 9.2.1) to all replicas. An additional level of difficulty comes from the fact that we would like to retain Byzantine fault tolerance throughout this endeavor, which means that not only does any measurement of system characteristics have to happen deterministically, it also has to be resilient against Byzantine disturbances, be they of malicious or non-malicious nature. Lastly, the actual reconfiguration decisions themselves have to be derived, preferably in an intelligent manner, so that they actually improve the system. Depending on the optimization goals we choose and the number of parameters we allow the reconfiguration component to adjust, this is no easy task, either.

We had to overcome all of these challenges in pursuance of our primary goal of optimizing BFT state machine-replicated systems, and will present our solutions to these problems, as well as evaluations of their respective implementations, in the following chapters. For a better overview, the next sections will briefly introduce each problem and summarize background knowledge where needed, in order to aid with gaining an initial understanding of our mo-

tivations and approaches. More detailed information will then be provided in each of the following chapters.

12.1 DETERMINISTIC METRICS

Approach

First, in order to have a basis for our reconfiguration decisions, we require a deterministically measurable metric. This task alone presents some challenges, namely:

- In a replicated state machine, how can we measure performance metrics like throughput or request latency, which by their very nature depend on exact measurements of wall-clock time (requests per *second*, end-to-end response time latency of X *ms*)? As all replicas operate autonomously, can have a different hardware and software bases, desynchronized clocks, and different network latency compared to other replicas, each replica perceives time and its related characteristics rather differently.
- In a Byzantine system setting, how can we be certain that malicious or erroneous participants in the cluster can not influence the measurements on which we want to base our reconfiguration decisions? This precludes solutions based on a centralized measuring component, e.g., in a master replica, since this could inject Byzantine measurements and unhinge the basis of our optimization decisions.

Also, some measurable metrics, e.g., processing time per request, may vary greatly between replicas, whereas others are supposed to be at least similar, e.g. request arrival rate. To successfully reconfigure our system, we need to choose a suitable metric once the issues mentioned above have been solved.

To this end, in Chapter 13, a novel approach for deterministically measuring system metrics in BFT SMR systems will be introduced.

12.2 DETERMINISTIC RECONFIGURATION

Ensuring Determinism During Reconfigurations

After solving the problem of deterministic measurements in a distributed, BFT scenario, we have to make sure our system is actually reconfigurable during runtime, i.e., that reconfiguration of system parameters does not introduce indeterminism or otherwise break the system.

For this requirement, we can rely on our favorite novel scheduling solution, UDS, which allows for deterministic adaptation of its parameters in between scheduling rounds. Similar to the evaluation prototypes used in Chapter 10, we simply have to make sure that after receiving the measurements of our chosen metric(s), all other components responsible for choosing and applying new configurations are deterministic as well. We achieve this mostly by intelligently putting this logic within threads that get scheduled by UDS itself, or, for matters which require coordination between replicas, by totally ordering these via the GCS. Some more details on this will be described in Chapter 14,

but for the most part, the brief description above actually captures the gist of our solutions to deterministic reconfiguration quite well.

For additional system parameters, such as the number of replicas, how many Byzantine faults we would like to be able to tolerate, or other system-internal configurations like queue or batch sizes within the SMR library used for prototype implementations, we have to defer to related work and future research. We do so, however, without loss of generality of many of our developed approaches.

12.3 SELF-OPTIMIZATION

Reconfiguration Decisions

Providing a system with deterministic metrics and ensuring reconfigurations during runtime doth not a self-optimizing system make, of course. The heavy lifting of actually deciding new configurations for the system will have to be performed by a further component, which has to have knowledge about the possible effects of all parameters under its control. This may sound simple, but is in fact utterly complicated, because (i) there are a great many parameters in a BFT SMR system that can be manipulated, (ii) each parameter has different effects on the system, and worst of all, (iii) some parameters could possibly affect the effectiveness or (even the effects themselves) of other parameters.

Parameter Space

To illustrate this with an example, consider an SMR system based on vertically scalable virtual machines, where each replica may be scaled from 1 through 16 vCPU cores[1]. Assume further that this system runs a working implementation of our proposed optimization approaches, i.e., can deterministically measure metrics yielding estimators for current system behavior and status of the system's environment, schedules threads with UDS to gain deterministic multithreading, and an optimizer component attempts to optimize the system using only the following parameters: UDS primaries, UDS steps per round, UDS total order, and number of CPU cores via vertical scaling. While the first three of these parameters could be technically infinite, assume that in this example they stay within realistic boundaries. Therefore, let UDS primaries be scaled up and down within $n \in [1, 16]$, steps be limited to within $[1, 3]$ and the kinds of possible total orders be `round-robin`, `all-at-once` (so, e.g., [0,0,0,1,1,1,2,2,2,...]), or `random`. Note that especially the way total orders are determined is a parameter which can possibly lead to an explosion of combinations by itself, if we were to assume a separate optimizer component capable of somehow determining optimal schedules for a given set of application requests (e.g., by learning likely locking patterns and optimizing the order for maximum parallelization and a minimum number of required rounds). Regardless of this last fact, assume that the optimizer component's task is to achieve the maximum possible request throughput while keeping request latencies low.

With only these 4 parameters as specified in the example, we have already established a space of $16 * 4 * 3 * 16 = 3072$ possible configurations to

1 Which is not all that many nowadays, considering that on AWS' EC2 platform, for example, machines with up to 448 vCPUs may be rented on-demand.

choose from. Of course not all of these combinations would make sense (16 primaries, five steps and one vCPU would come to mind, for example), but even after eliminating some obviously inane configurations, we could still be left with hundreds, if not thousands of potentially useful combinations. Next, the optimizer would have to find a strategy to map these combinations, or at least a subset of the most commonly used ones (which also have to be somehow determined first), to the currently observed system status, which either requires knowledge about the effects of parameters on system behavior, or a complete map of all possible combinations to a set of commonly observed circumstances.

Approaches

Measuring the effects of all combinations of these parameters is a task that could feasibly be achieved, if one had the time and resources to run thousands of tests with a well-defined set of application profiles, specifically created with the target application that is to be replicated in mind. For some systems, this effort might be warranted, but one of the subgoals of our research was also to make SMR more accessible, in order to promote its usage. Requiring developers to run days worth of tests for their application to run optimally would certainly be counterproductive to the general adoption of SMR.

Another approach, then, could be the inference of some generally applicable rules regarding the effects of these parameters, i.e., to determine whether some settings almost always result in a certain effect. An example of this would be the aforementioned effect of primaries on throughput and latencies, i.e., our suspicion that a higher number of primaries should generally result in higher throughput—given enough requests to promptly start scheduling rounds—, whereas latencies might also quickly rise in case we choose to many primaries for the current request rate. Based on inferred general rules like this, it would be possible to build optimizers which turn only those few knobs for which the effects are well-known. Just so, Chapter 14 will present our first version of a self-optimizing SMR system based on a simple rule-based algorithm, which can already achieve rather decent performance improvements. However, as we will also see in that chapter, even just validating the results for adjusting a single parameter can be a daunting and time-consuming task.

The next idea for tackling this problem was born on the grounds of currently much discussed applications of Reinforcement Learning (RL) all over the world, where agents are trained to perform a multitude of complex tasks—including the skillful and fine-grained controlling of complicated systems. Our problem of controlling parameters of a live system to reach desired outcomes was seen as a good fit to RL's strengths, which leads directly to the last efforts undertaken in the scope of this thesis.

12.4 REINFORCEMENT LEARNING

Paradigms

Before we simply dive into our work on this subject, we have to admit the rather substantial shift in context that comes with introducing a completely new optimization paradigm, which is itself currently a hot topic of research and by no means just a ready-made tool to simply apply to problems. There-

fore, this section will briefly introduce the most general terms and concepts behind Reinforcement Learning and further motivate our reasons for picking this technique, in addition to the general introduction back in Chapter 4.

12.4.1 *The Three Paradigms*

Basic Introduction to ML

Generally speaking, Machine Learning (ML), both as a field of research with an overabundance of published work, and as a concrete tool used in a myriad of applications and devices of our daily lives, is an entire class of paradigms based on the central idea that some machine or algorithm is fed with data, and that through the magic of mathematics, this algorithm can adapt itself so as to extract useful information from this data. ML is commonly subdivided into three major categories:

SUPERVISED LEARNING Input data is pre-labelled, e.g., by humans, and certain relationships within the data (e.g., categories) are learned by the machine being trained, so that it can later apply learned labels to new data without further need for anyone to pre-label it.

UNSUPERVISED LEARNING Similar to supervised learning, but without the pre-labeling step, i.e., the algorithm is given data with the goal of discovering hidden patterns or relationships within it. The idea is that unsupervised learning can detect classes or patterns within new or unknown, large, and multidimensional datasets on its own, e.g., for classification purposes, which humans might not have had the ability to analyze manually without the machine supporting them.

REINFORCEMENT LEARNING A paradigm that is quite different from the other two, where the machine is interacting with some sort of environment (even if only simulated) through actions, in order to learn optimal strategies for controlling this environment. "Optimal" has to be defined by rewarding or punishing the machine for good or undesirable actions, respectively.

Interest in RL

Given these rather broad definitions, it already becomes a little clearer how RL could be an interesting choice for our particular research problems: Our system constitutes a live environment, and we want an optimizer component to decide good reconfiguration actions that improve the system's performance during runtime. It would of course also be possible to model and abstract our problem differently, e.g., by collecting a vast amount of data from evaluation runs with a multitude of different configurations and application profiles which could then be used with (un-)supervised learning approaches to discover patterns within this data. In a way, this would be similar to our approach of inferring rules between parameters and system behavior, as suggested in the previous section and our rule-based optimization approach in Chapter 14, only that the deductions are performed by training ML models instead of humans.

Feasibility of Alternative Approaches

However, this would require substantial amounts of initial work to gather

sound data about a preferably very large set of diverse application profiles, with as many configuration parameter combinations as possible—which realistically itself can already take months of time, all before any training or optimization has taken place. Of course this is still an entirely feasible approach and a promising avenue of research that could be pursued. Since our premise was to equip SMR systems with a generalizable solution to self-optimize themselves in the face of possibly unforeseen changes in the environment, however, we looked towards RL as a likely better fit for our needs. Additionally, RL is currently being seen as a possible solution to optimizing complex and unpredictable systems (e.g., [95], [51], [66], [74], [79], [67]). After handling our Byzantine fault-tolerant setup using multiple interconnected machines and distributed consensus algorithms with configurable and deterministically multithreaded parallel execution of requests, we were convinced that its complexity was a good enough reason to favor an RL approach for our next contribution.

Current RL Applications

12.4.2 *Preliminary Background Information on Reinforcement Learning*

Having decided to take a closer look at RL as a tool to achieve our research goals, we needed to determine how exactly RL works, and what the requirements for using RL are. A brief summary of these investigations is given here, and a more detailed description of RL's background and our resulting approach will be provided in Chapter 15.

RL Basics

To begin with and put as simply as possible, Reinforcement Learning employs a so-called *agent*, which interacts with an *environment* through *actions*. This is, in its most basic form, what a Markov Decision Process (MDP) models, if one's particular problem can be regarded as such [104]. When a system is modeled as an MDP, it consists of a set of states S and a set of actions A. Choosing an action $a \in A$ when the system is in state $s \in S$, will transition to a new state $s' \in S$ with a given probability $P_a(s, s')$. Additionally, after this transition to s', a reward $R_a(s, s')$ is specified, which signals how "desirable" or "good" the chosen action a in state s was, depending on the semantics of the modeled problem. Therefore, the environment transitions from state to state under certain probabilities when given actions.

Terminology

Since time is thereby advanced in discrete steps, transitioning from one state to the next in an environment is usually called taking a *step* in RL[2]. With RL, the most common goal is to train an agent to influence the environment with its actions so that rewards are overall maximized over time. Hence, the training process utilizes the rewards received after each transition from s to s', and feeds them back to the agent, so it can judge its actions and learn an optimal

2 There is potential for a bit of confusion here, as we now have *actions* and *steps* within the context of RL, in addition to *critical actions* and *steps* as introduced in our discourse about UDS. We strive to simply separate these terms for the reader in the remainder of this thesis by stressing the proper context where necessary.

policy $\pi(s)$ for maximizing accumulated rewards over time[3]. In complex systems, the agent is usually not aware of the entire environment during this process, i.e., it is only fed an *observation* of the environment's current status after each step. Figure 12.1 shows this training loop in its basic form. In Chapter 15 we will see a slightly expanded version of this diagram, illustrating how the agent can be represented by a deep neural network (DNN, cf., [44]).

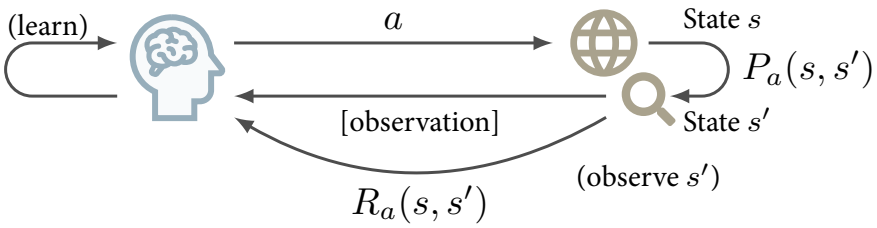

Figure 12.1. The basic training loop in a Reinforcement Learning setup. After an agent has been fully trained, the sending of rewards and learning can cease, and the agent will produce actions according to its learned policy when given observations.

Initial Problem Statement

With this basic idea of Reinforcement Learning in our minds, we can already identify some problems that need to be solved in order to employ RL for optimization of our system. First of all, it would be best to model our environment according to the rules outlined above, i.e., transform our system so that it (i) can progress in discrete time steps, (ii) can receive actions which reconfigure its parameters, (iii) provides a sensible observation of itself after each step, and finally (iv) has a component judging the transitions for providing a reward. Additionally, we would need to find, implement, and employ a preferably rather well-researched RL technique for learning optimal policies within this setup, so as to train agents that can optimize the system to some degree. As will be explained later, our choice fell on Deep Q-Learning (DQL) as a technique, which employs neural networks called Deep Q-Networks (DQN) to approximate the otherwise prohibitively large mappings between actions and expected rewards that complex environments often yield (cf., Section 15.2). This invited some unexpected difficulties, such as suddenly having to care for several dozens of virtual neurons, which sometimes can be as ~~stubborn~~opinionated as real children and just simply refuse to learn what we would so much like them to learn. Afterwards, a phase would follow in which agents are trained, fine-tuned, and evaluated, to find, or rather, create an agent that achieves adequate results (i.e., which can optimize our system in the face of variable load and application profiles). Finally, we would require a way to transfer this trained agent to a live system, where it could play the part of our optimizer component.

The challenges we had to overcome during this part of our research were substantial, not least due to the fact that we were almost completely unfamiliar with applying ML techniques to real problems, let alone with handling the finer

3 An agent is seen as the entire entity influencing the environment, e.g., including actuators or supporting libraries, while the policy is purely the strategy or logic within the agent responsible for choosing actions

intricacies of implementing and handling a Reinforcement Learning setup, when we started with this approach. Nonetheless, we managed to progress significantly on this roughly outlined roadmap of problems that would need to be solved, and will summarize our efforts on attempting to create a neural network-based, generalized solution for self-optimizing systems with greater complexity in Chapter 15.

13

To solve the issue of requiring a deterministic measurement as a basis for reconfiguration, we proposed a new deterministic Byzantine fault-tolerant distributed clock mechanism, which guarantees deterministic time intervals for all participants. These time intervals can then in turn be used for measuring performance to some degree of accuracy. The rest of this section will present our novel mechanism, called **By**zantine **T**ime **I**ntervals, or ByTI for short.

The algorithm itself was initially co-developed and co-analyzed by our research group as a team, with Prof. Hauck supplying the initial analysis of the algorithm's behavior under different failure cases, while later implementation, bug fixing, analysis, and evaluation tasks were then contributed by the thesis author.

13.1 RELATED WORK

Fundamental Approaches

In principle, there are two approaches to arrive at deterministic decisions in each replica. First, we could individually measure and then distribute values among all machines, agreeing on a value, similar to how atomic multicast protocols are already used to agree on a total order of messages in SMR. The second way would deterministically measure a metric in each replica, based on some notion of identical time in each participant.

Measure & Broadcast

For individual measurements that are broadcast to all replicas, we would need a way to deterministically select which values out of all received measurements are valid, and then implement a deterministic fusion function creating a final value to be used in each replica. The distribution process has to define a deterministic point in time when adaptation starts, e.g., the last message sent for distributing one of the values.

Distributing Measurements

Surprisingly, only little related work can be found regarding this topic. To the best of our knowledge, the only similar, published mechanism can be found in a master's thesis titled *ByTAM* [107], and is based on dissemination of observed values via the consensus protocol. This approach was then further advanced in a second master's thesis [103], where the author allowed sensors to operate in different fault modes—namely non-replicated, crash tolerant or Byzantine tolerant. If all replicated sensors perform measurements on the same metric, replicas can filter potentially faulty readings. In ByTAM for instance, all sensors broadcast their readings to all other replicas. Each of them waits for $\lceil (N+f)/2 \rceil$ sensor readings of different replicas before it deletes the f highest and lowest values and uses a deterministic function to build a mean value from the remaining readings. However, the authors aim at a general purpose monitoring system that uses SMR for replicating the measurement

components, instead of equipping an existing SMR system with monitoring capabilities.

Timeliness

Additionally, there is no guarantee that the results are timely, i.e. it is not known whether the values sensors sent were read in roughly the same time frame or were potentially scattered over a larger period of time, and are therefore hardly comparable. For example, since replicas wait for a quorum of sensor readings, at least one of them could (maybe maliciously) arrive considerably later than the others, up to the maximum delay allowed by the employed consensus protocol, which would mean decisions are made on potentially stale sensor values. Detailed research about the accuracy and decision delay or timeliness of such an approach does not seem to exist yet.

Deterministic Measurements

Local Time Intervals

For localized deterministic measurements in each replica, a deterministic time interval that each replica can use to measure is required, in order to, e.g., count arriving requests within that interval. The determination of such time intervals would have to happen based on some form of communication and consensus between the replicas. With this approach, the measurement itself would be deterministic, but the time base may not exactly reflect wall-clock time. Further, this solution would no longer require a fusion process (i.e., discarding of potentially bad values), as every replica simply measures its own values within the given deterministic time frame, which are guaranteed to be identical between participants as long as the correct metrics are chosen. A good and useful example for such a metric would be request counts per interval, which could directly be used to closely estimate current actual request arrival rates.

Approach

ByTI uses periodic messages sent by each replica to all other replicas to establish such deterministic time intervals, so it falls into this category. In this regard, it is similar to mechanisms like watermarks in stream processing, e.g. [11], and Byzantine clock synchronization algorithms, e.g. [33]. However, these ideas are not meant for defining deterministic time intervals for measurements. Therefore, to the best of our knowledge, there is no previous work with this approach as yet.

13.2 DEFINING THE ALGORITHM

We first describe our algorithm to create deterministic Byzantine time intervals, which create reliable and deterministic virtual timelines in each replica. Each virtual timeline is defined by a sequence of messages in the totally-ordered messages arriving at each replica, and each correct replica will identify the same sequences, called ByTI.

Assumptions

Model

We assume that each replica sends clock tick messages—in the following called

ticks for short—to each other replica with a constant rate R. The delay between two sequential ticks by the same node is thus $T := 1/R$. Malicious replicas may not send ticks at all, or use arbitrary and dynamically changing rates. With the latter, malicious replicas may try to distort the construction of ByTIs in correct replicas. Ticks are sent to all replicas via the same atomic multicast which is also used by the SMR system for ordering requests, thus also having a totally ordered sequence number in the total order, as any regular request would have. As clock ticks are inserted into the total order of all arriving messages, each replica sees all clock ticks in the same order and appearance with respect to other messages. Further, tick-messages are authenticated and only considered when originating from a valid sender. Since the messages originate from replicas participating in the consensus protocol, and since most modern consensus protocols already inherently check message authenticity, this requirement is usually fulfilled by default.

Algorithm

The goal of the following algorithm is to determine a ByTI with a length close to $m * T$, with $m > 0$ being an integral configuration parameter. First, the algorithm maintains a set of variables:

```
1  firstNo := 0                # first message in ByTI
2  lastNo := undef             # last message in ByTI
3  replicaTicks := [0,0,0,..] # tick counts per replica
4  goodReplicas := 0           # with exactly m ticks
5  badReplicas := 0            # with more than m ticks
```

`replicaTicks` is an array storing the received number of ticks per replica, where each position in the array is accessed by the replica ID, which is supposed to run from `0` to `N-1`. With each incoming tick message `msg` from replica `msg.sender`, the following algorithm is executed:

```
6  process( msg ) {
7     ticks := ++replicaTicks[msg.sender]
8     imprecise := false
9     # received exactly m ticks from this replica?
10    if( ticks == m ) {
11       if( ++goodReplicas ≥ N - f ) {
12          # Preliminary interval
13          lastNo := msg.seqNo
14       }
15    }
16
17    # received a tick too many from this replica?
18    if( ticks == m + 1 ) {
19       # One more replica with too many ticks
20       goodReplicas--
21       badReplicas++
```

```
22        imprecise := badReplicas > f
23     }
24
25     # close because m ticks from N replicas received?
26     # or close b/c regular interval is impossible?
27     if( goodReplicas == N ||
28        ( ticks == m + 1 &&
29        ( imprecise || goodReplicas ≥ N - f - 1 ))) {
30        # Decide
31        replicaTicks := [0, 0, 0, ...]
32        goodReplicas := badReplicas := 0
33        if( lastNo == undef || ticks == m + 1) {
34           # some kind of fault detected: close interval
35           # but reuse this tick in the next ByTI
36           lastNo := msg.seqNo - 1
37           replicaTicks[msg.sender] := 1
38           if( m == 1 ) {
39              goodReplicas := 1
40           }
41        }
42        decideByTI(firstNo, lastNo, imprecise)
43        firstNo := lastNo + 1
44        lastNo := undef
45     }
46  }
```

Deciding an Interval

In other words, this algorithm waits for exactly m ticks from at least $N-f$ different replicas. Are these received, the interval is temporarily *decided*, marked by `lastNo`, meaning it is finished and can be used by a reconfiguration component to measure metrics or determine possible reconfigurations of the system for optimizations. If there is still a chance to increase the number of nodes who have sent exactly m ticks, the interval will be extended. Otherwise, the interval is simply decided, and the next one is started.

Edge Cases

Theoretically, it can happen that the algorithm is not able to find at least $N-f$ replicas with exact m ticks, detected by having already more than f replicas with more than m ticks. In this case the interval is decided, too. However, since the actual wall-clock time of the interval is less clear in this case, it is marked with an additional `imprecise` parameter. This situation can occur for example in case of high jitter in the message delivery timings, or when not yet all replicas are operational—initially, after a reconfiguration or in a network partition. It is up to the decision function whether it wants to use imprecise intervals for measurements and further decisions or not.

In the following, we analyze how the algorithm behaves in case there are no faults, when there are crash-only faults and in case of Byzantine faults. For this analysis, we initially simplify a little by assuming that there is a constant network delay for ticks. In reality, message delivery will experience jitter, and we will discuss its consequences later.

Assumptions for Analysis

All the shown examples—except the Byzantine shortening attack in Section 13.2.3—will be for $N := 4, f := 1$ and $m := 2$, whereas our general

considerations remain independent of concrete numbers[1]. In all plots, T was set to 100*ms*. The variable B denotes the distribution of the length of decided ByTIs, while D denotes the distribution of the decision delay of a ByTI.

13.2.1 *Analysis: No Faults*

Figure 13.1 shows an example, recorded from a real evaluation run with no faults, and cropped to a small-time window with only 2 intervals visible. Ticks are sent with some arbitrary phase shift between replicas.

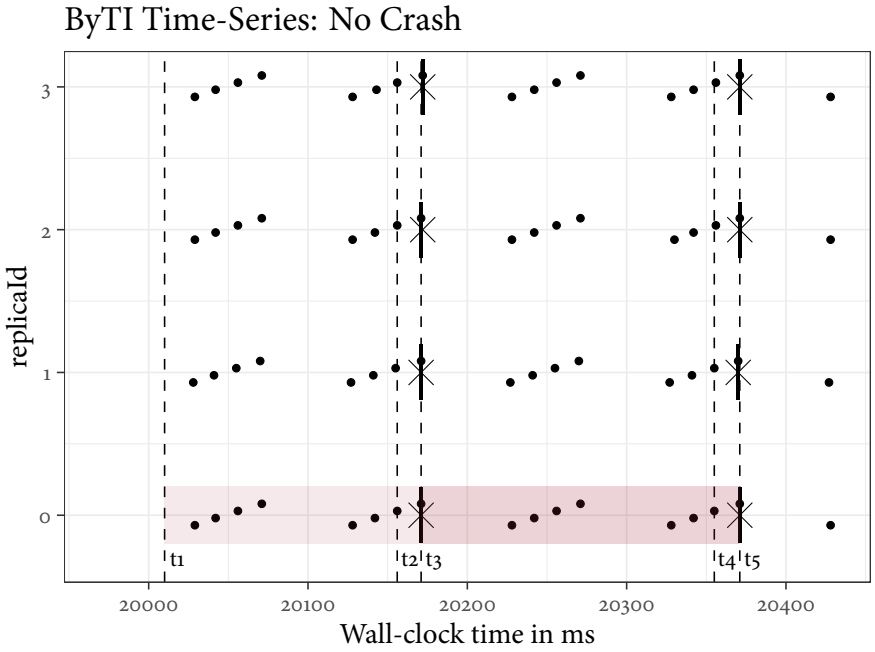

Figure 13.1. Time-Series plot, zoomed into a section of a recorded benchmark run, showing ByTI tick messages and intervals for each replica. Each dot is a received tick message from a replica, where y-position-shift in each replica swim-lane indicates which replica a tick was received from, analogous to the distribution of replicas along the y-axis (note that replicas also send ticks to themselves), each vertical line represents an interesting point in time (cf., explanatory text in Section 13.2.1), and the points in time when decisions about the respective ByTI were made are marked with X. The differently shaded areas show fully decided intervals, exemplary for one replica.

Starting ByTI

The algorithm can theoretically be started at an arbitrary point in time. For a general analysis we can therefore ignore the real-time axis in Figure 13.1, and consider instead the indicated relative point in time t_1 as an arbitrarily picked starting point. The initial ByTI is first preliminarily and transparently detected at t_2 within the algorithm (cf., Line 13), but extended to t_3 once the last tick of R_4 has been received, when it is decided to range from t_1 to t_3, as no more ticks for this interval can be expected. Note that when starting the algorithm

1 To properly demonstrate the shortening attack in a readable plot, we will use $m := 1$ for this case.

at an arbitrary point in time, e.g., with the initial start of the system, the very first ByTI can be expected to be shorter than $m * T$.

Example

The next ByTI is then determined to range from t_3 to t_4. As can be seen, replica R_3 gives the pace for time intervals in this example. All subsequent intervals will always be $m * T$ long in this fault-free case. It is interesting to note that different phase shifts will not change this behavior, since after the tick of some replica closes the first interval, this replica's ticks will then also close all following ones, as long as the initial phase shift stays roughly the same. This also depends on the clock-drift of each replica-local hardware wall-clock, which can be considered negligible for current consumer hardware and the time ranges we are considering [2]. If all nodes are behaving correctly, and after removing the first, potentially shorter, interval, B will have the following characteristics: $E[B] := m * T$, $\sigma_B := 0$. The decision time is always zero as the last m-th message closes the interval and decides it, as can also be seen in the example: $E[D] := 0$, $\sigma_S := 0$.

13.2.2 *Analysis: Crash Faults*

Figure 13.2 shows a similar example from another recorded evaluation run, but here replica R_4 has crashed and is not sending ticks anymore.

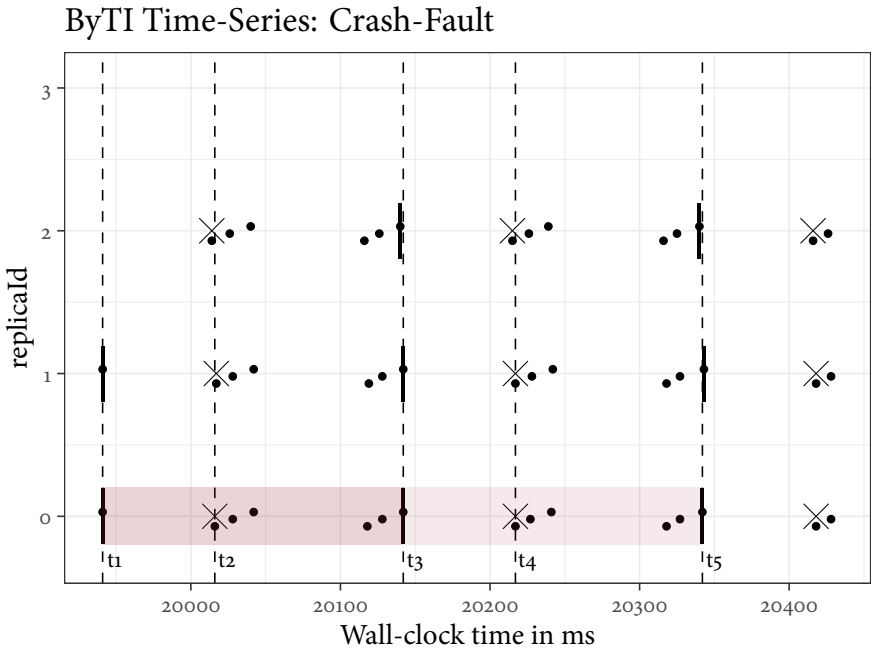

Figure 13.2. Time-Series plot, zoomed into a section of a recorded benchmark run, showing ByTI tick messages and intervals for each replica after one replica (R_4) has crashed. Legend is identical to Figure 13.1

ByTI Under Crash-Faults

In this case, the detection of intervals works differently. Since one partici-

2 Even if a replica had a dramatically drifting clock and would slowly mess up its tick rhythm, the influence on ByTIs can be expected to be minimal, as we will see shortly, when talking about Byzantine attackers.

pant is missing, the algorithm detects the $(m+1)$-th tick at t_4, and therefore retroactively closes the interval at t_3. All detected intervals have the same correct length of $m * T$, because the ByTI algorithm needs only $N - f$ ticks to complete an interval.

For crashes, and after removing the first interval, which could again be shorter after the algorithm starts, B has the same characteristics as without faults: $E[B] := mT$ and $\sigma_B := 0$. The decision delay depends on the distance between the last m-th message considered for the interval and the $(m+1)$-th message that actually decides it. This distance has uniform distribution $\mathcal{U}(0, T)$. Thus decision characteristics are: $E[D] := \frac{1}{2}T$ and $\sigma_D := \frac{T}{2\sqrt{3}}$.

13.2.3 *Analysis: Byzantine Faults*

For Byzantine cases, we distinguish between two attacks (with random Byzantine faults falling into one of the two categories by chance): Malicious replicas may either try to *extend* or to *shorten* B.

Figure 13.3 shows a malicious replica R_3 trying to extend the ByTI in all replicas.

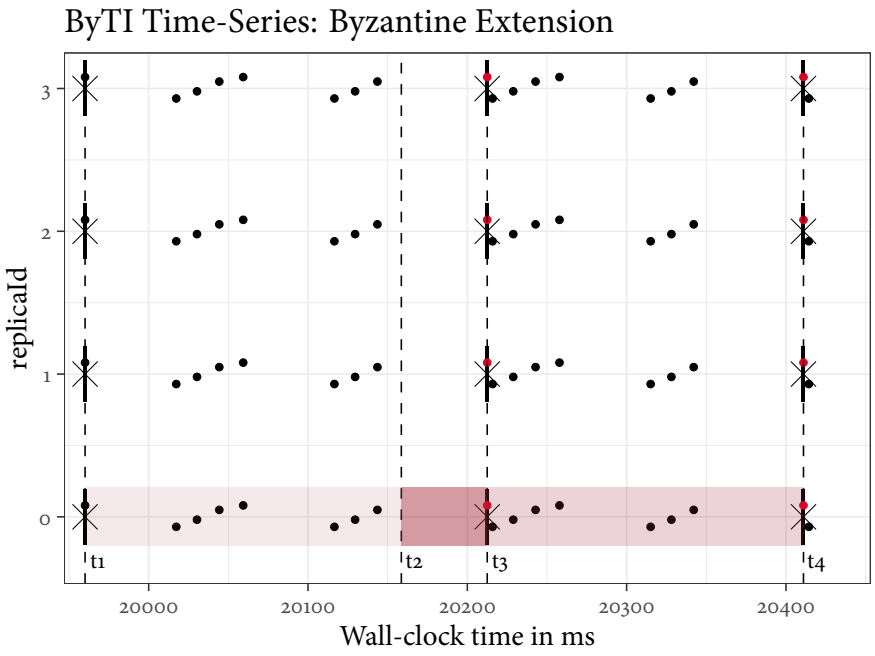

Figure 13.3. Time-Series plot of an artificially created demonstration showing ByTI tick messages and intervals for each replica if one replica (R_3) is trying to maliciously extend ByTIs. Legend is identical to Figure 13.1, and red ticks are the malicious ticks inserted by R_3.

Possible Attacks

As shown in the previous Section, simply omitting ticks would not help R_3, since this would be functionally equivalent to a crash-fault and the interval lengths would be unaffected. Following a more sophisticated approach, R_3 would send its m-th message shortly before the $(m+1)$-th message from a benign replica is delivered. This m-th tick by R_3 counts towards the current

interval, thereby extending it, because the algorithm tries to wait for exactly m messages from as many replicas as possible.

Ineffectiveness of Lengthening Attacks

After the first extension, however, the malicious replica cannot extend any further ByTIs: In Figure 13.3, we assume a previous, correctly closed ByTI up to t_1 had the regular length $m*T$, at which point R_4 decides to try its extension attack. The expected length of the interval starting at t_1 would normally reach until t_2, which the attacker is able to extend to t_3. However, another delay of R_3's m-th message would simply result in a ByTI of normal length $m * T$. Even worse, if the attacker happens to come too close to the next $(m + 1)$-th tick message of a benign replica (in our case R_0 near t_4), its messages may be delivered after this tick. In this case, the ByTI would be closed with the last regular m-th message. R_3 could conceivably try to shorten the interval starting from t_3—an attack we will look at in the next paragraph—, but this would simply cancel out the previous extension attack's effects. Therefore, we can reason that an attacker cannot effectively extend ByTI lengths, except for one negligible interval during the entire runtime of the system: $E[B] := mT$ and $\sigma_B := 0$. Decisions are once again instant, since in this attack, the delayed m-th tick is similar to a correct m-th tick and therefore closes the interval: $E[D] := 0, \sigma_S := 0$.

Shortening Attacks

Unfortunately, despite the robustness of our algorithm against the previous cases, a Byzantine replica can constantly shorten intervals. To achieve this, it initially sends its m tick messages within a shorter period than usual and then waits until $N - f - 1$ other replicas have sent exactly m ticks each. Then, the attacker sends its $(m + 1)$-th tick, thus immediately closing the ByTI at the time of the last m-th message arrival. Figure 13.4 shows this malicious attack for shortening ByTIs. For this example, to improve readability and keep the plot within the bounds of the page, m is set to $m := 1$, contrary to the previous examples.

The shaded areas in R_0's swim lane with timestamps t_2 through t_7 show the intervals after the attack, whereas the shading in R_1's lane and the timestamps with subscript $_e$ show what the expected outcome would be without an attack.

Example of Shortening Attack

To set up the attack, R_3 initially sends its m regular ticks before all other replicas, so its tick-count in the `replicaTicks`-array is prepared with m (first red tick in interval $[t_1, t_2]$ in Figure 13.4). Then it waits until $N - f - 1$ other replicas, in our example R_0 and R_1, have sent their m-th tick before immediately sending its $(m + 1)$-th tick. This results in the interval being closed retroactively at t_2 instead of the originally expected $t2_e$, shortening it considerably. All subsequent, shortened intervals work the same way. Note that the pattern of shortened ByTIs repeats after just 3 intervals. We further can see in the example that the three ByTIs $[t_2, t_3]$, $[t_3, t_4]$, and $[t4, t_5]$ have a total length of $2 * T$, as R_1 is used to open and close the whole sequence.

For a generalized analysis, we define P_i as the phase shift of R_i to R_1 in terms of how much later R_i sends its ticks compared to R_1. All P_i are independently and identically distributed random variables with a uniform distribution $\mathcal{U}(0, T)$. Without loss of generality, we assume an R_1 tick closed the previous interval. The length of the next ByTI in case of a shortening attack is

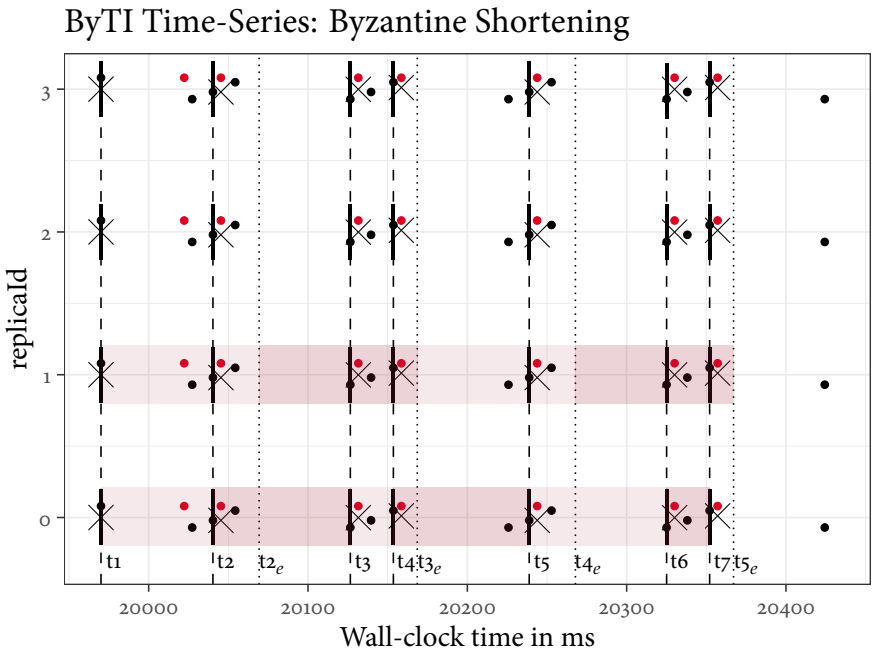

Figure 13.4. Time-Series plot of an artificially created demonstration showing ByTI tick messages and intervals for all replicas and $m =: 1$. One replica (R_4) is trying to maliciously shorten ByTIs. Legend is identical to Figure 13.1, and red ticks are the malicious ticks inserted by R_4.

then defined as $B := (m-1) * T + max(P_i)$ for all P_i of the $N-f$ correct replicas[3], as the attacker pushes R_1's m-th tick to the next ByTI but needs the m-th tick from all other correct nodes. This results in the following density function, expectation and standard deviation for B:

$$f_B(x) := \begin{cases} 0 & \text{if } x < 0 \\ \frac{N-f-1}{T^{N-f-1}}\, x^{N-f-2} & \text{if } 0 \leq x < T \\ 0 & \text{if } x \geq T \end{cases}$$

$$E[B] := \left(m - \frac{1}{N-f}\right)T$$

$$\sigma_B := T\sqrt{\frac{N-f-1}{N-f+1} - \left(\frac{N-f-1}{N-f}\right)^2}$$

A sequence of $N-f$ ByTIs under shortening attack therefore has a length S with an expectation value of $E[S] := (N-f) * m * T - T$. Even more interestingly, the standard deviation can be computed as $\sigma_S := 0$ as the phase shifts absorb each other.

3 Note that P_1 is zero and not considered

For $N = 4$ and $f = 1$ this results in the effect of a shortening attacker on a ByTI length to be expected as $(1 - \frac{1}{3*m})m * T$. The interval is $\frac{1}{3m}$ shorter than expected, so in our example, with $m := 1$, this would shorten ByTI by approx. 33%. Increasing m therefore mitigates the effects: With $m := 3$ the expected length an attacker can achieve would only be 11% shorter than normal. Hence, by adjusting m, the maximum expectable effect of an attacker can be configured, trading more tick messages against better ByTI accuracy in case of Byzantine faults.

For an attacker to achieve maximum effect, it would have to send its $(m+1)$-th message before the m-th tick of the last correct replica. Therefore, in order to delay decisions as far as possible, we assume the attacker sends its tick as late as possible, which results in $E[D] := \frac{1}{2}T$ and $\sigma_D := \frac{T}{2\sqrt{3}}$.

13.2.4 *Analysis: Network Delays*

Expected Effects of Jitter

In real systems, messages of course do not arrive with constant network delay, and will jitter, i.e., have varying arrival times. The arrivals of ticks of the same correct replica will therefore not occur exactly after T time units. These effects of jitter have the potential of extending or shortening individual ByTIs. However, averaged out over longer periods of time, one T will still contain one tick. Additionally, possible reordering of expected messages also influences the algorithm, especially when it comes to deciding the interval. For example, a delayed m-th message may not be considered because an earlier $(m + 1)$-th message already closed the interval. Even worse, the m-th message will then count as a first message for the next interval.

In practice, as we have seen in our evaluations, a very small phase shift between replicas emphasizes this behavior, whereas a more equally distributed phase shift can tolerate more jitter. A formal model of the algorithmic behavior including network jitter shall be omitted here as it has little to offer in addition to the previously shown models. Also, our evaluation in the next section will show the stability of ByTI in a common setup with $N = 4$ real replicas connected via a normal gigabit switch.

13.2.5 *Enabling View Changes*

For a truly runtime-reconfigurable system, the replica group needs to be reconfigurable, too, i.e., replicas need to be able to leave and join the ensemble. In existing BFT SMR systems, whenever the group of replicas changes, the underlying atomic broadcast algorithm has to carefully participate in this change. Since typically the ordering of messages is based on a consensus algorithm, this changing of cluster members is called a *View change*, in which the View number is incremented, and new consensus instances have to follow suit.

A simple, yet sufficient procedure for the ByTI algorithm to support such View changes is to close the last intervals within the old view with `imprecise` set to true, and doing the same for the first interval in the new View.

13.3 EVALUATION

We implemented the ByTI algorithm as presented in the previous section, and integrated it into our existing evaluation platform based on BFT-SMaRt. Afterwards, we set up experiments to measure actual interval lengths for $N = 4$, $f = 1$, $T = 100ms$, and $m = \{1, 3\}$ in the fault-free and crash cases.

Omitting Byzantine Evaluations

The results worked well and were very close to our theoretical models that we could reasonably assume our analyses to be sound, and Byzantine cases to look similar. Therefore, we opted to not spend significant time on implementing sophisticated Byzantine attackers, which would have to precisely measure other replicas' arriving tick intervals and system-specific network details like delays and jitter before becoming effective.

Hardware Setup

All tests were run on a cluster of 4 replicas with Intel i7-7700 CPUs at stock clock speeds, i.e., 3.6 GHz, with 32 GB of RAM per machine, at standard JEDEC DDR4-2400 timings. An additional 4 machines with Intel Xeon E31220 CPUs and 16 GB of RAM each were used to instantiate clients to put load on the replicas. The clients were connected to the replicas via a regular single Gigabit switch, while the replicas were additionally connected to each other via a second NIC and a second Gigabit switch to aid BFT-SMaRt's consensus protocol.

Methodology

In each test a small load was generated for 45 seconds by the client machines to stress the replicas' CPUs and the GCS during out ByTI measurements. Clients connected in groups of 4, with each group sending roughly 500 requests per second to the cluster. Requests were designed to be parallelizable and burn a small amount of CPU time. For warming up all systems before taking measurements, we included a 10-second pre-measurement phase in which client machines could establish connections and fire off some initial requests in each evaluation run.

Then, starting at $t_0 = 10s$, the first client group started sending their requests, 5 seconds later the next group of 4 clients started sending requests, and so on until after 15 seconds all 4 client groups were connected and stressing the system with about 2000 requests per second peak load. Each group stayed active for 30 seconds, so the first group stopped sending requests at $t = 40s$, then 5 seconds later the second group disconnected, and so on, until after 45 seconds no clients were active anymore and the evaluation run was shutdown to gather, analyze, and plot results. For the crash-fault evaluations, one replica was deliberately crashed after 28 seconds, and the remaining time of these tests the cluster continued operating with 3 replicas to record the behavior of our algorithm when coping with missing ticks. As mentioned above, T was set to $100ms$ in all evaluations, i.e., every replica sent out 10 evenly spaced tick requests per second to all other replicas.

13.3.1 *Evaluation: No Faults*

For our first tests, Figure 13.5 and Figure 13.6 show the distributions of ByTI lengths for the most basic case, with $m = 1$ and all replicas operating normally during the entire test.

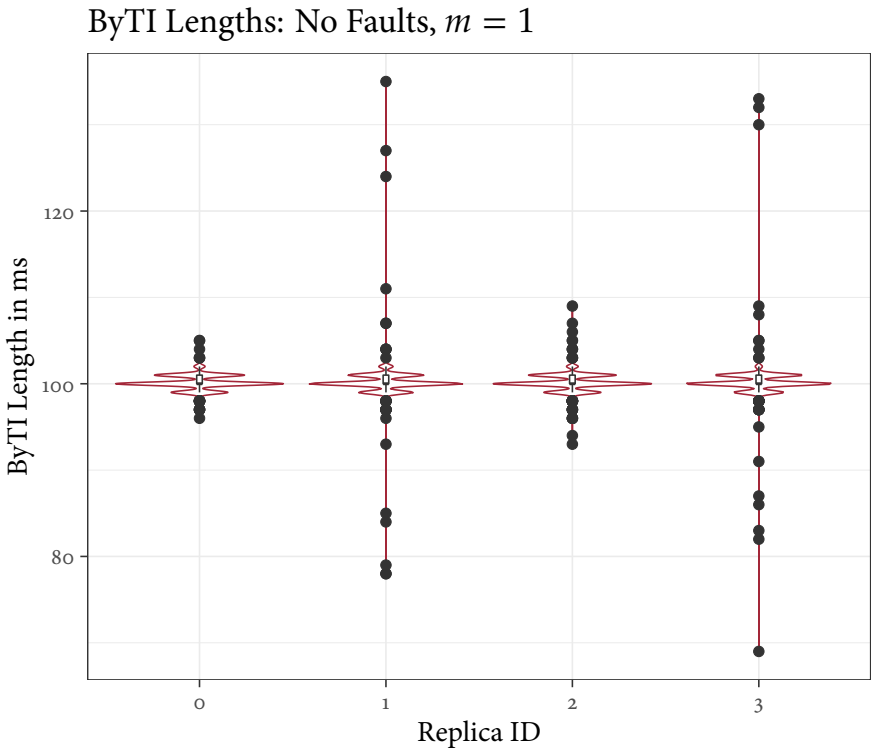

Figure 13.5. Combined Violin- & Box plots, showing distribution of ByTI lengths for $m = 1$ and no faults for each replica.

Results

Due to ByTI's automatic correction of variances, the overall mean length of intervals $mean(ByTILengths) = 100.13ms$ (identical in all replicas) is very close to the expected interval length of $T = 100ms$, with less than 0.2% of deviation. The standard deviations vary between replicas, from a minimum of $sd(ByTILengths) = 0.96ms$ in replica R_0 (as the leader of SMR cluster), to a maximum of $sd(ByTILengths) = 3.30ms$ in replica R_3. Figure 13.5 provides a more detailed insight of the distribution of interval length distributions across replicas.

The vast majority of intervals are close to our expected ByTI length of $100ms$. The few spikes in ByTI lengths in non-leader replicas happen consistently throughout different evaluation runs, settings, hardware and most other conditions, and we could not get rid of them completely, though not for a lack of trying. When looked at in detail, these shortened and extended intervals always happen due to delayed consensus instances, i.e., an entire set of several consecutive message batches being delivered with slight delays, resulting in longer intervals immediately followed by shorter ones. The effects seem to be

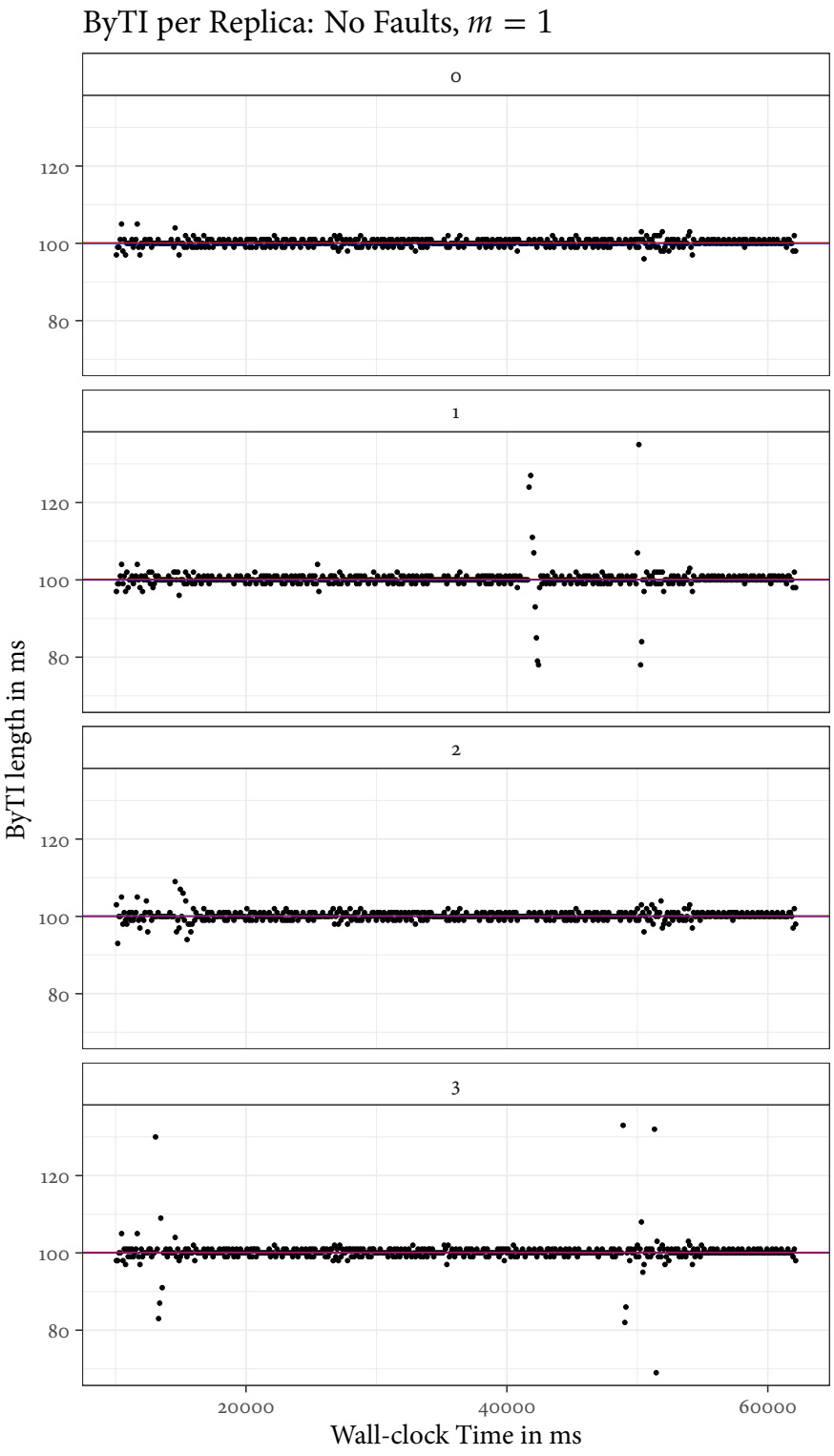

Figure 13.6. Scatter plot of all ByTIs across all replicas, with $m = 1$ and no faults. Each dot represents a closed ByTI; the individual plots show each replica's view.

inherent to our BFT-SMaRt-based benchmarking setup, as we have also seen similar hiccups in other evaluations throughout the last few years.

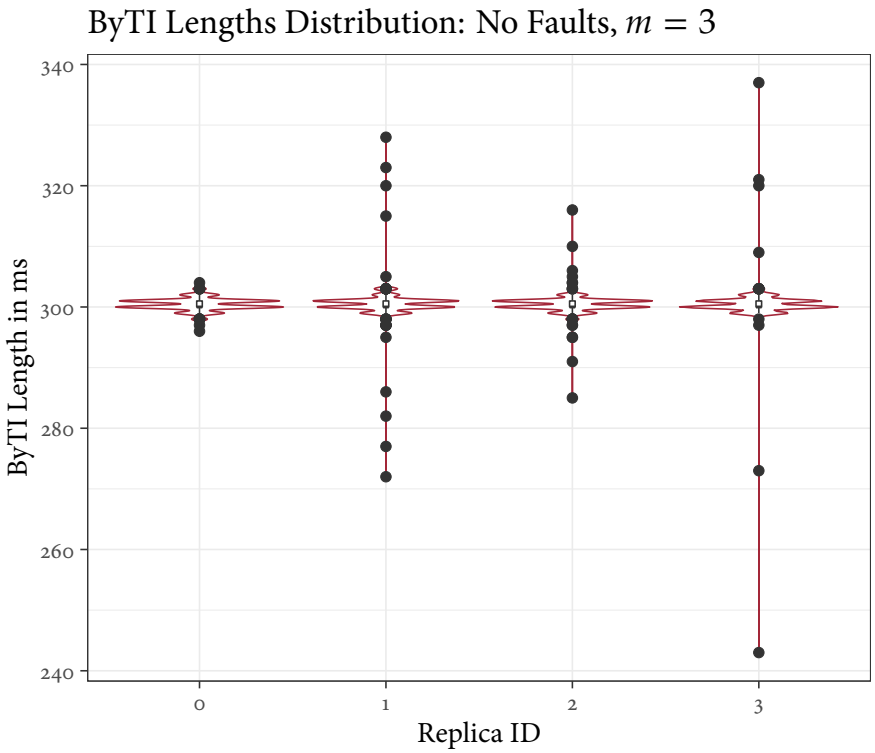

Figure 13.7. Combined Violin- & Box plots, showing distribution of ByTI lengths for $m = 3$ and no faults for each replica.

Increasing m

Even though these results already show that our algorithm works for our intended purpose, they can be further improved by increasing m. Figures 13.7 and 13.8 show the lengths of individual ByTIs during normal operation of the cluster for $m = 3$. The mean interval lengths range from $mean(ByTILengths) = 300.41ms$ in R_0 to $mean(ByTILengths) = 300.44ms$ in R_3, and are thereby even closer to our expected value of $300ms$ on a percentage basis than with $m = 1$. Proportionally, the standard deviation also improved considerably, ranging from $sd(ByTILengths) = 1.13ms$ in R_0 to $sd(ByTILengths) = 6.12ms$ in R_3.

With this example and $m = 3$, a decision about reconfiguration of the system for optimization purposes could still be reached multiple times per second with good stability and timeliness of the measured values.

Negligible Overhead

One drawback of our approach is the necessity of sending additional messages, putting additional stress the consensus and overall our cluster of machines. However, we argue that this increase in load is minor and acceptable: Considering that we put the replicas under a load of about 2000 requests per second for these tests, and with our chosen $T = 100ms$ and $N = 4$, tick requests amount to about 40 additional requests per second, which represents an added overhead of only about 2%. This overhead improves when T is increased, e.g., when reconfiguration decisions are required less often.

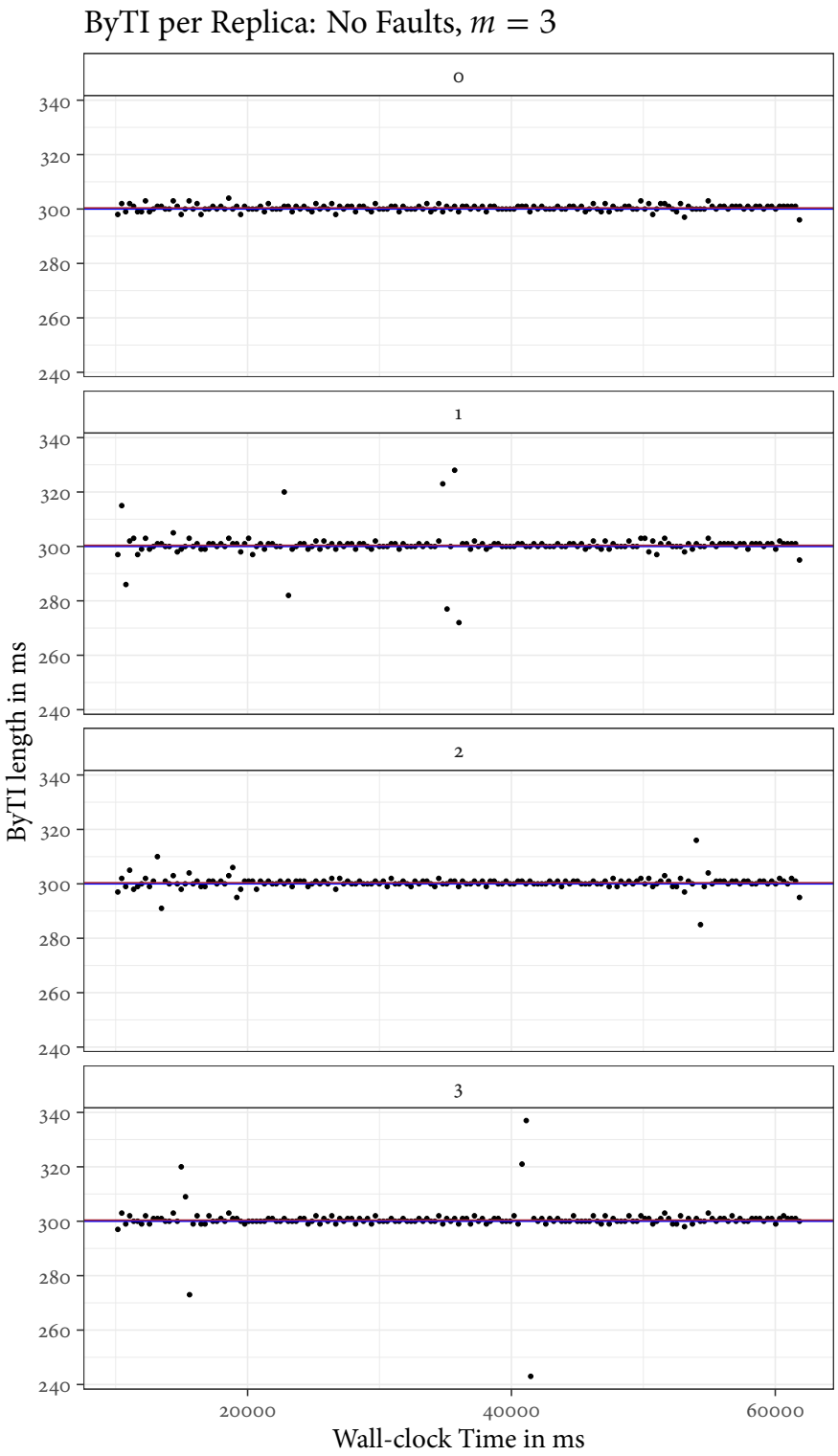

Figure 13.8. Scatter plot of all ByTIs across all replicas, with $m = 3$ and no faults. Each dot represents a closed ByTI; the individual plots show each replica's view.

13.3.2 *Evaluation: Crash Faults*

As shown in our analysis in Section 13.2.2, crashing replicas should not affect ByTIs.

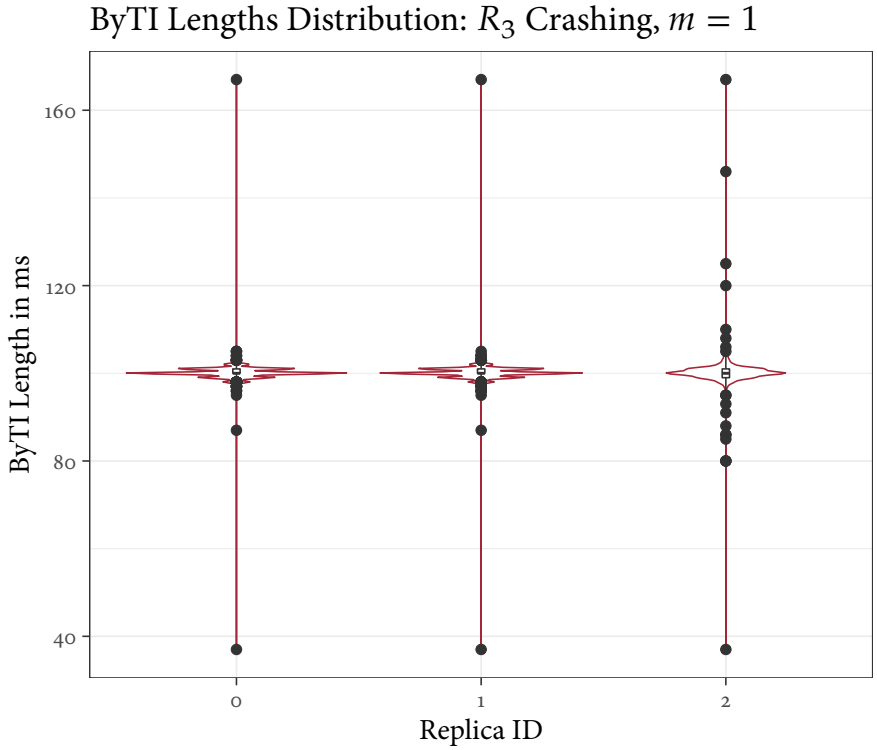

Figure 13.9. Combined Violin- & Box plots, showing distribution of ByTI lengths for $m = 1$ and R_3 crashing after 28 seconds.

Confirmation of Resilience Against Crashes

We set out to confirm this, and ran another set of evaluations with the same cluster, $m = \{1, 3\}$ and $T = 100ms$. The results for $m = 1$ can be seen in Figure 13.9 and Figure 13.10, and are as expected.

The crash does not affect ByTI lengths, except for one negligibly shortened interval directly when the crash occurs. The mean of intervals lengths is $mean(ByTILengths) = 100.12ms$ between all replicas, and standard deviation ranges from $sd(ByTILengths) = 4.24ms$ to $sd(ByTILengths) = 5.31ms$, ignoring R_3's data, of course.

Results of the last evaluation, for $m = 3$, are shown in Figure 13.11 and Figure 13.12.

Note that due to the low variability in interval lengths, plots for $m = 3$ with crashes are zoomed in more than previous plots. The mean in this last benchmark is $mean(ByTILengths) = 300.40ms$ between all replicas, and standard deviation ranges from $sd(ByTILengths) = 1.66ms$ to $sd(ByTILengths) = 2.24ms$, again ignoring R_3's data.

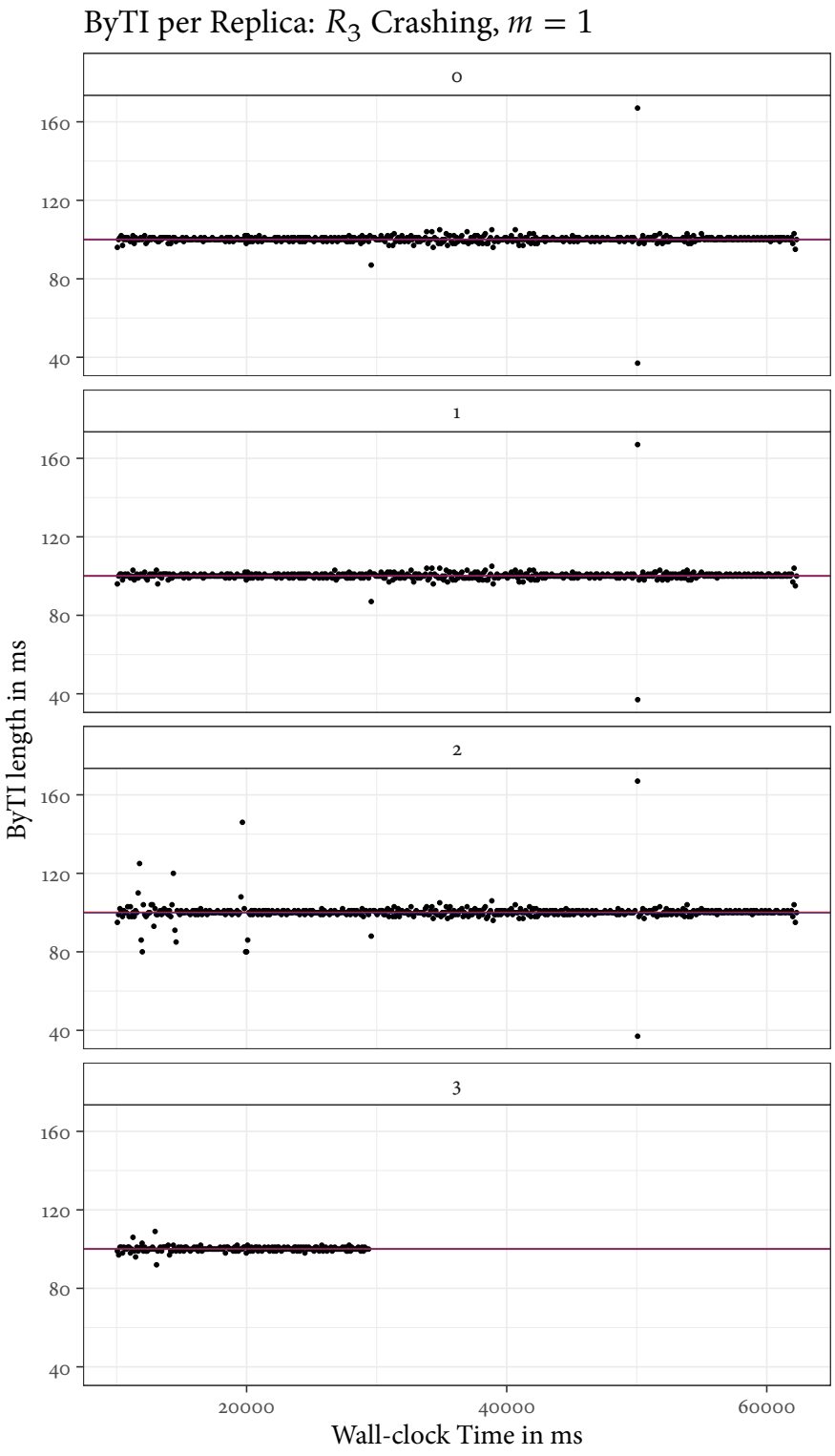

Figure 13.10. Scatter plot of all ByTIs across all replicas, with $m = 1$ and R_3 crashing after 28 seconds. Each dot represents a closed ByTI; the individual plots show each replica's view.

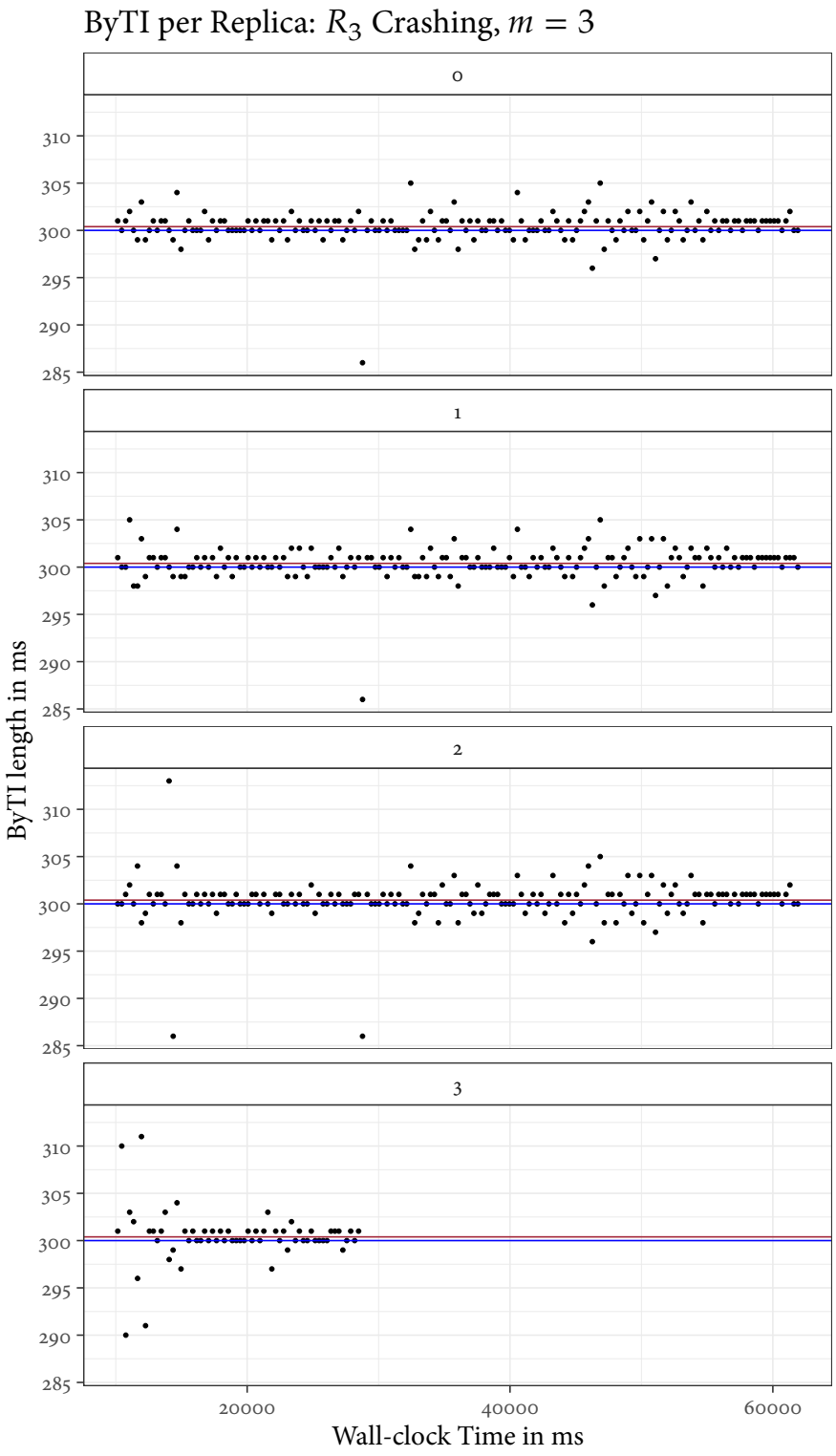

Figure 13.11. Scatter plot of all ByTIs across all replicas, with $m = 3$ and R_3 crashing after 28 seconds. Each dot represents a closed ByTI; the individual plots show each replica's view. The faintly visible red line is the mean of the actual, measured ByTI lengths, while the blue line shows the expected value $m * T = 300ms$.

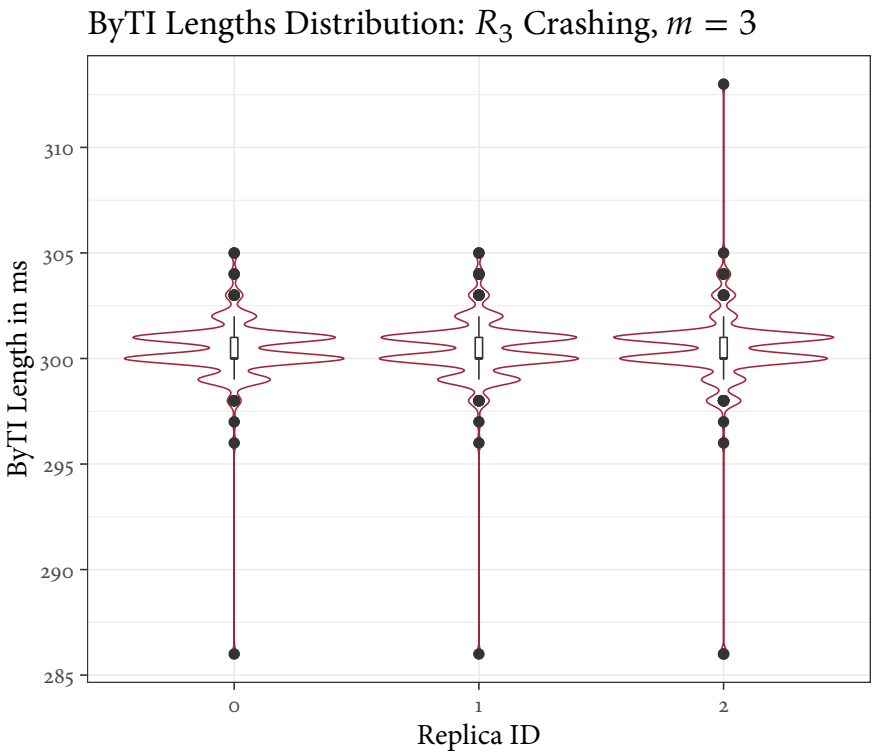

Figure 13.12. Combined Violin- & Box plots, showing distribution of ByTI lengths for $m = 3$ and R_3 crashing after 28 seconds.

13.4 SUMMARY

In this chapter, we defined, analyzed, implemented, and evaluated Byzantine Time Intervals, or ByTI for short, a mechanism for creating deterministic logical time intervals in replicated state machines. ByTI guarantees that each replica will see identical time intervals, and can therefore conduct deterministic measurements, e.g., by counting certain events per time interval to approximate current rates of this event. Note that these time intervals are not necessarily (or likely) equal to wall-clock time, although our analysis and evaluation has shown they are similar enough to enable sufficiently precise measurements for many purposes. ByTIs require replicas to send periodic clock tick messages that help to identify the interval borders, but do not need to carry any further information. No additional communication or message exchange is necessary between replicas, and no complicated decisions to handle malicious or crashed replicas are needed.

Unlike previous work, ByTI comes with a formal model, showing how close the ByTI length corresponds to the expected ideal length under different failure models. Message delivery and processing adds additional jitter that we have not modelled in detail, but our evaluation with a real system shows that jitter tends to smooth out when averaging intervals over time, even under high load.

Building Byzantine attackers which try to achieve maximum disruption—in our case extension and shortening attacks—was not attempted yet and has to be left to future work. However, we anticipate that the achievable effects in real systems are considerably smaller than those predicted by the mathematical models, because an attacker will have to take caution not to edge too close to the limits of tick intervals due to network and processing jitter, lest they risk failing with their attack.

Discussion

Finally, as the original purpose of this endeavor was the creation of a deterministic metric, in order to measure certain current system characteristic on the basis of which reconfiguration decisions could be made, we still have to choose at least one actual metric for our upcoming optimization efforts. Recall that our goal is the reconfiguration of certain UDS parameters in response to, e.g., the current load on the system, as hinted at in Section 8.1. For example, back in the UDS-SIM evaluations, we saw that the impact of RFDs[4] depends highly on the current load on the system, i.e., how many requests are arriving per time unit. In other words, we would like to measure request arrival rate, for which we now have a tool at the ready. By simply counting the number of requests that arrived within a ByTI and dividing by T, each replica will deterministically obtain the same approximated current request arrival rate, which in turn can be used to decide whether any actions against RFDs should be taken (e.g., adjusting the number of primary threads per round).

With reaching this milestone, we have fulfilled all the requirements for building systems that reconfigure themselves during runtime to reach certain optimization goals.

4 As a short reminder, since we have not used this term in a while: Round-filling Delays, i.e., a common problem of round-based schedulers, capable of affecting their performance under certain circumstances.

RULE-BASED AUTOMATIC SELF-OPTIMIZATION

14

Goal

In the last chapters, we have steadily built towards one primary goal, alongside some additional contributions: Enabling SMR systems with multithreading based on deterministic scheduling to self-optimize during runtime. To briefly recap the main idea in one sentence, we would like to utilize the dynamicity of our novel deterministic scheduling solution (Chapter 7) to react to sudden changes in the system's behavior, e.g., to change the number of primaries per round according to the current load on the system. The main milestones for this endeavor were, firstly, the design and creation of UDS, capable of reconfiguration during runtime, and secondly, an inventive Byzantine fault-tolerant mechanism for establishing a deterministic common time-basis among a set of replicated state machines.

Outline

This chapter will build on these two achievements to research, build, and evaluate a first version of our scheduling-based optimization approach, already capable of semi-autonomously improving performance of SMR systems. We also managed to successfully publish parts of this discourse in [4]. As with previous contributions, we have re-plotted the evaluation graphs of these peer-reviewed results for a more uniform presentation within this thesis, but they are still based on the original data gathered back then.

14.1 RELATED WORK

Before we dive into how we designed, implemented, and evaluated our approach, this section will briefly lay out and introduce additional related work which has not yet been presented in this thesis, but is closely related to our efforts. For example, we have so far omitted an entire class of approaches for parallelizing SMR—justifiable only by the didactic structure of this work—, which are currently also being hotly pursued by research groups invested in this problem of optimizing SMR.

Request-Level vs. Lock-Level

As discussed before, the current standard model for executing requests in replicated state machine is sequential—even nowadays—as is evident when looking at publically available frameworks for building such systems. A substantial amount of research on concurrent execution for SMR exists, aimed at parallelizing request execution and thus enabling the utilization of multiple processors. Previously, we have introduced related work focused on employing deterministic schedulers to enable this concurrency (cf., Section 7.3). Yet there is a second class of approaches which schedules requests not on a level of critical actions, but on a request-level, i.e., either parallelizing entire requests or executing them sequentially if they can not be safely processed concurrently. We call these two different classes of *request-level* and *lock-level*[1].

1 Technically speaking, according to our previously established terminology it should be called *Critical Action-level*, which somehow doesn't roll off the tongue quite as easily. We therefore accept this slight inaccuracy in favor of the other definition's pulchritude.

Solutions achieving concurrency on the basis of deterministic multithreading have already been presented in detail in Section 3.3.

14.1.1 *Request-level Scheduling Approaches*

Approach

Schneider [72] already suggested executing independent requests concurrently, whereas conflicting requests are executed under mutual exclusion. Two requests are considered conflicting if the order of their execution is relevant for achieving deterministic results, hence the granularity of parallelism is per-request. Since a system cannot predict potential conflicts from the invoked request alone, request-level approaches need a-priori knowledge about the behavior of request execution, e.g., information about causality dependencies explicitly provided by developers. For this classification into dependent and non-dependent requests, various concepts have been proposed.

The simplest one is the concept of *readonly* request, which never alter shared application state. CBASE introduces an additional parallelization phase between the usual agreement and execution phases for this purpose [56]. In this phase, a parallelizer takes the ordered requests and decides which can be executed in parallel, before workers finally execute them. For the parallelization decision the parallelizer depends on application-specific information that has to be provided to the platform by the application developers.

Late Scheduling Classification

Alchieri et al. [12] similarly introduce the notion of *classes of requests*, and demonstrate that separation of requests by type and allowing replicas to execute parts of the overall workload in parallel can yield significant performance improvements. Additionally, they present a reconfiguration scheme with which replicas can change the number of executing parallel threads on the fly. Later, these authors also gave a comprehensive overview on most request-level approaches in [13]. In this work, they distinguish *pipelined*, *late*, *early*, and *static* scheduling. Pipelining (used for example in the implementation of ZooKeeper [47]) allows multiple concurrent threads that each handle a part of each request, similar to pipelining in instruction execution in processors. Late scheduling deploys a deterministic scheduler in each replica that according to known dependencies deterministically assigns requests to executor threads (i.e., an implementation of Schneider's idea [72]). Early scheduling does the same but relies on a client-side classifier, which still requires a scheduler in the replica but reduces its overhead (e.g., [14]). Most recently, Batista et al. introduced their latest refinement of this early scheduling approach with several improvements over the 2018 version [20]. Finally, static scheduling only has a client-side classifier which selects one of a defined set of executor threads in each replica (e.g., *PSMR* [63] and its later optimization *opt-PSMR* [64]). The replica obeys this classification, and additionally the authors propose using entirely parallel request streams, each of them running their own consensus instance.

Fundamental Drawbacks

All of these solutions share the disadvantage that their classification depends on application-specific knowledge and must be done explicitly inside the replicated service or the clients, which adds additional complexity to the

development of such services, and to the potential adaptation of existing services looking to benefit from performant SMR deployment. Using this fact, a further classification of optimization approaches is possible, which distinguishes whether a solution requires prior knowledge about the application or not. Only the latter can be called *application-agnostic* approaches. Most request-level scheduling solutions do not fall into this category.

Speculative Approaches

As an approach which would not technically require prior knowledge, Eve follows a more speculative concept [54]. In Eve, a *mixer* component separates the incoming requests into batches whose contents are *unlikely* to interfere. These batches are then executed in parallel. However, instead of first agreeing on an order, Eve checks whether safety requirements have been violated after the actual execution. This requires a roll-back mechanism and falls back to sequential execution of the conflicting request if a violation occurs. While this technically sounds like Eve has no prior knowledge about an application, the mixer component can only do its magic if it is specifically implemented for an application, where it is a developer's task to impart knowledge about the application's potential request dependencies to the mixer implementation. We leave judgment of whether Eve truly is an application-agnostic approach to the reader.

Another central problem with request-level scheduling is that it is based on *potential* conflicts, while *actual* conflicts may not occur during runtime, e.g., due to conditional branching in the code. Even worse, some services may consist entirely of conflicting requests. As an example, imagine that all requests have to log their invocation into a log file or data structure, which is a shared resource that for some reason can not be concurrently accessed. In this case, all approaches presented above would have to entirely serialize their execution, while lock-level approaches can easily parallelize at least most parts of this service.

Orthogonality

It is important to note that some of these presented solutions are orthogonal to ours in the sense that combining UDS, an application-agnostic, lock-level–based deterministic scheduler, with prior knowledge about requests akin to *classes of requests*, could potentially allow it to reach even better scheduling decisions and to further increase its achieved concurrency.

14.1.2 *Other Approaches*

Storyboard as a Hybrid Approach

Storyboard [53] attempts to predict the order of lock acquisitions and releases for each request based on application information. Only requests with interfering locks need to be executed in a specific order, whereas other requests can be executed in parallel. These predictions are performed by an aptly named *predictor* component, which is implemented alongside the deployed service and supplied with information about its common locking patterns. Therefore, Storyboard represents an interesting combination when classified according to our established notions, making it a lock-level but not entirely application-agnostic approach, with similarities to Eve and its need for a carefully designed mixer component to achieve the best outcomes. In cases of mispredictions,

conflicts can be resolved by the replicas, which introduces additional communication overhead, but does not require any rollback mechanisms.

Parallel Consensus Instances

Finally, in COP [21], short for *consensus-oriented parallelization*, requests are organized as atomic jobs, so that one thread executes all operations inside the task pipeline instead of multiple different threads being responsible for the particular tasks of the replicated service. Every thread contains one (batch of) request(s), so that multiple requests/batches can be processed in parallel. As the requests need to be executed in a total order, the order is enforced before requests are forwarded to the service implementation.

14.1.3 *Null Requests Against RFDs*

Approach

In order to mitigate round-filling delays, Basile et al. suggested inserting additional null requests into the stream of requests to all replicas [19]. Null requests do not execute anything and terminate immediately, but are scheduled like any other request. Therefore, scheduling rounds are filled quickly, and RFD effects should vanish.

The rate of null requests gives an upper bound on how long a round-filling delay can become. In general, for a given number of primaries p and a desired maximum round-filling delay d, the frequency of null requests should be $(p-1)/d$. For example, with eight primaries and a null request every 10ms, a single application request thread needs to wait for at most 7 more threads to show up, which will happen after at most 70ms if no other ordinary requests arrive in the meantime. Or, solved for the required rate of requests, in case of 8 primaries and a maximum acceptable request latency of 5ms, we would need 1400 null requests per second to ensure that no request will ever get stuck in an otherwise empty scheduling round for too long.

Overhead

In our opinion, however, it is neither resource-efficient nor energy-saving to push this many requests through the group communication system (GCS) between replicas, simply for the sake of minimizing one particular problem of deterministic round-based schedulers. Even worse, null requests would only be necessary in low-load situations, whereas in high-load situations they would compete with actual requests and reduce concurrency due to UEBs.

To improve on this last point, replicas could inject null requests in an adaptive way: During high load, no null requests are created, whereas with low load, a reasonable number of null requests are issued as described above. This requires some means to measure the current load and an algorithm choosing the null request rate. Nevertheless, the system would still have to constantly complete numerous consensus instances per second especially during low load, just so that any potentially arriving application requests do not experience high RFD-induced latency before the system can adaptively pick up the pace. This would still consume unnecessary computing resources, even though adaptive null request insertion based on current load measurements would reduce this load significantly.

Indeterminism

We briefly investigated the possibility of deterministically inserting null requests locally in each replica, i.e., generating and inserting null requests

into the thread list directly within replicas, thereby avoiding the large network overhead of the previous suggestions. Unfortunately, such a solution fails because it would be impossible to know exactly how many null requests have to be added (and where exactly in Θ^2 we would add them) in order to finish rounds, since we have no reliable prior knowledge about how many critical actions threads will take, meaning they can unpredictably "spill over" into next rounds.

Therefore, we finally finish the discussion of related work and ideas at this point, and continue presenting our generalizable optimization approaches.

14.2 SYSTEM DESIGN

To optimize SMR systems during runtime, we require a set of components fulfilling different tasks for the greater whole, akin to our previously presented OptSCORE Architecture (Section 9.2.1),

- An SMR framework upon which we can base the implementations of our research ideas
- A flexible deterministic scheduling algorithm
- Deterministic metric(s) in each replica, on which to base reconfiguration decisions on
- Some kind of preferably smart entity capable of deciding new reconfigurations in reaction to changes in the monitored metric(s)

We will briefly explain each individual part of the system in the following paragraphs.

SMR Framework

BFT-SMaRt as a Basis

As with our previous work, our research prototype is based on BFT-SMaRt, which we augmented with implementations of the components mentioned above. Using BFT-SMaRt allowed us to extend a strong and stable framework instead of additionally having to develop a working, SMR system, which can be an epic journey in and of itself [36]. We left the BFT-SMaRt code base-forked from `https://github.com/bft-smart/library` in version v1.2 with its included group communication system untouched, and extend it by adding our own classes, implementing our components, which start their work after BFT-SMaRt's *DeliveryThread* delivers decided consensus instances containing ordered requests (*batches*).

In order to demonstrate that our approach enables high-performance Byzantine fault-tolerant applications, we performed all later evaluations with BFT-SMaRt in BFT mode. As BFT is the more demanding failure model, our results should be similarly applicable to systems covering CFT.

2 Using the notation introduced in Section 7.1

Parallel Request Handling

System Execution Model

To enable concurrent execution, each request delivered to the business logic of our application after being ordered by BFT-SMaRt is handled by its own thread. We added a layer which takes batches of decided requests from BFT-SMaRt, spawns a new request thread per individual request in the batch, and then hands off these threads to our implementation of UDS. Every thread executes the logic of exactly one request, after which a response is sent to the client and the thread conceptually terminates. In our implementation, we use thread pooling courtesy of the standard JDK Executor Framework to reuse the actual underlying system threads. According to our system model (cf. Section 3.4), the application logic has to ensure proper protection of all its shared data regions through the proper use of mutexes. It is not within the scope of our solution to analyze application code to enforce this or automatically translate non-conforming applications to our model. However, previous work exists which could automate this step, in theory [92]. Applications should also be designed and implemented in a way that conforms to well-known and tested standards for mutex-based multithreaded programming, and we assume that any application is correct in this regard, i.e., runs without race conditions, deadlocks and livelocks.

Reconfigurable Deterministic Scheduling

The background behind UDS has been extensively presented in this thesis. Since the configurable parameter space is so large, we tried to reduce this first basic self-optimizing system to the most effective parameter(s), and remembering Table 8.1, we had already collected the most common UDS parameters and their estimated effect on system performance. Therefore, our initial research with this prototype focused on modifying the primary count of UDS during runtime. As we will see, this already yielded encouraging results.

Deterministic Measurements

We base reconfigurations on the request arrival rate, as a simple yet rather effective estimator for the current load on the system. To deterministically measure this rate, each replica participates in our ByTI protocol as specified in Chapter 13. A replica first counts the number of requests arriving within a ByTI, after which it can deterministically calculate the current request arrival rate using the known (configured) length of a ByTI, e.g., $100ms$.

The resulting request arrival rate can be used as an estimation of how many clients are currently actively sending requests to the system, i.e., of the system's current load.

Staying Deterministic

Load rate estimates are calculated in an extra thread, which is deterministically inserted into the order of all threads. The automatic scaling component, deciding new configurations based on the value determined in the previous step, is executed next in the same thread. These computations are protected

by a dedicated lock to avoid race conditions. All of these measures ensure that each replica not only computes the same arrival rate value but also comes to the same reconfiguration decision for the same next UDS round.

Overhead Estimation

Lastly, a brief remark about the overhead our setup incurs: As we mentioned before in Section 13.3, with ByTI set to $T = 100ms$ and $N = 4$ replicas, we are looking at 40 additional request messages per second that have to be ordered and processed by the cluster. Compared to total throughput rates achieved by BFT-SMaRt based systems, this can be considered negligible. Ultimately, unlike solutions based on speculation, such as Storyboard [53], or systems based on following established schedules, like LSA [18], our additional ByTI ticks do not increase in number when throughput or lock frequency is rising, so that overhead always remains at a fixed, predictable, and low level. We consider this an advantage of our solution.

14.3 DETERMINISTIC SELF-OPTIMIZATION ALGORITHM

This section introduces our deterministic optimization algorithm, which decides reconfigurations of UDS primary thread counts during runtime, using the previously described arrival rate metric as its only input.

Fundamental Algorithm Idea

At its core, the reconfiguration problem revolves around ensuring that the number of primaries does not exceed the number of outstanding requests in low load scenarios (to avoid round-filling delays), whereas during load peaks primaries should be high (to better utilize multiprocessors). The following algorithm attempts to cast this rule into a mold in such a way that the resulting reconfigurations improve performance. We call this entire approach "rule-based self-optimization".

The following algorithm attempts to establish values which represent the last known best performance level supported by a UDS configuration with a certain number of primaries. We took to calling these values *borders*, since they are conceptually borders within the map of achieved performances between different configurations.

Algorithm Specification

Let p be the current number of primaries, p_{max} the highest configurable number of primaries, r the number of requests reported for the last ByTI, and $r_{max}[\,]$ an array of the highest observed r for each possible p. However, $r_{max}[\,]$ is not updated for the first r arriving after adapting p in order to avoid overshoot effects. Initially, $p := 1$ and $\forall i \in \{1..p_{max}\} : r_{max}[i] = 0$.

The algorithm maintains a border value b on which the placing of several virtual *borders* is based, defining which p is appropriate for the currently estimated load: If $r \in \left[\frac{i}{2}b, \frac{i+1}{2}b\right)$ then $p := i + 1$.

Simply put, if $r \in [0, b)$ then $p := 1$, if $r \in \left[b, \frac{3}{2}b\right)$ then $p := 2$, and so on. The scaling decisions are now simple: p will be adapted to $p := min\{max\{1, \lceil\frac{2r}{b}\rceil\}, p_{max}\}$. Note that these rule-based decisions rely on the assumption that throughput scales linearly with the number of primaries.

As b is initially unknown and may vary depending on the executed requests it is estimated and dynamically updated during runtime. First, b is initialized to a value higher than any expected r, so that the decision formula will always

yield $p = 1$ in the beginning. Then, the algorithm tries to adjust b in a way that scaling of p is optimal for the current application load:

If during a defined period of time t_m (measured by counting the number of reported measurement periods),

- p stayed constant AND
- $p < p_{max}$ AND
- for each reported r the condition $r > h \cdot r_{max}[p]$ holds for a threshold value $h < 1$ we introduce for this purpose, AND
- $r_{max}[p]$ did not change,

then we assume we run on p's maximum capacity and should try the next higher p to see whether we can improve performance. This is achieved by adapting b to $b := h \cdot r_{max}[p]\frac{2}{p}$.

In other words, we set b so that $h \cdot r_{max}[p]$ is the new virtual scaling border between the current p and $p + 1$. Then, p is reconfigured according to the formula for scaling p, as presented above.

A problem now exists in the fact that b can currently only ever be decreased by this algorithm. Therefore, we also introduce an "aging factor": If, after a number of measurement periods t_a, no reconfiguration was decided, we age b and reset all maximum-counters: $b := \frac{1}{h}b$ and $\forall i \in \{1..p_{max}\} : r_{max}[i] = h \cdot r_{max}[i]$. This ensures that the algorithm does not end up with a permanently low b, but instead is constantly forced to adapt to the current situation.

Finally, to give a rough idea of the magnitudes of these values: After some initial experimentation, we chose the values $h := 0.9$, $t_m := 7$, and $t_a := 15$ for the evaluations in Section 14.4.

This concludes the introduction of all the required parts for a simple self-optimizing SMR setup, and segues into the presentation of our evaluation results obtained with the prototype implementation of this system.

14.4 EVALUATION AND DISCUSSION

14.4.1 *System Setup*

To evaluate our reconfiguration-based approach, we set up the same 4 machines used for the previous measurements in the context of ByTI as replicas, each running an Intel i7-7700 with 8 logical processors at stock speed (3.6 GHz), and 32 GB RAM at standard JEDEC DDR4-2400 timings. Additionally, our trusty 4 PCs with Intel Xeon E3-1220 with 4 logical cores at stock speeds of 3.1 GHz and 16 GB of DDR4-2400 RAM were again used to spawn clients.

Evaluation Setup

An additional machine identical to the 4 replicas was used to coordinate test case runs, e.g., to start replicas, set up directories, start and instruct client machines, and to gather logged results for each test run. On all of these machines we installed Ubuntu 18.04 and Java 11.0.3 to run our customized SMR framework. All machines were connected via one Gigabit switch. The 4 replicas were

additionally connected to their own Gigabit switch on secondary network interfaces for intra-cluster communication.

In all evaluation runs, BFT-SMaRt was configured with BFT mode enabled, `batchtimeout` set to -1 to force fast batch decisions, `maxbatchsize` set to 400, and both `in-` and `outQueueSize` set to 500000. State transfer and disk logging was disabled so as not to interfere with benchmarking. All other BFT-SMaRt configuration parameters were kept at the defaults found in its public v1.2 release.

14.4.2 *Evaluation Requests*

Ensuring Comparability

A demonstration of our approach in comparison to other solutions is difficult in two regards: (i) A first advantage—easier development of applications due to the lack of required pre-classification of requests (cf., Section 14.1.1)—is not easily quantifiable and measurable, and (ii) most publications improving the performance of SMR applications use custom workloads and code to specifically demonstrate their claims. We decided to use a similar approach, where all variables can be controlled. At the same time however, we wanted to keep our evaluation highly generalizable, i.e., close to what real world applications would behave like.

Flexible Approach

Therefore, we designed three different evaluation request types (shortly explained in more detail), each emulating their own application profile. Each request type executes a fixed series of commands in order, where possible commands are:

- **L**ock$_i$: Execute lock of a specific mutex m_i
- **U**nlock$_i$: Execute unlock of a specific mutex m_i
- **C**alculate$_x$: Simulate the execution of business logic by fully loading a CPU core for a duration x in microseconds

Notation

Concatenating abbreviations of these commands yields a simple notation of the execution a replica performs for each request type. For example, $L_0C_{250}U_0L_1U_1$ means a replica locks the mutex m_0, simulates load for 0.25ms, unlocks the same mutex, quickly locks and unlocks mutex m_1, and then terminates the execution of this request. Responses do not contain any data.

Synchronous Clients

Request arriving in the system originate from multiple synchronous clients, i.e., each client sends a request to our SMR cluster, waits for a response, and only then sends the next request. We chose this approach for our evaluations because it promotes the effect of RFDs, without compromising the generality of our contributions and their underlying concepts. In other words, asynchronous clients would of course be supported just as well as synchronous clients, and would just as clearly show effects of RFDs if clients are sending requests with low rates, but showing these effects would be more convoluted than simply activating or deactivating certain clients.

Request types

We used the following three request types to (i) imitate common real-world application behavior and (ii) specifically illustrate both the benefits and drawbacks of our approach. All three request types intend to show specific execution patterns replicated applications such as sophisticated key-value (KV) stores could be expected to perform:

$C_{250}L_0C_{50}U_0L_1U_1$: Calculate for 0.25ms, lock mutex m_0 and calculate for 0.05ms, unlock mutex m_0 and do a bookkeeping operation, e.g., incrementing a global request counter protected by mutex m_1. We assume the latter costs virtually no time and has to be done by all three request types. This request imitates a short, fully parallelizable precalculation followed by an update to a shared data structure, e.g., a cryptographic signature check before modification of a value in a KV-store.

$L_0C_{250}U_0L_1U_1$: Lock mutex 0, calculate for 0.25ms, unlock mutex m_0 and do bookkeeping. This simulates a highly congested mutex in a heavily used central data structure, e.g., the insertion of a new key-value pair into a tree-based KV-store, which triggers a costly rebalancing operation. This type of request is a demonstration of executions that can not be properly parallelized.

$L_1U_1C_{500}L_1U_1$: Lock and unlock the bookkeeping mutex m_1, calculate for 0.5ms concurrently, then do bookkeeping. This imitates a computation which needs short access to shared data (in this case the bookkeeping variable), to then perform a parallelizable longer calculation such as the computation of statistics on the current KV-store status. With its considerably longer calculation, this also has the potential to demonstrate the *Unbalanced Execution Time*-problem within scheduling rounds (cf., Section 6.2),

14.4.3 *Evaluation Workload*

Methodology

In our following evaluation methodology, a *workload* is a set of instructions for our client machines, specifying exactly how many client instances to create and when to *activate* or *deactivate* which client instance. An *active* client instance sends one of the presented evaluation requests synchronously, i.e. sends a new request whenever the answer to the previous one was received. A *deactivated* client stays connected to the replica cluster, but does not send any requests, enabling fine-grained control over how many clients are active during certain times of a test run, and which kinds of clients we instantiate, i.e. the type of requests they send.

2-Peak Workload

We designed a dedicated workload—shown in Figure 14.1—to simulate real-world usage of a service, during which it may experience short usage spikes as well as phases of low load and intermittent phases of average usage. The workload runs for 300 seconds in total, first fluctuating between 1-9 clients in

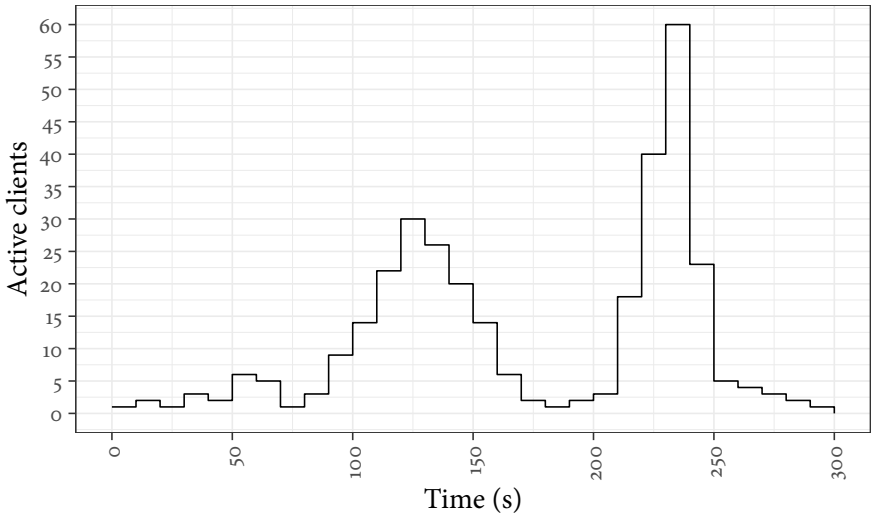

Figure 14.1. Two-peak workload with low- and high-load phases.

a low load phase, followed by a small spike with up to 30 clients putting average stress on the system. After a short lull period with 1 client, a fast ramp up to 60 clients simulates a load spike[3]. The end of the workload sees usage slowly subsiding again. For each client we can choose the type of requests being sent.

Using this workload, we can test how well a system responds to sudden throughput changes and how fast it can adapt itself to new conditions.

14.4.4 *UDS Reconfiguration Effects*

Confirming the Hypothesis

In Section Section 8.1, we argued that different deterministic scheduler configurations should significantly affect performance. To confirm this in a real system, as opposed to simulations based on a severely limited abstraction of one, we ran several fixed configurations of UDS with our workload, measured performance in terms of throughput and request latency, and compared the results against a baseline with single-threaded execution.

Figure 14.2 shows one of these tests, with all clients sending $L_1 U_1 C_{500} L_1 U_1$ requests, following the load profile defined by the workload. We measured throughput for a range of primary configurations, with 2 fixed critical operations per primary per scheduling round. The plot shows the overall throughput observed in one replica plotted against the elapsed real time of the evaluation workload. Configurations with more than 7 primaries were tested, but the results of these measurements were similar to the ones with 7 primaries. Therefore, we omitted them from the following discussion to maintain readability in the plots.

We can see several effects in Figure 14.2:

3 Note that due to the relatively long computation periods of the previously presented request types, these few clients suffice to load the cores of the replicas to demonstrate load spikes and resulting scheduling effects.

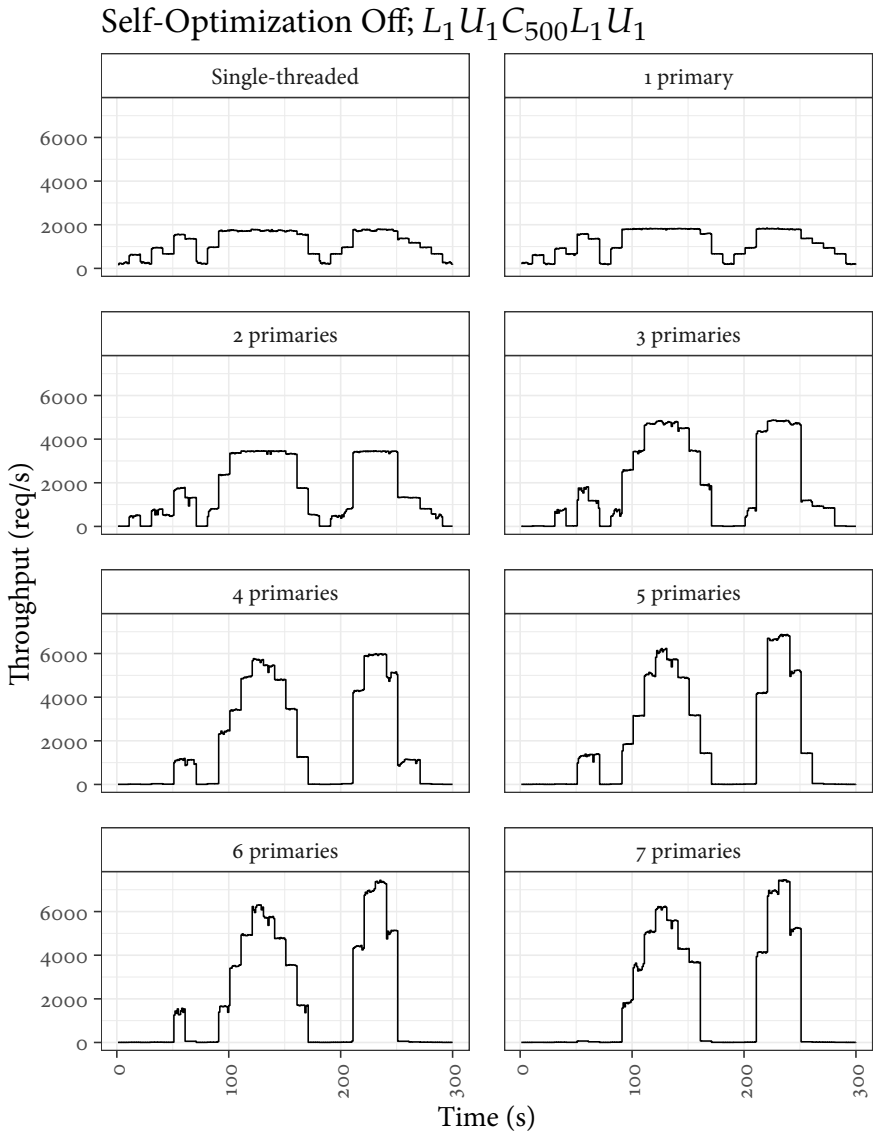

Figure 14.2. Influence of UDS on performance for $L_1U_1C_{500}L_1U_1$ requests compared to single-threaded execution. Automatic scaling was not activated, i.e., UDS was set to a fixed number of primaries (from 1 through 7, always with 2 steps/round) for each run.

- UDS with 1 primary is comparable to single-threading to the point where these setups are more or less indistinguishable.
- Since $L_1U_1C_{500}L_1U_1$ can run highly parallelized, the more primaries are configured the higher the throughput (see peaks).
- Due to RFDs, in low-load phases of the workload, throughput drastically declines with more primaries.

In our setting, ByTI are still sent every $100ms$, creating threads performing the ByTI logic, which are also scheduled by UDS. Therefore, the system never stalls forever, but still experiences extremely high latencies during low load for configurations with many primaries.

These results clearly demonstrate the validity of our claimed need for an adaptive scaling solution, which in this case would be based on scaling primaries.

14.4.5 *Simple Automatic Reconfiguration*

To this end, we implemented the simple automatic reconfiguration algorithm introduced in Section 14.3, which scales primaries depending on the currently estimated request arrival rate.

The two main purposes of the algorithm are

- to detect sudden drops in the arrival rate, which indicates few outstanding requests and a possibly impending stall due to RFDs. UDS should then immediately be reconfigured to fewer primaries in order to avoid this, and
- to detect rising arrival rates, which indicates an increase in system load. Here, UDS can be configured with more primaries the more requests are detected, in order to improve multicore utilization.

How to Read the Plots

The following subsections show the results of evaluations for all three request types and a workload with a combination of all of those requests, comparing static configurations of UDS against our automatic scaling algorithm. For improved readability, we only show the extremes of static configurations with 1 (blue dotted line) and 7 primaries (gray dashed line) in comparison to our automatic approach (orange solid line). Further, the graphs also show the number of primaries decided by our algorithm over time as a light-gray dashed and dotted line, referencing the axis on the right side of the figures.

Evaluation: $L_1U_1C_{500}L_1U_1$

Figure 14.3 shows the results for our evaluation-workload with requests of type $L_1U_1C_{500}L_1U_1$ (the same as used for Figure 14.2). Comparing the performance of different solutions in the displayed plots, showing throughput vs. time, should be relatively intuitive. Still, consider that conceptually, the area under each colored curve, i.e., its integral, represents the number of requests

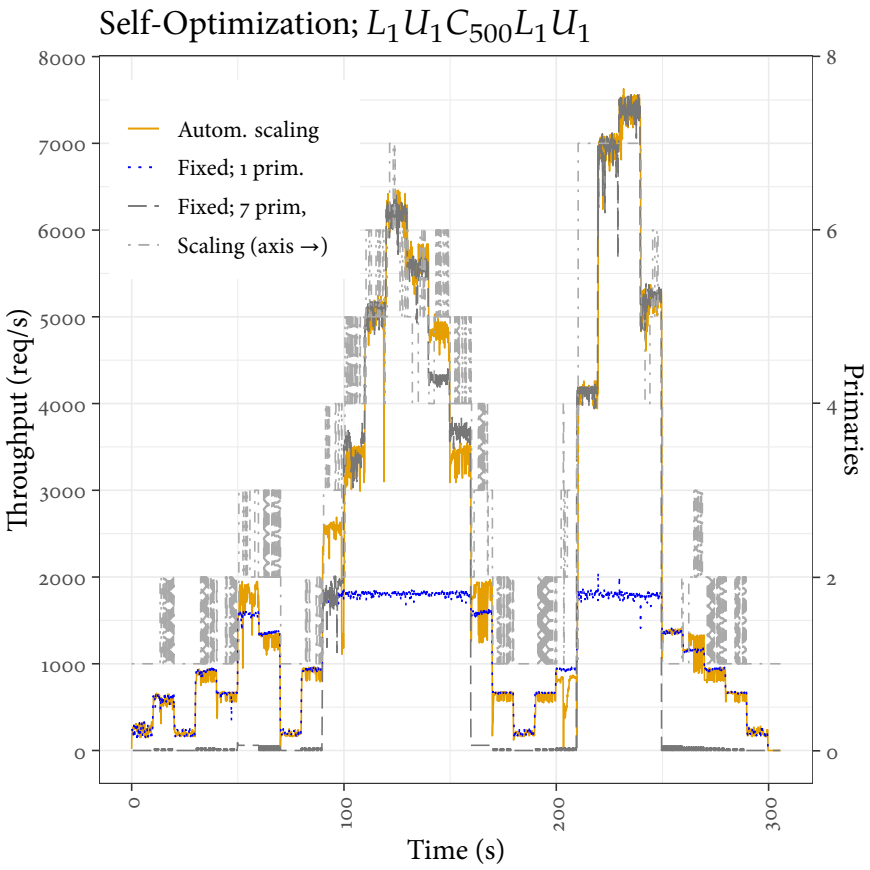

Figure 14.3. Automatic scaling compared to fixed configurations for $L_1U_1C_{500}L_1U_1$.

that were successfully finished in a given time period. Therefore, it can be roughly said that the larger this overall area, the overall better the approach, simplifying intuitive comparisons between the measured approaches. Automatic scaling performs better than fixed configurations in most cases, on high load keeping up with the previously best 7-primary configuration. During low-load phases, automatic scaling uses low primary numbers and achieves similar performance compared to a static 1-primary configuration, whereas the static 7-primary configuration stalls.

Scaling Algorithm Behavior

Our proof-of-concept scaling algorithm continuously tries to find a reasonable configuration without prior knowledge about the application. This can lead to oscillations between configurations, as seen e.g. between seconds 150-160. Most of the time, however, the algorithm still outperforms the static configuration, while seamlessly scaling up to higher loads even during fast spikes such as in second 210 or 220 of the workload. Overall, the scaling algorithm significantly outperforms any static configuration averaged over the entire workload for this particular request type.

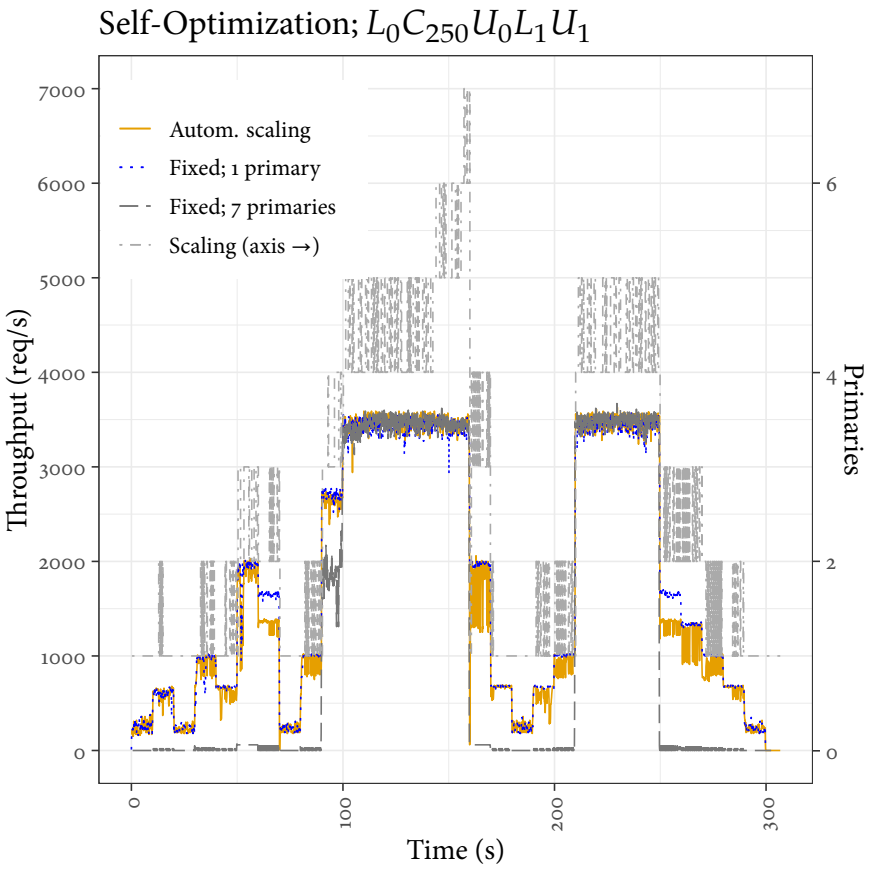

Figure 14.4. Automatic scaling compared to fixed scheduler configurations for $L_0C_{250}U_0L_1U_1$.

Evaluation: $L_0C_{250}U_0L_1U_1$

Figure 14.4 shows the results when all clients want to execute $L_0C_{250}U_0L_1U_1$, which has a single highly-contended lock and thereby effectively bottlenecks any parallelization efforts. Generally, the performance is good in most phases, closely trailing and during certain loads surpassing the 7-primary configuration, while only occasionally experiencing performance drops due to oscillations or suboptimal scaling decisions. This demonstrates the viability of our solution as a drop-in replacement for static schedulers, even when prior knowledge about the application would normally dictate the usage of a fixed configuration. Of course a further benefit of choosing a dynamic solution would be additional upgradeability and flexibility, should other execution patterns appear with later updates of the replicated application.

Evaluation: $C_{250}L_0C_{50}U_0L_1U_1$

Finishing the evaluations of each individual request type, Figure 14.5 presents the results for $C_{250}L_0C_{50}U_0L_1U_1$. This stresses our algorithm again, as the first 250µs of computation is fully parallelized by UDS independent of its configuration. Automatic scaling displays only minor overhead compared

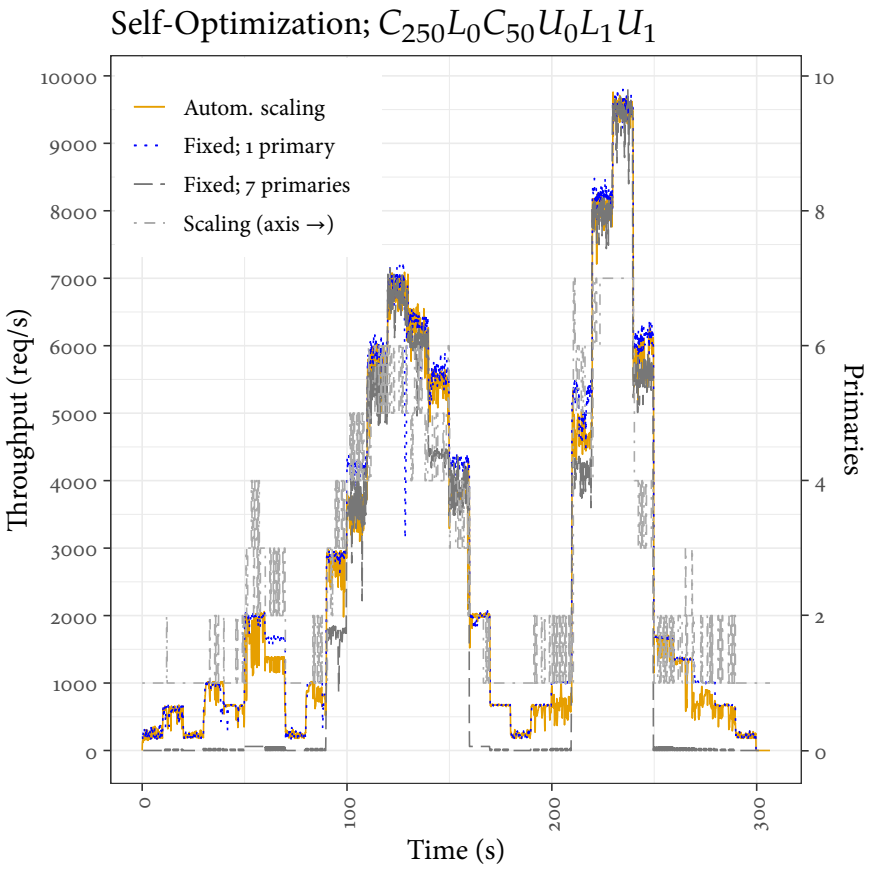

Figure 14.5. Automatic scaling compared to fixed scheduler configurations for $C_{250}L_0C_{50}U_0L_1U_1$.

to static configurations during all phases of the workload. These results thus show once again that our solution can be a viable replacement for a fixed configuration scheduler, gaining flexibility without significant drawbacks.

Evaluation: Mixed Requests

Mitigating RFDs vs UEBs

Finally, as discussed in, e.g., Section 6.2, additional scheduling problems such as UEBs can occur when requests of different types are present in the same application. However, for mitigating UEBs, we would expect configuration parameters other than the number of primaries (e.g., steps or modified total orders) to be better choices. Even so, we still observe the apparent effectiveness of our solution for a workload using a mix of all three presented request types in C14.6, although further performance increases with algorithms capable of adjusting different UDS parameters will still have to be tested in future work.

Once again, during the biggest load peaks, our algorithm manages to surpass the safe 1-primary configuration by over 80%, and during low-load phases it can easily maintain its lead over a 7-primary configuration.

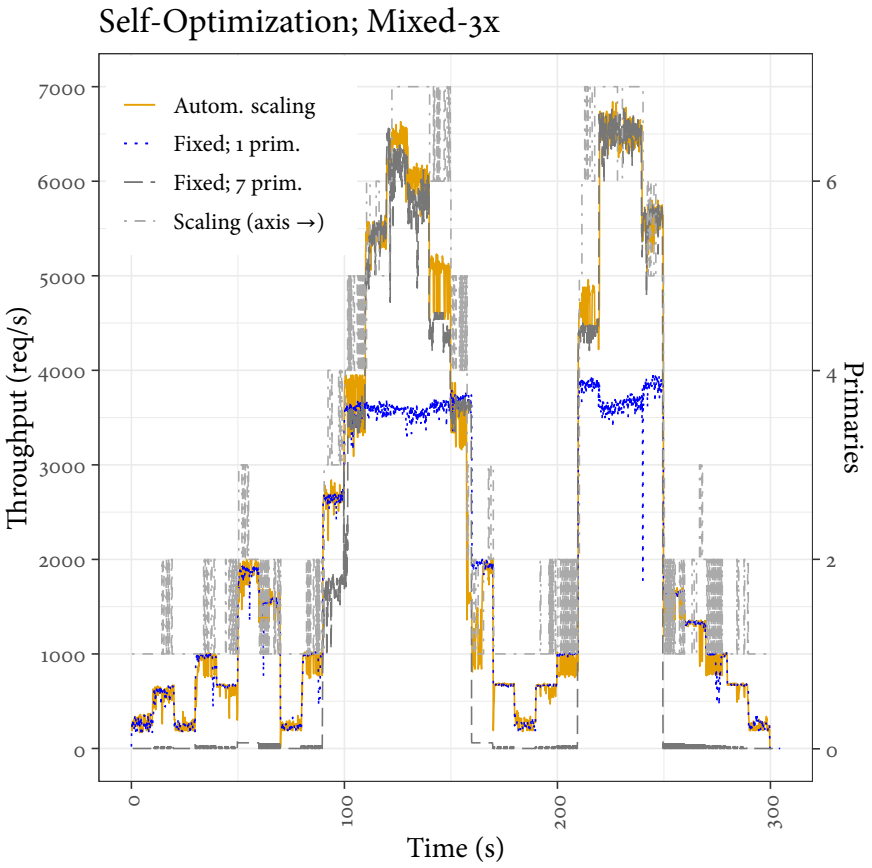

Figure 14.6. Automatic scaling compared to fixed scheduler configurations for mixed clients sending a mix of all 3 request types.

14.5 SUMMARY

In this chapter, we have presented our rule-based, i.e., based on inferred rules about the behavior of UDS, approach to optimizing BFT SMR systems with deterministic multithreading. After presenting several pages of related work in addition to the ones presented in previous chapters of the thesis, we introduced our system design and the prototype algorithm created for demonstrating our approach. The evaluations clearly validate our research efforts and show that this idea can work rather well for a variety of different applications.

Discussion

It is important to note, however, that the simple scaling algorithm introduced in this chapter is only meant as a proof-of-concept for demonstrating the effectiveness of our general automatic scaling and reconfiguration approach, and not meant to be a final solution in this regard. In fact, the algorithm is still rather rudimentary, of course: It scales only primaries without modifying e.g. the number or order of critical operations per round or any other parameters. Further, its decisions are solely based on the periodically estimated request arrival rate. It also does not use any prior knowledge about the application and can not learn patterns from repeated observations of certain request types. Yet, even this simple algorithm achieves roughly the same

performance of previous static schedulers in the worst cases, and significantly outperforms them in the best cases.

This contribution to the state of the art of optimizing SMR systems with deterministic multithreading warrants further research into more intelligent optimization algorithms. With our last few chapters coming up, we will shortly present our final contribution to the field in this regard, where we attempted to employ more advanced methods to reconfigure the system.

15

Realizing the Potential

Our prototype implementation of a self-optimizing system based on a simple algorithm, as presented in the previous chapter, is certainly a valid demonstration of the effectiveness of our general approach, but is unfortunately also limited in its scope by way of only focusing on a single configuration parameter and a single metric on which it bases its decisions. It would be easy to find applications as counter-examples where this rule-based approach would quickly reach its limits. For example, imagine critical infrastructure in a resource-constrained environment, which would like to utilize SMR as a technique for guaranteeing availability and reliability, such as, e.g., IoT edge nodes controlling smart traffic guidance solutions in a smart-city context[1]. In such a setting, the goal might not be maximum performance, i.e., higher throughput, but maybe rather energy efficiency, predictable latencies or thread prioritization for critical requests.

These goals would be difficult to reach with only adjusting single parameters, like the number of primaries per round. Quite to the contrary, they could require multiple and more involved adjustments, e.g., intelligent scheduling orders within rounds, giving each thread the exact number of steps it needs, or the switching between different implementations of the `prim()`-predicate to prioritize certain threads or to minimize UEBs. True self-optimization towards arbitrary goals and with multiple parameters in the mix would therefore require different strategies.

We previously introduced and partially motivated the approach of training an intelligent agent with the help of Reinforcement Learning, seeing as this was our chosen approach for trying to achieve more complex self-optimization. This chapter will detail additional background knowledge regarding RL, introduce our chosen approach for implementing a specific RL strategy known as Deep Q-Learning, present the problems we faced during this phase of our research, and finally show the limited and preliminary results alongside a discussion about their significance and remaining future work we identified.

15.1 REINFORCEMENT LEARNING BACKGROUND

Partially introduced in Section 12.4, the very basics of Reinforcement Learning should already be in our minds. Still, to follow the research we conducted using this technique, further explanations are in order. Seeing as ML, and lately particularly RL, have enjoyed their status as hot topics of global computer science research for decades now, the amount of publications and related work available in these fields is staggering, as are the depths one can dive to on any particular topic within these research fields. The main goal of our research was not to contribute to the state-of-the-art in RL, but to utilize it as a tool for

1 Similar scenarios are actively being researched, for example in the EU research project SORRIR, which we also contributed to during our PhD (cf., [2], [7], [1])

reaching our own research goals aimed at optimizing deterministically scheduled SMR systems. Therefore, we will try to keep the amount of background information on these topics to the required minimum and only provide the necessary details for understanding our motivations and efforts.

To gain and overview of our implemented solution, let us start by taking a closer look at the differences between available RL methods, and why our choice finally fell on DQL.

15.1.1 *Terminology and Basics*

Section 12.4 briefly introduced the terms *environment*, *agent*, *action*, *policy* and *step*. The differences between existing RL approaches can only be appreciated after the introduction of a few additional terms.

ML vs. RL Models

Firstly, the confusingly termed *model* can have two meanings in the context of ML and RL. For supervised and unsupervised ML methods, *model* describes the resulting algorithm or neural network the training phase yields, i.e., the mathematical model the ML method has produced after it has been fed with all of the training data. This model can then be used to perform the work it has been trained for by feeding it with new, actual data from the world. Current examples for this would be the recognition of faces to unlock smartphones or detection of skin cancer in patients ([85], [43]).

Model-Free vs. Model-Based

However, in the context of RL, a *model* ties back to the definition of MDPs as introduced before in Section 12.4. Here, the model denotes the information a fully specified MDP provides about future expected rewards and states. RL methods can then be classified as either *model-based* or *model-free*, depending on whether they use this information or not. For example, if an environment, such as e.g., a simple board game, has a well-defined set of states S in addition to known probabilities $P_a(s, s')$ and rewards $R_a(s, s')$ for all possible transitions, then an agent can for a current s always ask the environment for predictions about possible next states s' and associated rewards. An RL algorithm utilizing these predictions, i.e., this model of the environment, is *model-based*.

Conversely, RL methods that learn from interacting with the environment without relying on its predictions is called *model-free*. These methods are well suited for many real-world control problems, in which predicting information about future states is difficult or even impossible. One example of such a complex live system which is not fully modeled mathematically is the subject of this thesis.

On- vs. Off-Policy

Secondly, a further categorization of RL methods algorithms depends on which strategy the agent follows to choose its actions during training. Consider again that an agent is trying to maximize rewards by finding a policy through interaction with an environment. Therefore, conceptually, it has to have two policies: One for choosing actions during training, and the actual target policy which is meant to be learned. RL algorithms which use one and the same policy for both of these tasks are called *on-policy* algorithms, whereas techniques that use a separate policy for exploration during training are known as *off-policy* [77]. As for policies themselves, the policy which always chooses

the best action (according to current knowledge), i.e., the action yielding maximal future accumulated reward, is called the *greedy policy*. The importance of these distinctions becomes apparent when imagining an agent which has not yet extensively explored its environment, i.e., which does not know about many of the possible state transitions. If the algorithm learns on-policy, it is possible for the agent to get stuck in a local minimum, where it always picks known actions to greedily maximize reward, without further exploring the environment even though there could be greater rewards hidden in as yet unknown states.

Terminology

Furthermore, when training RL agents, there are some common terms we will be using for describing how the training is performed. Among these, the first is a training *episode*, which is a collection of steps the agent takes in the environment, until either a maximum configured number of steps has been reached or a terminal state is encountered. For many problems, such specific terminal states—e.g., the final winning move in a board game—exist and can therefore be reached by an agent during training. Whenever a terminal state is reached, the environment is *reset* back to an initial state, and the agent can again start interacting with a fresh environment. Environments are also usually reset after a fixed maximum number of steps to encourage re-exploration of initial conditions and avoid infinite interaction in environments that allow loops. Performing an entire *training run* means letting an agent run for many episodes, during which it gathers experience about the environment and in the best case learns to interact with it in an optimal way.

Hyperparameters

Lastly, while training an agent, values for all of these parameters (and depending on the specific RL method used, many additional ones) have to be chosen for each training run, and can have great impact on the efficacy of the training. The parameters used to adjust the training procedure of an agent are called *hyperparameters*, and the process of finding optimal hyperparameters for best training results is commonly known as *hyperparameter tuning* or also as *hyperparameter optimization*, and Section 15.4.1 will discuss our process of finding good hyperparameters for our problem.

15.2 DEEP Q-LEARNING

Recall the basic definitions of agents from Chapter 12, which says that agents follow a policy π to accumulate rewards, and the goal of an RL algorithm is to find a policy maximizing these accumulated rewards over time.

A formalized representation of this notion is commonly given as the maximization of the state-value function

$$\max_{\pi} V_{\pi}(s) \quad \text{where} \quad V_{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right]$$

Value-Function

In this notation, $V_{\pi}(s)$ is the function calculating the expected return, i.e., value, when starting from a state s_0 and picking actions following policy π. γ is the *discount factor* which discounts future rewards r, i.e., adjusts how far

into the future the agent should plan. With a high gamma (e.g., 0.99), the agent might prioritize picking actions which give lower rewards now, because of the prospect of setting up the environment to give larger rewards in the future. In other words, this function represents how desirable a given state is when following a learned policy from this state.

Bellman Equation

Given a policy, future expected rewards can be defined recursively, which yields the following equation named after Richard Bellman [87]:

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V_\pi(s')$$

where $R(s, \pi(s))$ is the mean of the probabilistic reward the agent receives when transitioning from s to s' thanks to action a, which is chosen by $\pi(s)$. $P(s' \mid s, \pi(s))$ is the probability of transitioning to s' under π, as defined by the MDP (cf., Chapter 12).

15.2.1 *Q-Learning*

State-Action Value Function

Since so far we have not mapped this abstract meaning of policies to concrete actions our agent could take, it is useful to define the reward that is expected when taking an action a from a state s and then following policy π, which is called the state-action value function:

$$Q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$$

Q-Values

We can again recursively define these so called *Q-values* ([80], [76]):

$$Q_\pi(s, a) = R(s, a) + \gamma \sum_{a'} \pi(a' \mid s') Q_\pi(s', a')$$

In other words, Q-values signify the expected, γ-discounted reward for choosing a in s and then following π.

A well-known and successful RL-technique is *Q-Learning*, which estimates Q-values by starting from an initial, randomized guess and then iteratively updates its Q-values during each step i it takes in the environment using the following update rule:

$$Q_i(s', a') = (1 - \alpha) Q_{i-1}(s, a) + \alpha r_i + \alpha\gamma(\max_b Q_i(s'', b))$$

Update Rule for Q-Values

Here, α is a learning factor, commonly called *learning rate*, which governs how rapidly Q-values changes with each step, and $\alpha\gamma(\max_b Q_i(s'', b))$ is the weighted and discounted expected maximum reward from the next state s'' for possible actions b. By exploration of the environment, an agent can update

and thereby learn these Q-values directly. In [80], the authors describe that Q-values are assumed to be saved using a look-up table which associates Q-values with all possible combinations of states $s \in S$ and actions $a \in A$. This finally gives an agent a concrete tool for determining an optimal policy, because the optimal policy from a given state s is always the one maximizing future Q-values—which are known after thorough exploration and can be looked up in the table. From our previous definitions, it should be clear that Q-Learning is a model-free method, because it does not depend on any predictions from the environment to learn Q-values and arrive at an optimal policy.

15.2.2 *Deep Neural Networks for Function Approximation*

Exploding Q-Tables

However, for sufficiently complex environments with large or even infinite state and action spaces, the look-up table would become prohibitively large or would require infinite storage. To apply Q-Learning to problems of this sort, we would need a way to approximate the function $Q(s,a)$ mapping state-action pairs to Q-values. Approximating this function can nowadays be achieved by employing Deep neural networks (DNN), which are neural networks composed of multiple layers of neurons [44]. Figure 15.1 shows this training loop, which we already saw in its basic form in Section 12.4, but this time with a neural net inside the agent trying to approximate $Q(s,a)$.

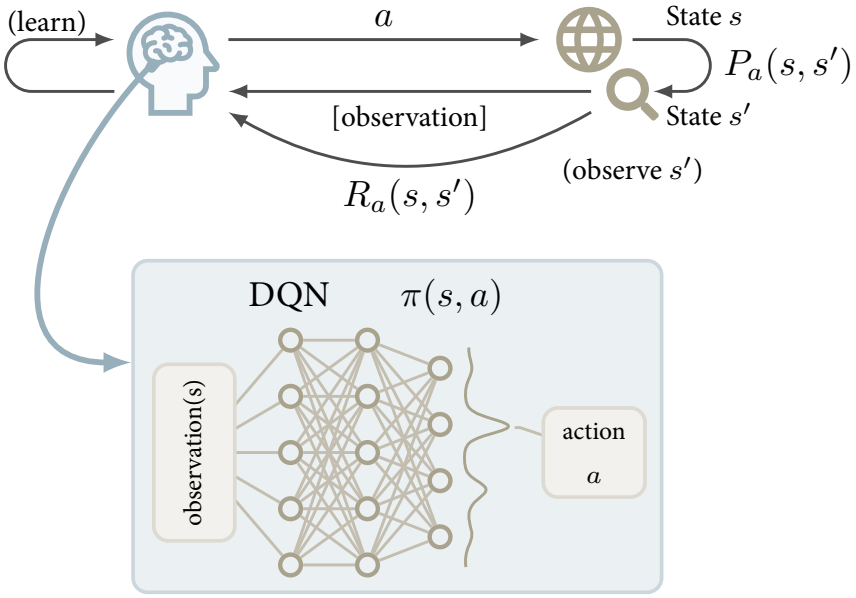

Figure 15.1. The training loop in a Q-Learning setup where the $Q(s,a)$ is approximated by a deep neural network. The learning step updates network weights, i.e., the activation levels for each individual neuron in the network, based on how far off its predicted Q-value was from the actually received reward from the environment.

The exact theory behind employing DNN for Q-Value estimation shall be outside the scope of this chapter and thesis; instead we will only briefly introduce the most important terms here. For an overview on DNN in general, we recommend [62], and readers interested in more details are directed to [66],

which introduced Deep Q-Networks (DQN) as a method for building neural networks capable of approximating Q-values.

DQN Novelties

DQN introduced mainly two new techniques that alleviate some of the problems earlier, similar, approaches experienced. We briefly mention these so as to introduce the terms we will later see in our discussions about hyperparameters. The first new technique, experience replay, aims at smoothing out the learning process over the entire state space, as opposed to localized updates due to an agent currently exploring a subspace of the state space. To illustrate what this means, imagine for example an agent navigating through a 2D game world, and the current action maximizing future rewards is to move north-east. Then, all collected observations will be from the north-eastern part of the game world, biasing the training results. The solution by Mnih et al. was to introduce a replay buffer into which state transitions are cached, and during the training, i.e., the updating of neural network weights to approximate the Q-function, random mini-batches of previous experiences are taken from the replay buffer and used to apply Q-learning updates. This is also supposed to mimic how biological neural networks (i.e., brains) also learn from old experiences even while making new ones.

The second technique is the utilization of two distinct DQ-networks Q and $\hat{Q}$, also called *target* network, which is periodically cloned from Q to improve the stability of the learning algorithm. Further details on this can be found in [66].

Additional DQN Hyperparameters

Important for this discussion is that these two techniques introduce additional hyperparameters with which the training of agents can be influenced. For one, the size of the replay buffer, as well as the size of the mini-batches drawn from it in each training step, can be configured. Additionally, the parameter $period_\tau$ governs how often $\hat{Q}$ is cloned from Q. Later refinements of this latter technique also introduced τ to signify how the weights of Q are copied to $\hat{Q}$, where $\tau = 1$ signifies direct cloning of Q's values, and $0 < \tau < 1$ describes more gradual soft updates of the weights every $period_\tau$ steps.

Lastly, to emphasize exploration especially in the beginning of a training run, the actions yielded by the training policy—learned by the network Q—are occasionally overridden with random actions. The probability of choosing a random action is governed by ε. Experience shows that for many problems, starting with $\varepsilon \lesssim 1$ at the beginning of the training, and then decaying ε over time within a configured number of training steps to an ending value $\varepsilon_{end} \gtrsim 0$ yields good training results.

This introduction of background knowledge suffices for the remaining discussions in this thesis and provides us with all the necessary vocabulary to understand the upcoming presentation of our implementation considerations and evaluations.

15.3 CREATING THE ENVIRONMENT

Requirements

For our system to be usable as an environment in which RL agents can be trained, a few functional requirements have to be met:

- All interactions between the environment and the agent happen in discrete time steps. Therefore, our prototype implementation used for researching and evaluating our previous contributions has to be reigned in and transformed so that it progresses in well-defined, discrete time units.
- The environment needs to be able to receive actions from the agent, has to understand and apply these actions as reconfigurations to its parameters, and then transition to a sensible next state using this applied action.
- After a transition, the environment has to provide both a meaningful observation of its new state and a reward signifying how good the newly reached state, influenced by the action of the agent, is.
- Since we would like to deploy trained agents within a BFT SMR setup, provided observations have to be deterministically derived from the system.

When actually implementing an RL setup nowadays, an additional non-functional requirement surfaces: The vast majority of up-to-date libraries with sufficient documentation and support are written in Python. Since our prototype was so far entirely based on BFT-SMaRt, and we implemented all of our additional functionality, including UDS, in Java, this gap had to be bridged, too.

We spent considerable time and effort implementing a setup for utilizing RL as a state-of-the-art optimization technique in a Java-based, deterministically multithreaded system. Since we consider the fulfillment of all of these requirements a part of our research contribution, this section will briefly describe the final resulting architecture.

15.3.1 *Discrete Time*

Transforming the Environment

To introduce discrete time steps into a system like ours, which is meant to receive client requests and at any time has tens, if not hundreds of parallel threads running to handle network communication, consensus, and request execution, we first cut down the complexity of the system considerably. Since our research was focused on the execution stage of SMR systems, i.e., on parallelizing request execution strictly after they have been ordered by the system's GCS, we could safely separate these two conceptually disjoint components apart. In practice, this meant that we isolated the execution stage of the SMR setup, which begins where BFT-SMaRt's consensus module provides a stream of totally ordered and batched threads. The remaining collection of classes consist mainly of (i) a thread acting as a mock-up for BFT-SMaRt's delivery thread, which usually supplies ordered threads as received from the system's clients, (ii) and our implementation of UDS. This also had the considerable benefit that we could test our execution stage locally on one system, without distributing multiple machines across a network, as long as we did not break

determinism and paid attention that re-integration with the rest of the system was still possible later on.

Within this shrunken down core of our system, we had to identify a way to introduce global barriers at which all threads could wait once a step had completed, and resume once an action had been received from the agent. The primary purpose of UDS is to introduce determinism into thread's accesses to shared data by selectively blocking threads so that their critical actions are ordered (cf., Chapter 7). Hence, all threads affecting determinism are already controlled by UDS by default, which meant that any global barriers would have to be introduced within UDS.

Introducing Discrete Time

Two possibilities existed for blocking all threads at deterministic points in time within our system: In between scheduling rounds and whenever a ByTI has been decided. We implemented both of these options using a Java `Condition` object called `stepCondition`, where threads starting a new round via `startRound()` (cf., Line 6 in Chapter 7) call `stepCondition.await()` once a step is finished, and thereby block themselves until they are woken up by a call from a different thread to `stepCondition.signal()` on this same `Condition` object. Later, we trained agents using both of these system setups, the results of which will be discussed in Section 15.5.

15.3.2 *Actions*

Receiving New Configurations

Without yet having explained how an agent would transmit actions to our UDS environment, let us assume that incoming actions arrive via a method call to `receivedAction(int action)` within our Java-based UDS implementation. In order to apply this action and transition to a new state, we utilized UDS' `reconfigure()` method in `startRound()` (Line 9), which allows for deterministic reconfiguration of UDS or other system parameters[2]. The main design decision here was to have a separate thread wait for incoming actions from the agent, which then acquires the lock of the `Condition` object at which the thread currently trying to start a new round within UDS is blocked (because of calling `stepCondition.await()`) and finally `signal()`s this waiting thread. This requires a mapping from the actions chosen by the agent to reconfigurations that can be applied to the system's parameters, which is rather trivial for easier parameters such as the number of primaries for the next round. For more complicated decisions, specialized translation logic could be implemented but was not yet needed for our evaluations.

15.3.3 *Observations and Rewards*

Obtaining Observations and Rewards

After the environment has received the action, it can run freely until the next call to `stepCondition.await()` occurs, either because the next round is starting or because a ByTI has been decided. During the execution, we mea-

2 As long as these updates to other parameters also happen under the protection of the internal lock used within UDS to protect its state.

sure and log deterministic metrics about the environment which are then summarized and provided to the agent. Depending on the version of the environment, these observations contain different metrics, such as the number of terminated threads in the last round or ByTI, the number of rounds finished during a ByTI or the currently estimated throughput calculated similarly to the one in Chapter 14. The full specification of observations can be found in Appendix 2.

Additionally, for training purposes a few extra non-deterministic metrics (such as the CPU time utilized vs. the lengths of rounds, or request latencies) were measured, which together with the observation could be used to calculate the reward for a given step. This indeterminism is acceptable during training as long as only the reward calculation is affected by it, since the finished DQN representing our policy will only receive observations from our runtime environment when deployed to a live system.

15.3.4 *Bridging the Language Barrier*

At this point, our environment was more or less ready for use within an RL setting. As mentioned before, however, the only realistic options for implementing a custom DQN-based RL agent require the use of Python. We will detail this implementation in the following section, but can already present our final architecture for connecting our Java-based environment to the DQN implemented in a Python framework here.

Architecture

On the Java side, we implemented an encapsulating class running in its own thread, which opens a blocking ZeroMQ socket[3] on which the environment waits for actions and replies with observations. The Python environment similarly utilizes a ZeroMQ socket, on which it waits for observations, determines an action and sends it out via the socket. Thereby, both environments move forward in lockstep.

Figure 15.2 shows an abstract overview of this architecture, including the sequence of steps that takes place in each training loop. To follow this diagram, one can simply start anywhere in the loop (e.g., the moment an agent sends an action to the Java-based environment using the øMQ socket), and then follow the sequence of arrows through the logic of the application.

15.4 IMPLEMENTING THE DQN APPROACH

Framework Choice

As mentioned before, all the currently well-maintained, popular, and well-documented libraries for working with custom ML problems are exclusively written in Python, with only few and comparably badly documented alternatives in other languages available. The two most popular choices for using general ML approaches implementing or implementing custom ones are Google's *TensorFlow*[4] and Facebook's *PyTorch*[5]. Both are open source frameworks capa-

3 https://zeromq.org/

4 https://www.tensorflow.org/

5 https://pytorch.org/

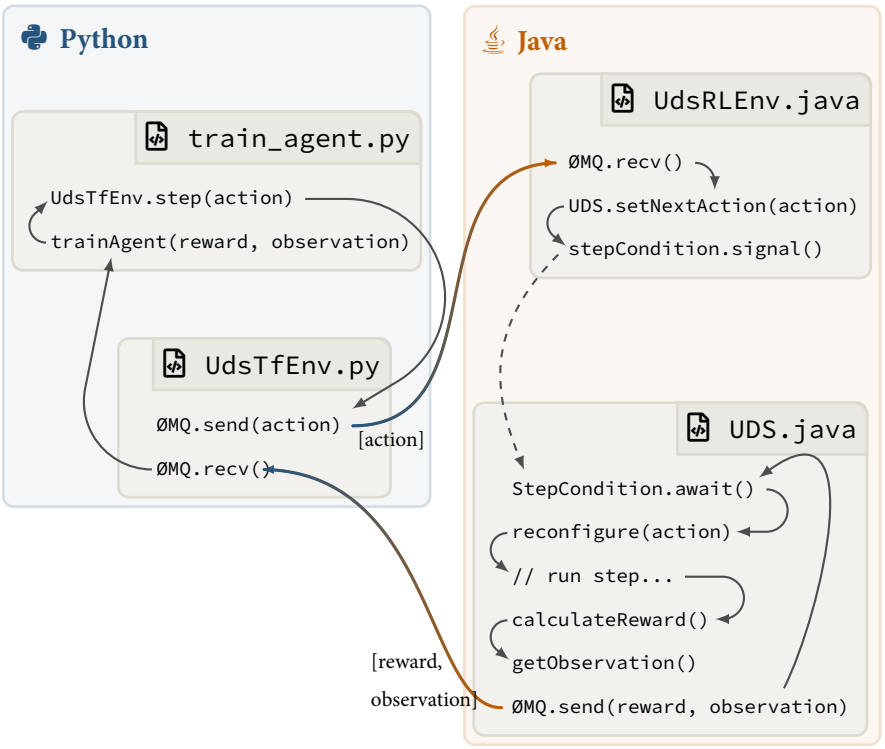

Figure 15.2. The main architecture which transparently hides the Java-based environment from our Python-based RL framework, enabling our training loop via øMQ communication between the two. Also shown is how the Java environment supports discrete time steps by waiting on `stepCondition` and receiving notifications via `stepCondition.signal()`.
Arrows show the temporal sequence and which logical action follows the next. The dashed arrow signifies an indirect wake-up call to the currently scheduling UDS-Thread waiting on the `stepCondition`. The observations are encoded in JSON format and transmitted via øMQ sockets.

ble of utilizing current hardware to accelerate ML algorithms, and the choice of the correct framework comes down to preference for most projects. For Reinforcement Learning in particular, however, TensorFlow provides better support by providing a framework for agent-based learning and the use of DQ-networks. Hence, our choice for implementing a DQN RL-Agent fell on TensorFlow, with the additional `tf-agents`-library providing RL specific abstractions.

We already introduced our overall architecture in Figure 15.2. Within it, we can see that the parts of our setup based on TensorFlow consist mainly of a training script and a custom `UdsTfEnvironment`, which hides the øMQ-based communication link to the Java environment.

15.4.1 *Hyperparameters*

A training run is primarily specified by the hyperparameters for the Python environment and metadata for the Java environment, describing for example the way in which threads are generated to simulate load on the system. The

training script receives all hyperparameters for a training run via command line when starting up.

Motivation

Since success with training neural networks such as DQN is highly dependent on the proper selection of values for their hyperparameters, finding suitable approaches for tuning these parameters—also often called hyperparameter optimization [93] can be viewed as an entire research area of its own [82]. Since we implemented a custom RL environment for a niche field where, to the best of our knowledge, no prior experience exists about the best hyperparameters apart from general recommendations derived from similar problems, we had to first establish a baseline. For understanding what we mean by this, consider that we were concurrently designing, implementing, testing, and optimizing the entire framework during this process, meaning there were multiple moving parts at all times. Numerous training runs with early prototypes were plagued by abysmal performance (i.e., resulting agents were utterly unable to learn usable policies), which was not always easily attributable to a single cause. In order to remove doubt about the hyperparameters being responsible for these initial problems, we set out to finding a baseline of hyperparameters that was likely to lead to successful training with the approximate network and observation sizes and shapes we were using in our environment. In other words, we attempted to find a collection or range of values for each individual hyperparameter that would work for our general problem.

Approach & Methodology

Since hyperparameter tuning has been researched for many years, there are numerous proper methods for selecting the best hyperparameters for a novel ML problem. However, since these are once again optimization approaches, they always rely on the existence of a well-defined objective function, i.e., goal, as well as metrics that can rank resulting parameter combinations (cf., Chapter 4). Given that this was our very first experience with RL, and we had to constantly learn new methods ourselves, update the framework, and fine-tune our approaches during this part of our research, we utilized one of the oldest optimization techniques in existence: Manual tuning and exploration of parameters and effects, also sometimes cheekily called grad-student descent (GSD) [82]. However, we still conducted this part of our research properly and documented the intermediate results and the baseline values we thusly settled on for our subsequent efforts. For gauging the goodness of hyperparameters, we created a training run setup with significantly reduced complexity, especially in terms of unpredictable randomness, by only feeding the system with one type of request with an easily parallelizable application profile, and at predefined levels of request rates that did not jump around too erratically. Using this setup, we proceeded by adjusting the value of a single hyperparameter during a cohort of multiple training runs, and comparing all results gathered within these adjustments of one hyperparameter. The best values for a particular hyperparameter were then chosen by picking the top runs from this cohort, where a good run was determined by looking at the learned policy of the agent in detail using time-series plots of its actions vs. the current load on the system. We then used our domain knowledge to identify sensible policies, i.e., those that most likely maximized throughput and minimized latencies. In

less general terms, the single action our agents were allowed to perform was used to adjust UDS primaries from between 1 through 8, and following our experiences from Chapter 14, we chose those runs as the best ones where chosen primary counts were close to the number of currently active clients. Figure 15.3 shows one single episode from towards the end of one such training run where the training worked acceptably well, and which was in this case used to determine that more episodes, i.e., longer training runs, can be beneficial to an agent's performance.

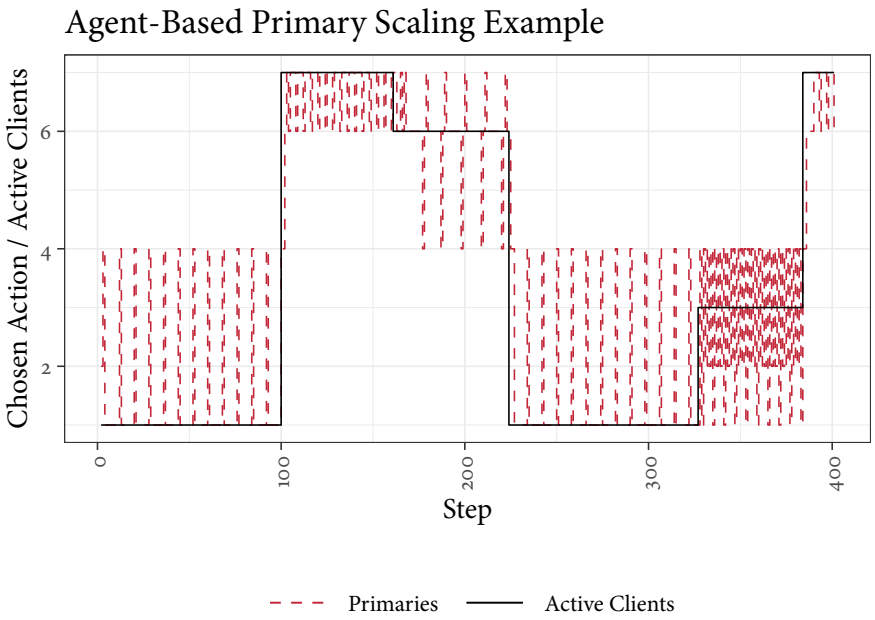

Figure 15.3. Example episode from hyperparameter optimization runs. The black line shows the number of currently active clients, the red dashed line displays the actions chosen by the agent.// `numEpisodes`=600, `episodeSteps`=400, α=1×10^{-3}, a (64,32) neural net (cf., Table 15.1 for an explanation), `replayBufferSize`=60000, and `miniBatchSize`=10

Example

This specific measurement used the version of our environment which returned an observation after every round. In the figure, the currently active number of clients synchronously sending requests is shown in black, and in red we plotted the agent's decisions about how many primaries a next round should have after receiving observations containing details about the last three rounds. We can see that overall, the behavior of the agent is rather similar in nature to the behavior of the rule-based algorithm we developed in Chapter 14. The agent frequently tries to reconfigure itself to find out whether a change of primaries positively affects performance, and in case it does, it permanently switches to a higher number of primaries. Conversely, if performance rapidly drops, it goes back to one primary as the safe base configuration. One major difference in the behavior of this agent when compared to our rule-based algorithm is that the agent tends to prefer larger steps when testing new primary configurations, and that sometimes it oscillates around the currently optimal

number of primaries instead of settling on the optimal value and exploring from there.

Determined Hyperparameters

Table 15.1 presents all hyperparameters our final implementation uses, the ranges we tested for these hyperparameters, and the range or collection of values we settled on for further refining our system setup. For some hyperparameters we did not explicitly test values systematically ourselves, but used values recommended by peers, both within the faculty and on the Web. These values are marked with "–" in the middle column.

Hyperparameter	Tested Range [steps]	Final Value(s)
`numEpisodes (e)`	10 - 600 [10]	400-600
`episodeSteps (s)`	10 - 1000 [10]	400-500
`initialSteps`	10 - 1000 [10]	`s`
`replayBufferSize`	10 - 20000 [10]	$(\mathtt{e} \times \mathtt{s}) \div 2^{\dagger}$
`miniBatchSize`	1 - 50 [10]	5 - 10
`DQNLayers`	(1-100) & (16-100,8-64) [7]¶	(32,20) - (64,32)
α	1×10^{-1} - 1×10^{-4} [10]	3×10^{-3} - 1×10^{-3}
γ	—	0.95
τ	—	0.5
$period_{\tau}$	—	50
ε	—	$0.99 \rightarrow 0.1^{*}$
`lossFunction`	—	Huber loss [46]
`optimizer`	—	Adam [55]
`activationFn`	—	ReLU [41]

Table 15.1. Hyperparameter ranges we manually tested, and the baseline values we settled on for our further research. For each hyperparameter testing range, the value in parentheses specifies the number of tests in this range we performed, with values for these tests being roughly uniformly distributed between the range limits. Parameters which we did not extensively (or at all) test ourselves are marked with —.

† After discussion with a colleague the size of the replay buffer was set to half the total steps taken during the entire training run.

¶ $(l_1, l_2, \ldots, l_i)$ stands for i hidden fully connected ("dense") layers in the DQN, with l_i neurons on each layer.

* ε was set to decay from 0.99 to 0.1 within `numEpisodes/3` episodes.

The `lossFunction`, `optimizer` and `activationFunction` are listed for completeness' sake, were chosen after researching popular state-of-the-art practices in RL and discussions with a colleague, and do not need to be further explained here.

15.4.2 *Calculating Rewards*

Influence of Rewards on Results

We quickly learned that the rewards our environment receives each step greatly influence whether training runs produce agents with sensible policies. Intuitively, this makes perfect sense, as after all rewards are the only way to communicate to the agent whether it is learning a desired policy or not. The main question then is how to best calculate a reward for each step. This problem might

sound simple at first, and for some optimization problems it is. For example, most games usually inherently contain some sort of score or counter, and rewards can simply be related to increasing score counters. In other problems, such as controlling robots, usually the reward is given when a robot moves in the correct direction or manages to stay upright.

Regarding our optimization problems, we specified better performance of the system as our goal, which we can measure mainly via throughput rates. Therefore, setting rewards for our system might seem like a simple task: Reward an agent for each step which achieved high throughput. However, this does come with immediate difficulties, as throughput rates are dependent on both an agent's action and the current overall system load. Put differently, if there are no clients active at the moment, then we would experience low throughput rates simply because there are no requests to put through our system, but we should not punish our agent for this.

Domain Knowledge vs. Straightforward Rewards

Therefore, we have to incorporate some degree of domain knowledge into our reward calculations, where we endow the formula which yields results for each step with knowledge about the system's behavior under certain circumstances. This is fine in general, but quickly devolves into fine-tuning even more parameters in an equation of ever-increasing complexity, where at the end, instead of calculating rewards we could just as well have built an algorithm similar to our rule-based approach in Chapter 14. Finding a good balance between these two extremes was potentially the greatest problem we faced during our final research efforts, and unfortunately it is one we have not solved in finality.

By this, we mean that regardless of the rewards we used to educate our agents, the only ones that remotely worked were those that incorporated a great deal of domain knowledge, which we ourselves gained by way of experiencing and interacting with our system for years. Conversely, and to our great chagrin, all approaches which tried to utilize simpler rewards failed, in the sense that agents never seemed to learn sensible policies.

Reward Shaping

To improve our reward calculations, we tried to consult related work, and found that the process of deriving a way rewards are best calculated is also called *reward shaping* ([99], [42]). However, reward shaping is focused more on how existing, already working rewards can be optimized to improve learning times. To understand this, consider that one central problem RL agents face when exploring an environment is often the temporal disconnect between current rewards and long-term rewards gained in the future, which are still ultimately based on correctly chosen actions at the current time. This temporal delay between the agent choosing an action and later receiving a reward ultimately rooted in this earlier action has been called *reward horizon* in prior literature [99]. The theory behind RL says that given enough time with the environment, and with the help of techniques like experience replay, agents can eventually learn optimal policies even in environments with large reward horizons. One deciding factor of *how fast* agents can figure out an optimal policy in complex environments, however, is the size of reward horizon. In other words, in environments with short delays between actions and pertaining rewards rooted in these prior actions, agents would require less training

time, i.e., steps in the environment, before becoming adept at optimizing the problem at hand. Our problem, unfortunately, was at this stage not yet how to improve training times or fine-tune the system, but how to get agents to learn proper policies from sensible rewards at all, since even with hundreds of thousands of steps in our environment agents did overall not learn satisfactory policies.

In the remainder of this chapter, we will therefore briefly summarize the results we did manage to extract from our many attempted agent training runs, and present future work we identified.

15.5 PRELIMINARY EVALUATIONS

Goal Recap

Let us quickly remind ourselves of the original research goal of this chapter: We were to demonstrate the suitability of RL as an optimization approach by showing the performance of a self-optimizing system prototype under different loads and application profiles. Our previous, rule-based approach, while working admirably given its simplicity, had the drawback of only being capable of adjusting one single parameter—UDS primaries—, based on a single metric. To directly see whether our RL-based approach could be superior to the rule-based approach, we wanted to set up a comparison where we use the exact same request types and similar workloads as the one used in the evaluations of Chapter 14[6]. We first aimed at training an agent to perform at least as well as the rule-based algorithm in configuring primaries, and had some initial success with this endeavor, as has already been shown in Figure 15.3. In order for agents to produce deterministic reconfiguration decisions outside the training environment, when they are replicated among the state machines of the SMR setup, we ensured that TensorFlow as our framework of choice supports this determinism by setting special variables before starting an agent [109].

Planned Approach

Afterwards, our plan was to allow the agent to not only modify UDS primaries, but also the number of steps each primary thread would get per scheduling round, in an attempt to demonstrate that RL-based optimization can control multiple parameters and learn from more metrics than just simply the estimated throughput. Coupled with application profiles that demand multiple critical actions, such as our previously introduced $L_1U_1C_{500}L_1U_1$ or $C_{250}L_0C_{50}U_0L_1U_1$, we expected improved performance if a system could grant both required steps to such requests within one single scheduling round, instead of having to delay them to the next round.

To set this experiment up, we first implemented a random load generator which produces workload patterns that are very similar to the ones of our 2-peak workload seen in the previous chapter. Then we increased the agent's action space, so it could decide between 1-8 primaries and 1-2 steps, i.e., 16 actions in total, when combined. We proceeded with executing hundreds of training runs over the course of several months, tuning parameters within our

6 We say similar workloads because using our exact 2-peak workload from the previous chapter could easily lead to overfitting, i.e., an agent learning the load pattern and predicting it perfectly, while actually being inept in real systems with different loads

previously established guiding limits, and constantly trying out variations of reward calculations.

Reward Calculation Example

Most reward calculations we trained with were based on variations of the formula

$$r_t = \texttt{normalize}(\Omega_t) \times (1 - \texttt{normalize}(\texttt{mean}(\delta)_t)) \times \varphi$$

, where Ω stands for throughput and δ signifies the latencies of threads executed in the last step. Simply put, we attempted to calculate a score using achieved throughput and the inverse of average latencies experienced by these threads. φ was used to slightly scale rewards in size to observe possible effects on training. Ω was calculated using terminated threads during the last step, and wall-clock measurements of the duration a step took. In the case of the environment that was based on ByTI-based stepping, this amounted to the same calculation as the one used in Chapter 14. For the round-based stepped environment, we primarily used domain knowledge-driven rewards, such as the squared difference between configured primaries and currently active clients. `Normalization` of values was performed by first observing the range of values that can be expected after several thousand steps in a specific training run, then manually configuring the lower and upper limits of the basic normalization formula $\texttt{normalize}(x) = (x - \texttt{lowerLimit}) \div (\texttt{upperLimit} - \texttt{lowerLimit})$, and re-running the training run to its end. To understand why normalization of reward (and also observed) values is usually done when training DQ-networks, one needs to understand that common loss functions—including Huber loss used in our training—calculate the distance between expected Q-values and actual Q-values for the purposes of adjusting network weights by usually squaring these values, to approach correct values faster when estimates are off by a large margin. This, however, also means that when non-normalized values are fed directly from the environment into the equations governing neural network training, large numbers tend to lead to diverging behavior due to this inherent squaring[7]. Normalizing reward and observation values helps with avoiding this problem. To test whether this applied to our case, too, we tried feeding non-normalized values to the agent during some training runs, with no discernible success or difference to the other runs, except for abnormally large loss values we observed during training using TensorBoard[8] plots.

Normalization of Values

Example Policies

Partially successful training runs were obtained using rewards incorporating domain knowledge. For example, when we explicitly punished agents for configuring more primaries than there were currently active clients, in order to avoid RFDs, e.g., by subtracting a high fixed value from the reward whenever this happened or by adjusting the reward negatively by the squared distance between clients and primaries, some successes were achieved. However, agents never seemed to be able to fully learn this by themselves just using throughput

7 It is to be noted that not only loss is affected by this, but the actual process of applying weight updates to the neural networks has many pitfalls too. This is explained excellently in [97].

8 https://www.tensorflow.org/tensorboard

rates, even though those were of course noticeably affected by RFDs during our training runs, as manual inspection of countless training episodes proved.

One example of how agents were only partially successful while learning from rewards calculated purely from throughput and latencies can be seen in Figure 15.4.

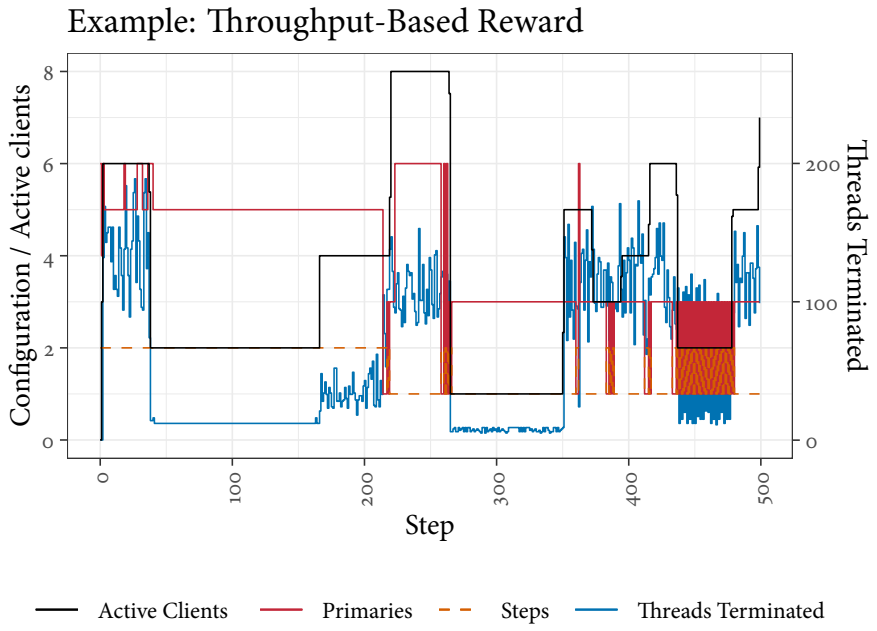

Figure 15.4. Example episode from training runs trying to teach step scaling. The black line shows the number of currently active clients, the red solid line displays the agent-chosen primary configuration, the dashed line shows the chosen steps, and the blue line is referencing the second axis, showing the number of threads that terminated in the last step in our ByTI-stepped environment.// `numEpisodes=200`, `episodeSteps=500`, α=2×10^{-3},`DQN layers=(32,20)` (cf., Table 15.1 for details), `replayBufferSize=25000`, and `miniBatchSize=5`

In this plot, we can observe that the agent failed to learn to properly scale primaries to avoid RFDs, and its additional control over steps did not lead to better performance, either. The only minor success that can be claimed here is that the agent seems to have understood the value of reconfigurations and at least partially tries to adhere to the best practices for primary scaling. As can be seen from step 220 to 250, however, instead of scaling up steps to two, which would have led to better performance, the agent actually decided to scale steps down to 1, yielding worse performance than was observed in the beginning of the episode between steps 0 and 75, even though there were fewer active clients and configured primaries in this period.

Effects of Reward Decay

Another modification we had some training successes with was the punishment of long periods of inactivity, i.e., many rounds of non-changing configurations. To this end, we introduced a counter signifying how many rounds ago the last change to the UDS configuration parameters took place, and subtracted this number from the final reward value, weighted by an additional

parameter. This often led to desirable behavior like the periodic discovery attempts seen in Figure 15.3.

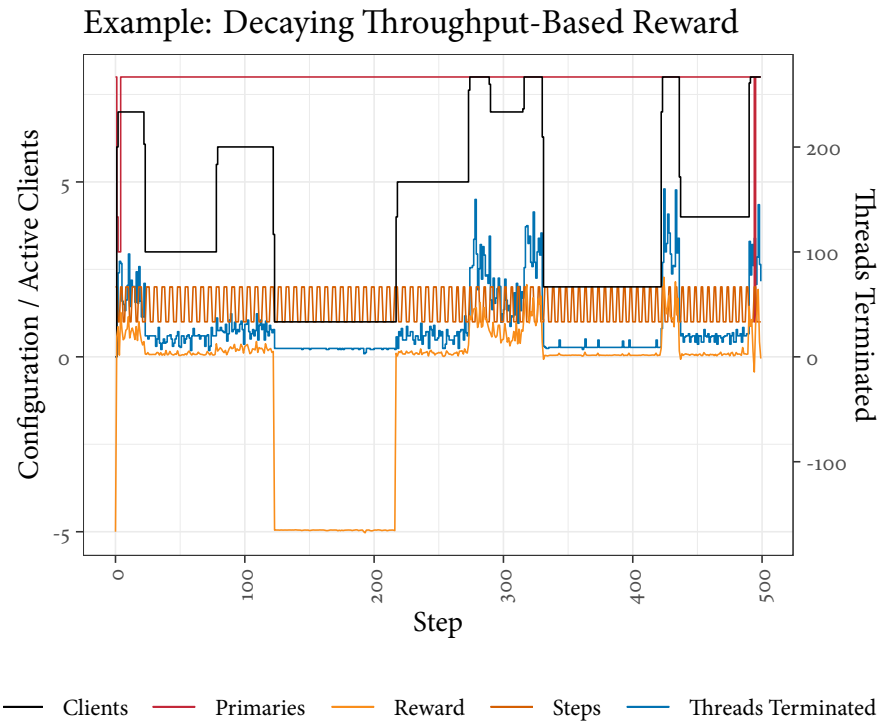

Figure 15.5. Example episode showing how agents react to decaying reward punishing inactivity. The black line shows the number of currently active clients, the red solid line displays the agent-chosen primary configuration, the dashed line shows the chosen steps, and the blue line is referencing the second axis, showing the number of threads that terminated in the last step in our ByTI-stepped environment. Additionally, reward is drawn in orange.// `numEpisodes=200`, `episodeSteps=500`, α=2×10^{-3},`DQN layers=(50,25)` (cf., Table 15.1 for details), `replayBufferSize=35000`, and `miniBatchSize=10`

Example

A last example shall display how agents also often managed to find unique but ultimately unusable policies whenever we tuned parameters in the system For this, consider Figure 15.5. In this plot, we can see that the agent learned to avoid the decaying reward punishments by constantly alternating UDS steps, which counted as a configuration change in this instance. However, at the same time, it did not learn to properly scale primaries or deduce the relationship between its actions (i.e., primaries and steps) and throughput, despite the large negative rewards dispensed for achieving abysmal performance during low load phases.

Time Constraints

It is evident from looking at these examples that using such agents in a full system environment and pitting them against our rule-based algorithm from before would not lead to any improvements regarding the performance of the system. An unfortunate and harsh truth is that overall, training these agents and tuning our RL setup did not lead to any of the major breakthroughs as we had hoped, and in the end time was a deciding factor which led us to stop these measurements and focus on writing up our results instead. One main reason

for this was that especially the later training runs in our ByTI-stepped environment, which were meant to produce agents that could be directly plugged into a real system for comparison to the rule-based approach, took upwards of 4 hours each on our available hardware. Even with stopping training runs early due to insights gained during the training, e.g., by using TensorBoard plots of loss values and chosen action distributions, we still could not complete more than 3-6 training runs per day, which ultimately was not enough to optimize the training process to a point where it produced sensible policies.

Therefore, in the following discussion, we will briefly summarize those parts of our research that still contribute to the state-of-the-art, and identify several key aspects of future work with which this approach should be pursued in the coming months and years.

15.6 DISCUSSION

The final research goals tackled by this chapter were to investigate ways to realize a fully self-optimizing SMR system, by first researching the most promising approaches, choosing a method, and then implementing and evaluating this method in a prototypical system. We achieved most of these goals and contributed to this field of research by laying fundamental groundwork required for building self-optimizing deterministic systems.

First, our research into various optimization methods led us to classifying approaches and discarding those that would not fit our use case and prototypes. We motivated our process and how we settled on RL as our chosen approach in Chapter 4, Chapter 12, and finally also in the beginning of this chapter where we further explained how RL and specifically DQL work and fit our circumstances.

Contributions

Afterwards, we solved multiple problems regarding the construction of a working measurement setup for training RL agents within the context of deterministically multithreaded systems. The main contribution here is a finalized and fully working training framework, taking care not only of interfacing TensorFlow as the underlying RL library with our BFT-SMaRt-based system prototype, but also of the entire measurement and analysis workflow required for researching such systems. A central `TestCoordinator` component within our architecture is capable of reading detailed training run specifications from a specialized SQLite database, which includes all information required to rerun a single training run over and over again in the pursuit of scientific reproducibility. Creating training run specifications and saving them in the mentioned central database happens via an intuitive command line tool which future researches can easily use to continue our research or reproduce our current results.

Resulting Platform

Finally, using our entire measurement setup, we performed multiple hundred training runs spread over thousands of hours of work to tune, fix, and build the system, and afterwards to measure and train agents capable of optimizing our prototype. During this process, we identified several usable ranges of training hyperparameters as presented in Table 15.1, which can be used as

a stable baseline for training agents as long as the surrounding system parameters do not change significantly. While we did not ourselves achieve great success regarding the training of intelligent agents capable of exploiting their learned system knowledge to optimize performance, we have still finished important groundwork for any future researchers trying to finalize RL-based, fully self-optimizing systems.

Future Work

The future work we identified in this regard could therefore focus mainly on utilizing this platform to further tune the training process in order to produce capable agents. Furthermore, these trained agents would then have to be plugged into our full, distributed system setup with several replicated machines, in order to live-test their performance under real conditions. From the data we have gathered so far, it is our conviction that this approach would eventually yield agents that can take in all the metrics that can be deterministically measured during runtime and produce intelligent reconfiguration decisions maximizing throughput and minimizing request latency.

These results represent the final contributions of our thesis and the conclusion of the last of our originally specified research goals as presented in Chapter 5. In the upcoming final chapter of this part we will provide a brief summary of our comprehensive work on performance optimizations for deterministically multithreaded SMR systems.

SUMMARY

16

The previous four chapters contained in-depth discussions of our efforts towards optimizing the performance of fault-tolerant, SMR-based, deterministically scheduled systems. We will now briefly recap the main contributions in this last chapter, before ending the thesis with a last part providing a larger summary and an outlook on this interesting field of research.

The overarching main goal that led us through this part of our research was to determine ways of utilizing the reconfigurability of UDS in order to improve the overall performance of SMR systems. We initially looked at general optimization approaches for complex systems, which we presented in Chapter 4, and identified two approaches that we would try to utilize in our further work, as detailed in Chapter 12. Afterwards, we identified several problems that needed solving before we could apply our chosen optimization approaches to our systems.

ByTI

Creation of Prototype System

One of the major problems in this regard were missing methods for deterministically measuring metrics in BFT SMR contexts. We tackled this problem by developing a novel distributed clocking mechanism capable of providing such metrics to SMR replicas even in the face of failures and adversaries. Considerable efforts were then poured into building a fully working prototype of an SMR setup incorporating this novel clocking mechanism. The resulting system prototype was used to provide evaluations of this solution. The algorithm behind our mechanism, an analysis of its behavior, as well as the evaluation results were presented in Chapter 13.

Rule-Based Optimization

Optimization Algorithm & Measurements

The system prototype also served as the basis for our further research, which looked into exploiting knowledge we had gathered about the behavior of such systems during our previous years of working with them. In particular, we set out to designing an optimization algorithm that was based on inferred rules we uncovered regarding the relationship between the performance of systems scheduled with UDS and the number of primaries that were configured during particular circumstances, such as low and high load phases (cf., Chapter 8). After specifying this algorithm, we implemented and integrated it into our prototype platform, and performed thorough evaluations, all of which we detailed in Chapter 14. The results demonstrated in a live system prototype that

- UDS reconfigurations during runtime can have significant impacts on performance,

- our solution can be a drop-in replacement for existing platforms utilizing deterministic scheduling, adding flexibility without any apparent drawbacks, and finally
- that even with a simple self-optimizing algorithm based on one metric and adjusting one parameter, considerably improved system behavior can be achieved [4].

These results encouraged us to proceed with our plan to further introduce dynamicity to our system and attempt to pave the way towards fully self-optimizing systems.

RL Agent-Based Optimization

Obtaining Required Background Knowledge

To this end, in our final research endeavors, we dove into the considerable body of work surrounding Reinforcement Learning methods, in order to implement an agent capable of recognizing by itself the correct metrics and actions required to improve system performance for a wide variety of applications and setups.

Summarized Contributions

The first problems we faced in this regard were centered around implementing an RL-based training setup using our prototype system, since the latter was not initially prepared to progress in discrete time steps. After solving this problem and integrating the Python-based RL-framework TensorFlow with our Java-based environment, now capable of stepping through individual rounds or ByTI in sequence, we put considerable effort towards finding a reliable set of hyperparameters in the large space of possible tunings for DQN-based RL approaches. This was followed by months of modifications to the overall prototype, tweaking observed environment metrics and especially reward calculations in the pursuit of sensibly looking policies our agents were to learn. Chapter 15 introduced the required background knowledge for following us through this process, and presented our work and all of our findings, including our preliminary evaluation results using examples from the training runs we performed. Our main contributions lie in the novel application of RL to this specific optimization problem, for which we laid solid foundations and provided future researchers with an entire framework capable of training RL agents by interacting with a real deterministically multithreaded environment. Finally, we identified the most promising avenues for future work in this area of research.

This summary concludes the presentation of our contributions and finishes the discussion of the research goals we initially set out to accomplish (cf., Chapter 5).

In the next and final part of this thesis, we are left with providing an overall summary of our entire body of work we finished during our PhD, and close with an outlook and our opinions about the state of this area of research.

V

OUTLOOK & CONCLUSIONS

SUMMARY AND OUTLOOK

17

In this final chapter, we will briefly summarize the contributions of our thesis, interspersed with short recaps of how we approached each problem and achieved our research goals as set out in the first part of the thesis (cf., Chapter 5).

17.1 SUMMARY

Main Goal Recap

Let us first recall the main goal of this thesis, as formulated in the beginning, which read as follows:

MAIN GOAL Utilizing deterministic multithreading, investigate approaches that can enable state machine-replicated fault-tolerant systems to self-optimize themselves during runtime, while requiring minimal input from developers or operators. Research the most promising approaches, using—where possible—prototype implementations, measurements, evaluations, and comparisons to similar existing solutions.

Using this main goal, we defined several specific research goals, which will be compared to our contributions in the following sections. However, before we introduced the definition of this main goal and its sub-goals, the initial chapters of our thesis introduced each of the terms mentioned in both the thesis title and the goals.

17.1.1 *Foundational Knowledge, Related Work, and Background*

FT Background Knowledge

In Chapter 2, the foundational ideas and terminology of fault-tolerant distributed systems were detailed, and we learned about SMR as a concept that would follow us through the entirety of our work. To illustrate our introductory explanations and give a pedagogically interesting first overview of the most common approaches to achieve FT, we built up an imaginary system from scratch, starting with a completely fragile system $v0$ all the way up to distributed, replicated system $v5$, capable of tolerating even Byzantine faults.

Deterministic Scheduling Background Knowledge

In the subsequent Chapter 3, these foundations were expanded by our introduction to deterministic multithreading as a tool for parallelizing the execution of SMR systems without sacrificing determinism. The goal of this is to boost their performance, naturally, but came with a few new terms and concepts we also introduced to the reader. In addition, since this thesis is mainly situated at the intersection between the fields of deterministic multithreading and optimization, we started detailing first related work in the former field. This also frequently mentioned certain drawbacks this related work has in terms of its flexibility, and started teasing UDS, i.e., our first contribution in this thesis.

Optimization Approaches

Finishing the foundational part, we followed this by a short overview over the rather large field of optimization approaches in Chapter 4. Here, we also introduced the concepts of metrics and optimization goals, in addition to outlining our process of choosing the two main optimization approaches we pursued in this thesis alongside a short discussion of each approach.

Thereafter, the fifth and last chapter of the first part used the established terminology from the previous chapters to define our research goals, the first of which we have reiterated above.

17.1.2 *UDS*

Specification and Implementation of UDS

In order to enable SMR systems to self-optimize during runtime, we surmised that among other requirements, a flexible scheduling algorithm could allow for the necessary flexibility in the execution stage of such systems to react to sudden changes in their environment. This had also been shown in previous work (cf., [38]) and was the basic motivation for our first research efforts. Building on prior work that was underway when we joined the research group of Prof. Hauck, we contributed to the first published deterministic scheduling algorithm that could be reconfigured during runtime [5]. This conceptual algorithm, called the Unified Deterministic Scheduler, then had to be implemented, tested, bug fixed and optimized in order to be usable for our further research. Therefore, our next efforts were aimed at investigating UDS' behavior in an event-based simulation framework, which led to the first insights about its potential to optimize system performance by utilizing its reconfigurability. These steps taken together accomplished in full our first research goal as presented in Section 5.1.1.

17.1.3 *Efficiency Optimization*

Resource Efficiency Results

While implementing UDS in a system prototype on the basis of the BFT-SMaRt framework, we noticed through discussions within our research group that deterministically multithreaded SMR systems presented a unique opportunity to vertically scale underlying hardware, which would otherwise (without parallelized execution) a relatively meaningless endeavor. Therefore, we decided to test our first actual UDS implementation by using it to enable parallelization in an SMR system, but without yet exploiting its dynamicity. We followed through with thorough measurements in Chapter 10, showing that vertically scaling SMR systems is not only possible but can unlock significant cost savings in the right environments, i.e., in cloud settings where automated hot-plugging of hardware resources is possible [3]. Our second research goal as defined in Section 5.1.2 was thereby achieved, too.

17.1.4 *Performance Optimization*

For our third and last research goal, we identified three additional sub-goals which were to be met in order to accomplish true self-optimization for multi-threaded SMR systems (cf., Chapter 12).

ByTI

The first of these sub-goals aimed at finding a solution to provide deterministic measurements in a BFT setting, on the basis of which reconfiguration decisions could be reached during runtime. To this end, we introduced Byzantine Time Intervals, a novel mechanism for establishing a common time basis in BFT SMR systems, which is largely unaffected by failures and attacks and can be used to observe such metrics as request arrival rates or throughput by counting threads within each ByTI.

Rule-Based Self-Optimization

The second research sub-goal tied back to the first of our initially chosen optimization methods, and stated that a proof-of-concept system should be built with which all prior hypotheses about UDS' performance boosting reconfiguration effects and the usefulness of ByTI could be tested. We implemented such a system and equipped with a simple algorithm that exploited our domain knowledge about performance effects of the UDS parameter specifying the number of primary threads per scheduling round. This was followed by detailed evaluations using multiple request types emulating several common application profiles, which followed a workload pattern similar to what an SMR system could experience during daily load cycles. Our results showed that self-governing performance optimization by way of reconfiguring deterministic scheduling during runtime is entirely feasible for SMR systems and yields significantly improved performance under many realistic conditions [4].

Self-Optimization based on RL agents

Finally, the last of our sub-goals required to complete our third main goal in the context of performance self-optimization had us consider advanced optimization approaches, which led to us choosing Reinforcement Learning as the method with which we were to prove a more generalizable self-optimization solution possible. We approached this problem by first researching the state-of-the-art in RL methods, followed by an implementation of a DQN approach using TensorFlow. Problems encountered during these efforts were numerous and are detailed in Chapter 15, but were largely overcome. In the end, we produced a fully working RL training environment for training DQN-based RL-agents and teaching them the art of optimizing multithreaded systems. The envisioned final result, i.e., a functioning self-optimizing system prototype utilizing a trained RL-agent to choose reconfiguration actions, was not fully achieved in the end. This was mainly due to a persistent set of problems relating to the learned policies of our agents, but our contribution can still be considered substantial even towards this last goal. The training platform for creating RL agents in this specialized field of research is an important milestone towards the eventual adoption of this advanced optimization technique for SMR systems.

Therefore, we fully accomplished two of the three sub-goals of the third and final main research goal, with an enormous percentage of work also completed relating to the last sub-goal.

Overall, we consider our research contributions towards the overarching main goal of the thesis substantial and significant, and hope that this thesis managed to convey not only our ideas and how we scientifically approached the problems that presented themselves, but also successfully captured our general fascination with these research topics and the significance of our contributions towards this area of research.

17.2 FUTURE WORK

Before shortly closing with an outlook and personal opinions about the topics discussed in our work, we would like to give a short overview of future work and interesting open research questions we encountered during our studies.

Open Questions in Deterministic Multithreading

UDS brought the field of deterministically scheduled systems forward a major step, but of course there are still many improvements that could be made not only to the algorithm itself, but also to the general state of this field of research. Let us start by listing a few open areas in which UDS itself could be further improved.

Model-Checking UDS

First, although we consider UDS quite mature and well-tested by now, having scheduled millions upon millions of threads using a wide variety of request profiles and workload patterns, the fact of the matter is that UDS has not yet been thoroughly model-checked using, e.g., tools like TLA$^+$ [57]. We briefly mentioned this while introducing UDS, but can spend a few more words on this topic here. A completely model-checked algorithm would certainly instill even more trust in the underlying scheduling logic of UDS, but the effort required to fully model UDS in a language such as TLA$^+$ is probably beyond what a single researcher could achieve without spending several months, or maybe even years, just on this problem alone. We confidently say this because (i) we had a closer brush with this topic during the early years of our PhD, when we discussed the possibility of fully model-checking UDS, especially after we found and fixed the bugs detailed in Section 8.2, and (ii) because researches within our research group have direct experience with model-checking similarly complex algorithms due to spending years on the problem of verifying a single consensus algorithm for BFT SMR systems. Nonetheless, in the context of interesting future work, this could still be a worthy endeavor, especially if UDS should be used in critical systems.

More Features for UDS

Second, there are some features missing from the current version of UDS that have been planned for a long time but have never been fully refactored into UDS as it is today. One such feature is the support for `wait/signal` operations (cf., [96]) for threads scheduled by UDS. ADETS-MAT supports this feature, and preliminary work has been done within our research group that theoretically prepares UDS for this, as well, but so far we have not had the time or resources to pursue this further. A second feature would handle deadlock detection in scheduled threads. This also ties back to model-checking the core

logic of UDS, because as we briefly mentioned in Section 8.2, we can also not yet be completely sure whether UDS itself might possibly introduce dead- or livelocks given sufficiently complicated requested locking patterns.

Optimized Schedules Utilizing Prior Knowledge

And finally, circling back to the topic of optimization, there are of course additional approaches utilizing UDS' dynamicity that we have not had the opportunity to pursue. One example of this could be the incorporation of prior knowledge about application request locking patterns, similar to how Storyboard optimizes schedules based on such information [53]. If we had additional knowledge about locking patterns, optimization algorithms aimed at solving optimal scheduling problems could try to find the best UDS total orders for achieving special goals like computational resource efficiency or simpler ones like low request latencies. In this case, however, the approach changes from a pessimistic one, i.e., a fully deterministic system, to an optimistic one, where systems need to communicate with each other in case a prediction was wrong and needs to be rolled back. Nonetheless, depending on the quality of the prior knowledge about locking patterns, the performance benefits would likely outweigh the additional overhead introduced by occasional rollbacks and by the extra scheduling optimizer component.

Future Efficiency Optimization Goals

Prototype Implementation for Vertical Scaling

While our prototypical implementation and analysis of operating systems and virtualization solutions supporting vertical scaling during runtime is a big step towards systems capable of saving immense operating costs, we have of course not yet implemented a readily usable framework bringing this vision into reality. The next step regarding this topic could therefore aim at building a working prototype that demonstrates our results in an actual live system, scaling hardware up and down while load on the system changes. This could be especially useful for cloud providers looking to utilize the unique FT guarantees that SMR could offer their clients.

Combination with Previous Work

In addition, as we mentioned briefly in Section 10.1.2, our approach is orthogonal to most, if not all the other existing efficiency optimization approaches from previous work, and is thusly primed for combination with any of these solutions (such as utilizing trusted subsystems to reduce the number of replicas required for BFT).

Further Optimizing Performance Optimizations

Of course a wide array of options presents itself also in regard to further researching self-optimizing SMR systems.

Combination with Orthogonal Solutions

Similar in nature to the orthogonality of our efficiency optimization approach, our general lock level-based parallelization readily lends itself to combination with many of the request-level approaches presented in Section 14.1.1. Equipping, for example, Eve ([54]) with multiple independent UDS instances each executing a batch could introduce additional parallelism to the system without adding any additional uncertainty.

Expanding on the Concept of ByTI

Moving towards ByTI as a mechanism that enables deterministic measurements, there is some work left after our analysis in Chapter 13, to truly vet the core ByTI algorithm and make sure that there are no hidden logical errors, similar to what we discussed above regarding UDS and model-checking approaches. In addition, so far we have only counted threads within one ByTI to estimate throughput or request arrival rates. It would be interesting to see how far this concept could be driven, i.e., whether the detailed counting of individual locks taken within a ByTI can provide further insight that could be used by an optimizer component.

Intelligent Optimizer Components

This perfectly segues into the remaining work that is to be done regarding the actual component in our system that decides on reconfigurations in order to optimize the system. We have only properly demonstrated a simple optimization algorithm, and could not fully finish the generalized approach using RL agents. Future work could therefore either continue trying to train agents based on our training framework, or it could try to take another optimization approach altogether.

17.3 OUTLOOK

After over one hundred and seventy pages of discussions about our concrete scientific goals, fact-based approaches, and solutions to specific problems, we would like to finish the thesis with a few opinions on the current state-of-the-art in the field of deterministically multithreaded systems and hopefully leave the reader with a content feeling after reaching the end of this work.

SMR as a Standard Approach

While the field of state machine-replicated systems has a relatively active, albeit small, research community behind it, we feel that the topic has not yet reached the widespread attention it deserves in a world where every other day a new story breaks about failures in systems costing our society time, money, or in extreme cases sometimes even lives. SMR can be a wonderful technique to securing systems against a wider variety of faults and even attacks, even though it comes at high resource and development costs. The research shown in this work has hopefully contributed significantly to the efforts of making sure that the performance of SMR systems is not too far off of regular, non-replicated systems. However, no amount of performance optimization takes away the fact that there are currently no publically usable, stable, tested, and ideally open-source SMR libraries except for BFT-SMaRt, i.e., the one research library everyone in this field tends to use if they are not building their own research prototypes. We would therefore love to see greater public attention on this interesting topic and the inevitable influx of developers that would try to improve these current shortcomings of the field of SMR.

Better Framework Support

Validity of All Approaches

As one of our final remarks, we would also briefly like to comment on request- vs lock-level optimization and the apparent rift in the research community between the groups following those two approaches. Even though the groups behind each of these approaches are—after all—trying to achieve the same goal, i.e., better performance and usability of SMR systems to promote their widespread usage, a lack of regard for other solutions and ideas can some-

times be clearly felt. This becomes especially apparent when reading papers in the context of SMR—even survey papers meant to give a broad overview over the field—about optimization approaches based on multithreading, which tend to completely ignore the existence of an entire class of approaches that exists next to their own ones. If we could wish for better collaboration between these groups in the spirit of scientific progress and innovation, we would.

Final Summary of Contributions

To summarize our contributions in a few last, long sentences, we have contributed to the introduction of a novel, runtime-reconfigurable deterministic scheduling algorithm, which we implemented, tested, and integrated into a fully working SMR system prototype. This was followed by the demonstration of significant cost savings potentials in virtualized SMR systems, that could be unlocked by combining deterministic multithreading with vertically scalable hardware resources. Finally, we significantly furthered the capabilities of SMR systems to self-optimize in regard to performance by introducing a deterministic BFT metric on which reconfiguration decisions can be based during runtime, which we used to implement two separate optimization approaches to demonstrate (i) considerable performance upsides for a variety of application profiles and (ii) the viability of RL as generalizable method for equipping SMR systems with intelligence that can handle multiple configuration parameters and optimization goals.

With our truly final words, we would like to express our gratitude for the opportunity to pursue a PhD in the context of these interesting topics, and hope that this thesis was able to inspire its readers with at least a tiny amount of love for this rather niche field of research.

BIBLIOGRAPHY

PEER-REVIEWED PUBLICATIONS BY THE AUTHOR

[1] C. Berger, P. Eichhammer, H. P. Reiser, J. Domaschka, F. J. Hauck, and G. Habiger. "A Survey on Resilience in the IoT: Taxonomy, Classification, and Discussion of Resilience Mechanisms." In: *ACM Comput. Surv.* 54.7 (Sept. 2021). ISSN: 0360-0300. DOI: 10.1145/3462513 (cit. on p. 151).

[2] J. Domaschka, C. Berger, H. P. Reiser, P. Eichhammer, F. Griesinger, J. Pietron, M. Tichy, F. J. Hauck, and G. Habiger. "SORRIR: A Resilient Self-organizing Middleware for IoT Applications [Position Paper]." In: *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things - M4IoT '19*. Davis, CA, USA: ACM Press, 2019, pp. 13–16. ISBN: 978-1-4503-7028-8. DOI: 10.1145/3366610.3368098 (cit. on p. 151).

[3] G. Habiger, F. J. Hauck, J. Köstler, and H. P. Reiser. "Resource-Efficient State-Machine Replication with Multithreading and Vertical Scaling." In: *2018 14th European Dependable Computing Conference (EDCC)*. Sept. 2018, pp. 87–94. DOI: 10.1109/EDCC.2018.00024 (cit. on pp. 83, 87, 176).

[4] G. Habiger, F. J. Hauck, H. P. Reiser, and J. Köstler. "Self-optimising Application-agnostic Multithreading for Replicated State Machines." In: *2020 International Symposium on Reliable Distributed Systems (SRDS)*. 2020, pp. 165–174. DOI: 10.1109/SRDS51746.2020.00024 (cit. on pp. 133, 172, 177).

[5] F. J. Hauck, G. Habiger, and J. Domaschka. "UDS: A Novel and Flexible Scheduling Algorithm for Deterministic Multithreading." In: *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2016, pp. 177–186. DOI: 10.1109/SRDS.2016.030 (cit. on pp. 38, 41, 47, 74, 176).

[6] J. Köstler, H. P. Reiser, G. Habiger, and F. J. Hauck. "SmartStream: Towards Efficient Byzantine Resilient Data Streaming through Speculation and Sharding." In: *ACM SIGAPP Applied Computing Review* 21.3 (Sept. 2021), pp. 19–32. ISSN: 1559-6915, 1931-0161. DOI: 10.1145/3493499.3493501 (cit. on p. 89).

NON-PEER-REVIEWED PUBLICATIONS BY THE AUTHOR

[7] P. Eichhammer, C. Berger, H. P. Reiser, J. Domaschka, F. J. Hauck, G. Habiger, F. Griesinger, and J. Pietron. *Towards a Robust, Self-Organizing IoT Platform for Secure and Dependable Service Execution*. Tech. rep. 2019. DOI: 10.18420/FBSYS2019-03 (cit. on p. 151).

[8] G. Habiger and F. J. Hauck. *Systems Support For Efficient State-Machine Replication*. Tech. rep. 2019. DOI: 10.18420/FBSYS2019-04.

[9] G. Habiger, F. J. Hauck, J. Köstler, and H. Reiser. *Vertikale Skalierung für aktiv replizierte Dienste in Cloud-Infrastrukturen*. Tech. rep. Institute of Distributed Systems, Ulm University, 2016.

PEER-REVIEWED REFERENCES

[10] M. K. Aguilera, W. Chen, and S. Toueg. "Failure Detection and Consensus in the Crash-Recovery Model." In: *Distributed Computing* 13.2 (Apr. 2000), pp. 99–125. ISSN: 0178-2770, 1432-0452. DOI: 10.1007/s004460050070 (cit. on p. 10).

[11] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. "MillWheel: Fault-tolerant Stream Processing at Internet Scale." In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1033–1044. ISSN: 2150-8097. DOI: 10.14778/2536222.2536229 (cit. on p. 114).

[12] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone. "Reconfiguring Parallel State Machine Replication." In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. Sept. 2017, pp. 104–113. DOI: 10.1109/SRDS.2017.23 (cit. on p. 134).

[13] E. Alchieri, F. Dotti, P. Marandi, O. Mendizabal, and F. Pedone. "Boosting State Machine Replication with Concurrent Execution." In: *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. Foz do Iguaçu, Brazil: IEEE, Oct. 2018, pp. 77–86. ISBN: 978-1-5386-8489-4. DOI: 10.1109/LADC.2018.00018 (cit. on p. 134).

[14] E. Alchieri, F. Dotti, and F. Pedone. "Early Scheduling in Parallel State Machine Replication." In: *Proceedings of the ACM Symposium on Cloud Computing*. Carlsbad CA USA: ACM, Oct. 2018, pp. 82–94. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267825 (cit. on p. 134).

[15] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2 (cit. on pp. 9, 10).

[16] P. Barret, A. Hilborne, P. Bond, D. Seaton, P. Verissimo, L. Rodrigues, and N. Speirs. "The Delta-4 Extra Performance Architecture (XPA)." In: *[1990] Digest of Papers. Fault-tolerant Computing: 20th International Symposium*. 1990, pp. 481–488. DOI: 10.1109/FTCS.1990.89386 (cit. on p. 14).

[17] C. Basile, Z. Kalbarczyk, and R. Iyer. "A preemptive deterministic scheduling algorithm for multithreaded replicas." In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* 2003, pp. 149–158. DOI: 10.1109/DSN.2003.1209926 (cit. on pp. 21, 39, 56).

[18] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. "Loose synchronization of multithreaded replicas." In: *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* 2002, pp. 250–255 (cit. on pp. 20, 139).

[19] C. Basile, Z. Kalbarczyk, and R. K. Iyer. "Active Replication of Multithreaded Applications." In: *IEEE Transactions on Parallel and Distributed Systems* 17.5 (May 2006), pp. 448–465 (cit. on pp. 56, 136).

[20] E. Batista, E. Alchieri, F. Dotti, and F. Pedone. "Early Scheduling on Steroids: Boosting Parallel State Machine Replication." In: *Journal of Parallel and Distributed Computing* 163 (May 2022), pp. 269–282. ISSN: 07437315. DOI: 10.1016/j.jpdc.2022.02.001 (cit. on p. 134).

[21] J. Behl, T. Distler, and R. Kapitza. "Consensus-Oriented Parallelization: How to Earn Your First Million." In: *Proc. of the 16th Int. Middlew. Conf.* ACM, 2015, pp. 173–184 (cit. on p. 136).

[22] J. Behl, T. Distler, and R. Kapitza. "Hybrids on Steroids: SGX-based High Performance BFT." In: *Proc. of the 12th Europ. Conf. on Comp. Sys. (EuroSys)*. ACM. 2017, pp. 222–237 (cit. on p. 89).

[23] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. "CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution." In: *SIGARCH Comput. Archit. News* 38.1 (Mar. 2010), pp. 53–64. ISSN: 0163-5964. DOI: 10.1145/1735970.1736029 (cit. on p. 22).

[24] C. Berger, H. P. Reiser, and A. Bessani. "Making Reads in BFT State Machine Replication Fast, Linearizable, and Live." In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. Chicago, IL, USA: IEEE, Sept. 2021, pp. 1–12. ISBN: 978-1-66543-819-3. DOI: 10.1109/SRDS53918.2021.00010 (cit. on p. 13).

[25] A. Bessani, J. Sousa, and E. E. P. Alchieri. "State Machine Replication for the Masses with BFT-SMART." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2014, pp. 355–362. DOI: 10.1109/DSN.2014.43 (cit. on pp. 27, 90).

[26] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery." In: *ACM Trans. Comp. Sys.* 20.4 (Nov. 2002), pp. 398–461 (cit. on p. 84).

[27] T. D. Chandra and S. Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems." In: *Journal of the ACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/226643.226647 (cit. on p. 13).

[28] J. Coffman E. G., M. Garey, and D. Johnson. "Approximation Algorithms for Bin Packing: A Survey." In: *Approximation algorithms for NP-hard problems* (1996), pp. 46–93 (cit. on p. 25).

[29] V. V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. Bessani. "FITCH: Supporting Adaptive Replicated Services in the Cloud." In: *Distributed Applications and Interoperable Systems*. Ed. by J. Dowling and F. Taïani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 15–28. ISBN: 978-3-642-38541-4 (cit. on pp. 84, 89).

[30] M. Correia, N. F. Neves, and P. Verissimo. "BFT-TO: Intrusion Tolerance with Less Replicas." In: *The Computer Journal* 56.6 (June 2013), pp. 693–715. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/bxs148 (cit. on p. 89).

[31] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. "Paxos Made Transparent." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: ACM, 2015, pp. 105–120. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815427 (cit. on p. 23).

[32] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. "Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 388–405. ISBN: 9781450323888. DOI: 10.1145/2517349.2522735 (cit. on p. 22).

[33] A. Daliot, D. Dolev, and H. Parnas. "Linear Time Byzantine Self-Stabilizing Clock Synchronization." In: *Proc. of the Conf. on Princ. of Distr. Sys. (OPODIS)*. LNCS 3144. Berlin, Heidelberg: Springer, 2004, pp. 7–19 (cit. on p. 114).

[34] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. "DMP: Deterministic Shared Memory Multiprocessing." In: *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '09*. Washington, DC, USA: ACM Press, 2009, p. 85. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508255 (cit. on p. 19).

[35] T. Distler, C. Cachin, and R. Kapitza. "Resource-Efficient Byzantine Fault Tolerance." In: *IEEE Trans. on Comp.* 65.9 (2016), pp. 2807–2819 (cit. on p. 89).

[36] T. Distler. "Byzantine Fault-tolerant State-machine Replication from a Systems Perspective." In: *ACM Computing Surveys* 54.1 (Apr. 2021), pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3436728 (cit. on pp. 27, 137).

[37] T. Distler, I. Popov, W. Schröder-Preikschat, H. P. Reiser, and R. Kapitza. "SPARE: Replicas on Hold." In: *Pro. of the Netw. and Distr. Sys. Sec. Symp. (NDSS)*. Internet Soc. 2011 (cit. on p. 89).

[38] J. Domaschka, T. Bestfleisch, F. J. Hauck, H. P. Reiser, and R. Kapitza. "Multithreading Strategies for Replicated Objects." In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. New York, NY, USA: Springer-Verlag New York, Inc., 2008, pp. 104–123. ISBN: 3-540-89855-7 (cit. on pp. 21, 31, 37, 44, 56, 176).

[39] E. N. (Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." In: *ACM Computing Surveys* 34.3 (Sept. 2002), pp. 375–408. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/568522.568525 (cit. on p. 19).

[40] C. A. Floudas and C. E. Gounaris. "A Review of Recent Advances in Global Optimization." In: *Journal of Global Optimization* 45.1 (Sept. 2009), pp. 3–38. ISSN: 0925-5001, 1573-2916. DOI: 10.1007/s10898-008-9332-8 (cit. on p. 25).

[41] X. Glorot, A. Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks." In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by G. Gordon, D. Dunson, and M. Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323 (cit. on p. 163).

[42] M. Grześ. "Reward Shaping in Episodic Reinforcement Learning." In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '17. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 565–573 (cit. on p. 164).

[43] S. Haggenmüller, R. C. Maron, A. Hekler, J. S. Utikal, C. Barata, R. L. Barnhill, H. Beltraminelli, C. Berking, B. Betz-Stablein, A. Blum, S. A. Braun, R. Carr, M. Combalia, M.-T. Fernandez-Figueras, G. Ferrara, S. Fraitag, L. E. French, F. F. Gellrich, K. Ghoreschi, M. Goebeler, et al. "Skin Cancer Classification via Convolutional Neural Networks: Systematic Review of Studies Involving Human Experts." In: *European Journal of Cancer* 156 (Oct. 2021), pp. 202–216. ISSN: 09598049. DOI: 10.1016/j.ejca.2021.06.049 (cit. on p. 152).

[44] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks." In: *Science* 313.5786 (July 2006), pp. 504–507. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1127647 (cit. on pp. 111, 155).

[45] S. C. Hoi, D. Sahoo, J. Lu, and P. Zhao. "Online Learning: A Comprehensive Survey." In: *Neurocomputing* 459 (Oct. 2021), pp. 249–289. ISSN: 09252312. DOI: 10.1016/j.neucom.2021.04.112 (cit. on p. 28).

[46] P. J. Huber. "Robust Estimation of a Location Parameter." In: *The Annals of Mathematical Statistics* 35.1 (Mar. 1964), pp. 73–101. ISSN: 0003-4851. DOI: 10.1214/aoms/1177703732 (cit. on p. 163).

[47] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-Scale Systems." In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11 (cit. on p. 134).

[48] L. Jehl and H. Meling. "Towards Byzantine Fault Tolerant Publish/Subscribe: A State Machine Approach." In: *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*. Farmington Pennsylvania: ACM, Nov. 2013, pp. 1–5. ISBN: 978-1-4503-2457-1. DOI: 10.1145/2524224.2524232 (cit. on p. 89).

[49] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. "Deterministic scheduling for transactional multithreaded replicas." In: *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*. 2000, pp. 164–173. DOI: 10.1109/RELDI.2000.885404 (cit. on p. 20).

[50] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey." In: *Journal of Artificial Intelligence Research* 4 (May 1996), pp. 237–285. ISSN: 1076-9757. DOI: 10.1613/jair.301 (cit. on p. 28).

[51] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. "Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation." In: *Proceedings of the 2nd Conference on Robot Learning*. Ed. by A. Billard, A. Dragan, J. Peters, and J. Morimoto. Vol. 87. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 651–673 (cit. on pp. 28, 110).

[52] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. "CheapBFT: Resource-Efficient Byzantine Fault Tolerance." In: *Proc. of the 7th ACM Europ. Conf. on Comp. Sys. (EuroSys)*. ACM. 2012, pp. 295–308 (cit. on p. 89).

[53] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler. "Storyboard: Optimistic Deterministic Multithreading." In: *Proc. of the 6th Worksh. on Hot Topics in Sys. Dep. (HotDep)*. 2010 (cit. on pp. 135, 139, 179).

[54] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. "All about Eve: Execute-Verify Replication for Multi-Core Servers." In: *Proc. of the 10th USENIX Symp. on Oper. Sys. Des. and Impl. (OSDI)*. 2012, pp. 237–250 (cit. on pp. 135, 179).

[55] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization." In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. 2015 (cit. on p. 163).

[56] R. Kotla and M. Dahlin. "High Throughput Byzantine Fault Tolerance." In: *Proc. of the Int. Conf. on Dep. Sys. and Netw (DSN)*. IEEE. 2004, pp. 575–584 (cit. on p. 134).

[57] L. Lamport. "Specifying Concurrent Systems with TLA+." In: *Calculational System Design* (Apr. 1999), pp. 183–247 (cit. on pp. 77, 178).

[58] L. Lamport. "The Part-Time Parliament." In: *ACM Transactions on Computer Systems* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/279227.279229 (cit. on p. 13).

[59] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359545.359563 (cit. on p. 14).

[60] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem." In: *ACM Transactions on Programming Languages and Systems* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176 (cit. on pp. 14, 15).

[61] T. Liu, C. Curtsinger, and E. D. Berger. "Dthreads: Efficient Deterministic Multithreading." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: ACM, 2011, pp. 327–336. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043587 (cit. on p. 22).

[62] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. "A Survey of Deep Neural Network Architectures and Their Applications." In: *Neurocomputing* 234 (Apr. 2017), pp. 11–26. ISSN: 09252312. DOI: 10.1016/j.neucom.2016.12.038 (cit. on p. 155).

[63] P. J. Marandi, C. E. Bezerra, and F. Pedone. "Rethinking State-Machine Replication for Parallelism." In: *Proc. of the 34th IEEE Int. Conf. on Distr. Comp. Sys. (ICDCS)*. IEEE. 2014, pp. 368–377 (cit. on p. 134).

[64] P. J. Marandi and F. Pedone. "Optimistic Parallel State-Machine Replication." In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. Nara, Japan: IEEE, Oct. 2014, pp. 57–66. ISBN: 978-1-4799-5584-8. DOI: 10.1109/SRDS.2014.25 (cit. on p. 134).

[65] R. Marler and J. Arora. "Survey of Multi-Objective Optimization Methods for Engineering." In: *Structural and Multidisciplinary Optimization* 26.6 (Apr. 2004), pp. 369–395. ISSN: 1615-147X, 1615-1488. DOI: 10.1007/s00158-003-0368-6 (cit. on p. 26).

[66] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. "Human-Level Control through Deep Reinforcement Learning." In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236 (cit. on pp. 28, 110, 155, 156).

[67] E. Mocanu, D. C. Mocanu, P. H. Nguyen, A. Liotta, M. E. Webber, M. Gibescu, and J. G. Slootweg. "On-Line Building Energy Optimization Using Deep Reinforcement Learning." In: *IEEE Transactions on Smart Grid* 10.4 (July 2019), pp. 3698–3708. ISSN: 1949-3053, 1949-3061. DOI: 10.1109/TSG.2018.2834219 (cit. on pp. 28, 110).

[68] M. Olszewski, J. Ansel, and S. Amarasinghe. "Kendo: Efficient Deterministic Multithreading in Software." In: *Proc. of the 14th Int. Conf. on Arch. Support for Progr. Lang. and Oper. Sys. (ASPLOS)*. ACM, 2009, pp. 97–108 (cit. on pp. 19, 21, 57).

[69] R. Padilha and F. Pedone. "Augustus: Scalable and Robust Storage for Cloud Applications." In: *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. Prague, Czech Republic: ACM Press, 2013, p. 99. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465362 (cit. on p. 89).

[70] H. Rahmanian and M. K. K. Warmuth. "Online Dynamic Programming." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017 (cit. on p. 28).

[71] H. P. Reiser, J. Domaschka, F. J. Hauck, R. Kapitza, and W. Schroder-Preikschat. "Consistent Replication of Multithreaded Distributed Objects." In: *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*. 2006, pp. 257–266. DOI: 10.1109/SRDS.2006.14 (cit. on pp. 21, 56).

[72] F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." In: *Acm Computing Surveys* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167 (cit. on pp. 14, 134).

[73] W. S. Shin and A. Ravindran. "Interactive Multiple Objective Optimization: Survey I—Continuous Case." In: *Computers & Operations Research* 18.1 (Jan. 1991), pp. 97–114. ISSN: 03050548. DOI: 10.1016/0305-0548(91)90046-T (cit. on p. 26).

[74] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. "Mastering the Game of Go without Human Knowledge." In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature24270 (cit. on pp. 28, 110).

[75] D. Skeen and M. Stonebraker. "A Formal Model of Crash Recovery in a Distributed System." In: *IEEE Transactions on Software Engineering* SE-9.3 (1983), pp. 219–228. DOI: 10.1109/TSE.1983.236608 (cit. on p. 10).

[76] S. Sun, Z. Cao, H. Zhu, and J. Zhao. "A Survey of Optimization Methods From a Machine Learning Perspective." In: *IEEE Transactions on Cybernetics* 50.8 (Aug. 2020), pp. 3668–3681. ISSN: 2168-2267, 2168-2275. DOI: 10.1109/TCYB.2019.2950779 (cit. on pp. 25, 154).

[77] H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering. "A Theoretical and Empirical Analysis of Expected Sarsa." In: *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*. Nashville, TN, USA: IEEE, Mar. 2009, pp. 177–184. ISBN: 978-1-4244-2761-1. DOI: 10.1109/ADPRL.2009.4927542 (cit. on p. 152).

[78] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. "EBAWA: Efficient Byzantine Agreement for Wide-Area Networks." In: *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. San Jose, CA, USA: IEEE, Nov. 2010, pp. 10–19. ISBN: 978-1-4244-9091-2. DOI: 10.1109/HASE.2010.19 (cit. on p. 89).

[79] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, et al. "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning." In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-019-1724-z (cit. on pp. 28, 110).

[80] C. J. C. H. Watkins and P. Dayan. "Q-Learning." In: *Machine Learning* 8.3-4 (May 1992), pp. 279–292. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00992698 (cit. on pp. 154, 155).

[81] T. Winter and U. Zimmermann. "Discrete Online and Real-Time Optimization." In: *IFIP World Computer Congress on Fundamentals - Foundations of Computer Science*. Österreichische Computer Gesellschaft, 1998, pp. 31–48. ISBN: 3-85403-117-3 (cit. on p. 26).

[82] L. Yang and A. Shami. "On Hyperparameter Optimization of Machine Learning Algorithms: Theory and Practice." In: *Neurocomputing* 415 (Nov. 2020), pp. 295–316. ISSN: 09252312. DOI: 10.1016/j.neucom.2020.07.061 (cit. on p. 161).

[83] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. "Separating Agreement from Execution for Byzantine Fault Tolerant Services." In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 253–267. ISSN: 0163-5980. DOI: 10.1145/1165389.945470 (cit. on p. 89).

OTHER REFERENCES

[84] Amazon Web Services, Inc. *EC2 On-Demand Instance Pricing*. https://aws.amazon.com/ec2/pricing/on-demand/. 2022 (cit. on p. 98).

[85] Apple Computer Vision Machine Learning Team. *An On-device Deep Neural Network for Face Detection*. https://machinelearning.apple.com/research/face-detection. 2017 (cit. on p. 152).

[86] N. Ascheuer, M. Grötschel, S. O. Krumke, and J. Rambau. "Combinatorial Online Optimization." In: *Operations Research Proceedings 1998*. Ed. by P. Kall and H.-J. Lüthi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 21–37. ISBN: 978-3-540-65381-3. DOI: 10.1007/978-3-642-58409-1_2 (cit. on p. 26).

[87] R. Bellman. *Dynamic Programming*. First. Princeton, NJ, USA: Princeton University Press, 1957 (cit. on p. 154).

[88] D. P. Bertsekas. *Reinforcement Learning and Optimal Control*. 2nd printing (includes editorial revisions). Belmont, Massachusetts: Athena Scientific, 2019. ISBN: 978-1-886529-39-7 (cit. on p. 28).

[89] E. L. Bosworth. *The Power Wall*. http://www.edwardbosworth.com/My5155_Slides/Chapter01/ThePowerWall.htm. 2010 (cit. on p. 17).

[90] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004 (cit. on p. 25).

[91] I. Cutress and J. De Gelas. *Sizing Up Servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 - The Server CPU Battle of the Decade?* https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc-7000-cpu-battle-of-the-decade/2. 2017 (cit. on p. 93).

[92] J. Domaschka. "A Comprehensive Approach to Transparent and Flexible Replication of Java Services and Applications." PhD thesis. https://oparu.uni-ulm.de/xmlui/handle/123456789/2512: Ulm University, June 2013 (cit. on p. 138).

[93] M. Feurer and F. Hutter. "Hyperparameter Optimization." In: *Automated Machine Learning*. Springer, Cham, 2019, pp. 3–33 (cit. on p. 161).

[94] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis. Irvine: University of California, 2000 (cit. on p. 11).

[95] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. *Soft Actor-Critic Algorithms and Applications*. 2018. DOI: 10.48550/ARXIV.1812.05905. eprint: arXiv:1812.05905 (cit. on pp. 28, 110).

[96] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept." In: *The Origin of Concurrent Programming*. Ed. by P. B. Hansen. New York, NY: Springer New York, 1974, pp. 272–294. ISBN: 978-1-4419-2986-0. DOI: 10.1007/978-1-4757-3472-0_10 (cit. on pp. 21, 178).

[97] A. Karpathy. *Yes you should understand backprop*. https://karpathy.medium.com/yes-you-should-understand-backprop-e2f06eab496b. 2017 (cit. on p. 166).

[98] L. Lamport. *Distribution*. https://lamport.azurewebsites.net/pubs/distributed-system.txt. 1987 (cit. on p. 9).

[99] A. D. Laud. "Theory and Application of Reward Shaping in Reinforcement Learning." PhD thesis. USA: University of Illinois at Urbana-Champaign, 2004 (cit. on p. 164).

[100] Y. Li. *Deep Reinforcement Learning*. 2018. DOI: 10.48550/ARXIV.1810.06339. eprint: arXiv:1810.06339 (cit. on p. 29).

[101] M. Loreau, S. Naeem, and P. Inchausti. *Biodiversity and ecosystem functioning: synthesis and perspectives*. Oxford University Press on Demand, 2002 (cit. on p. 3).

[102] I. H. Osman and J. P. Kelly. "Meta-Heuristics: An Overview." In: *Meta-Heuristics*. Ed. by I. H. Osman and J. P. Kelly. Boston, MA: Springer US, 1996, pp. 1–21. ISBN: 978-1-4612-8587-8. DOI: 10.1007/978-1-4613-1361-8_1 (cit. on p. 25).

[103] B. Palma. "Monitoring Byzantine Fault Tolerant Systems to Support Dynamic Adaptation." Accessed: 2018/04/06. MA thesis. Instituto Superior Técnico, Universidade de Lisboa, 2017 (cit. on p. 113).

[104] M. L. Puterman. "Chapter 8 Markov Decision Processes." In: *Handbooks in Operations Research and Management Science*. Vol. 2. Elsevier, 1990, pp. 331–434. ISBN: 978-0-444-87473-3. DOI: 10.1016/S0927-0507(05)80172-0 (cit. on p. 110).

[105] Risk Steering Committee. *DHS Risk Lexicon 2010 Edition*. https://www.dhs.gov/xlibrary/assets/dhs-risk-lexicon-2010.pdf. 2010 (cit. on p. 3).

[106] K. Rupp. *48 Years of Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. 2020. DOI: 10.5281/zenodo.3947824 (cit. on p. 17).

[107] F. M. R. Sabino. "ByTAM: A Byzantine Fault Tolerant Adaptation Manager." MA thesis. Instituto Superior Técnico, Universidade de Lisboa, 2016 (cit. on p. 113).

[108] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. 3rd ed. distributed-systems.net: Maarten van Steen, 2017 (cit. on p. 10).

[109] TensorFlow API Documentation. *tf.config.experimental.enable_op_determinism*. https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism. 2022 (cit. on p. 165).

[110] H. Weatherspoon and J. D. Kubiatowicz. "Erasure Coding Vs. Replication: A Quantitative Comparison." In: *Peer-to-Peer Systems*. Ed. by G. Goos, J. Hartmanis, J. van Leeuwen, P. Druschel, F. Kaashoek, and A. Rowstron. Vol. 2429. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 328–337. ISBN: 978-3-540-44179-3. DOI: 10.1007/3-540-45748-8_31 (cit. on p. 12).

SUPERVISED STUDENT THESES AND STUDENT PROJECTS

[111] M. Benz. "Consensus Replacement in a Modular State-Machine Replication Framework: Trial and Evaluation." Master's Thesis. Institute of Distributed Systems, Ulm University, 2021.

[112] M. Benz. "Enabling Snapshotting in Multithreaded BFT-SMaRt." Master's Project. Institute of Distributed Systems, Ulm University, 2018.

[113] M. Benz. "Modular State Machine Replication." Master's Thesis. Institute of Distributed Systems, Ulm University, 2019.

[114] P. Butz. "Implementation, Deployment and Evaluation of UDS." Master's Thesis. Institute of Distributed Systems, Ulm University, 2017.

[115] U. Eser. "Benchmarking BFT-SMaRt with YCSB." Master's Project. Institute of Distributed Systems, Ulm University, 2021.

[116] O. Finnendahl. "Re-implementing etcd in Java." Master's Project. Institute of Distributed Systems, Ulm University, 2018.

[117] M. Jäckle and C. Vogel. "Provisioning, Monitoring and Snapshotting of BFT-SMaRt." Master's Project. Institute of Distributed Systems, Ulm University, 2017.

[118] M. Kempfle. "Integration of etcd4j and BFT-SMaRt Parallel." Master's Project. Institute of Distributed Systems, Ulm University, 2010.

[119] A. Knittel. "Implementation of asynchronous request handling in BFT-SMaRt." Master's Project. Institute of Distributed Systems, Ulm University, 2016.

[120] T. Nguyen. "Parallelizing a Java Re-implementation of etcd." Bachelor's Thesis. Institute of Distributed Systems, Ulm University, 2018.

VI

APPENDIX

UDS SIMULATION DETAILS

1 FULL CONFIGURATION FILE FORMAT

```
# UDS Model
# Simulation time
TIME_UNTIL_MODEL_STOPS = 100000
GLOBAL_SEED = 19884212691337420

# Number of primaries in one round
NUM_PRIMARIES = 4
NUM_PROCESSORS = 2

# Number of steps per round per primary
NUM_ROUNDSTEPS = 2

# Defining the order of the steps in the total order (supported: ROUND_ROBIN, ALL_AT_ONCE, RANDOM)
# ROUND_ROBIN: [1,2,3,4,1,2,3,4,...]
# ALL_AT_ONCE: [1,1,2,2,3,3,4,4,...]
# RANDOM: (as the name implies)
TOTAL_ORDER_SCHEDULING = ROUND_ROBIN

# System scheduler
# Time a thread has on a core until it has to yield
PROCESSOR_RUN_TIME = 5
# Timeunit of the simulation
SYSTEM_TIME_UNIT = MILLISECONDS

# Generator
# Number of initial threads at the beginning of the simulation
NUM_REQUEST_AT_START = 1
# Threads generated after the initial threads
NUM_REQUEST_AFTER_START = 249
# Choose the generator: NORMAL (using the definitions given here to generate request),
# or MAXTESTING[1|2|3] (automatically trying to find the maximum of requests per second the system can handle)
GENERATOR_TYPE = NORMAL
# Distribution can be EXP, NORMAL or UNIFORM (not used by MAXTESTING generators)
GENERATOR_DISTRIBUTION = NORMAL
# Parameters for thread generation distribution, e.g., mean and standard deviation for normal distributions
# 3 parameters required (if the distribution only needs 2, the first 2 are picked; the last can be set to -1.0)
GENERATOR_DISTRIBUTION_PARAMETERS = 10.0; 2.0; -1.0
# Size of request batches generated by the thread generator
GENERATOR_BATCH = 3

# Requests
# Nameprefix of the requests (used for mapping the following parameters to each request type)
```

```
43      GENERALIZED_REQUEST = REQUEST_A; REQUEST_B
44
45      # Probability that a request of the given type is generated in
            the ThreadGenerator (must sum up to 1)
46      #[Nameprefix]_PROBABILITY = X
47      # Name shown in the traces and debug outputs
48      #[Nameprefix]_VISUAL_NAME = ReqA
49      # Number of critical actions the request is using
50      #[Nameprefix]_STEPS = 2
51      # ID of the mutexes the request needs (use ";" as separation and
             0.0 if no lock is needed; 0.1 for IO)
52      # Mutex IDs have to be full Integers (e.g. 1.0, 2.0, 19.0)
53      # Each lock has to appear two times: positive will lock,
            negative will unlock.
54      # The unlock can be in another step. Multiple unlocks can be in
            one step, but one lock at most.
55      # The hold time based on the distribution will be used after the
             first lock in the list
56      # or at the end if there is no lock (after all unlocks)
57      #[Nameprefix]_LOCK = X; Y; -Y, -X
58      # Distribution type of the steps
59      # (time the process is simulating work per step; UNIFORM and
            NORMAL is supported; ";" as separator)
60      #[Nameprefix]_DISTRIBUTION = NORMAL; UNIFORM
61      # Parameters of the distribution of the given steps
62      # (";" as separator of the steps and "," as separator of the
            parameters within a step)
63      #[Nameprefix]_PARAMETERS = 5.0, 1.0; 8.0, 2.0
64
65
66      # Request A
67      REQUEST_A_PROBABILITY = 0.6
68      REQUEST_A_VISUAL_NAME = ReqA
69      REQUEST_A_STEPS = 4
70      REQUEST_A_LOCK = 0; 1; 2, -2, -1; 0
71      REQUEST_A_DISTRIBUTION = NORMAL; NORMAL; NORMAL; NORMAL
72      REQUEST_A_PARAMETERS = 9.0, 1.0; 8.0, 1.0; 7.0, 0.3; 9.0, 0.8
73
74      # Request B
75      REQUEST_B_PROBABILITY = 0.4
76      REQUEST_B_VISUAL_NAME = ReqB
77      REQUEST_B_STEPS = 3
78      REQUEST_B_LOCK = 0; 2, -2; 0
79      REQUEST_B_DISTRIBUTION = NORMAL; UNIFORM; NORMAL
80      REQUEST_B_PARAMETERS = 8.0, 2.0; 7.5, 9.8; 6.0, 4.0
```

2 JSON-FORMATTED OBSERVATIONS

The following listing shows a full example of one observation as it was sent by the Java Environment in the version that is stepped using ByTI. Note that (i) all values except the reward are normalized in the Python environment before being given to the agent, and (ii) the `avgActiveClients` values are included for logging and debugging purposes, but are not in fact given to the agent, since they constitute indeterministic knowledge.


```
 {
"reward":0.090183673469387761,
"observation":[
   {
      "schedulerId":"UDScheduler #0",
      "threadsTerminated":489,
      "roundsFinished":99,
      "totalStepsConsumed":724,
      "estimatedThroughPutPerSec":4890.0,
      "confPrim":5,
      "confSteps":2,
      "avgActiveClients":6.0,
      "roundsSinceLastConfigChange":275
   },
   {
      "schedulerId":"UDScheduler #0",
      "threadsTerminated":489,
      "roundsFinished":99,
      "totalStepsConsumed":723,
      "estimatedThroughPutPerSec":4890.0,
      "confPrim":5,
      "confSteps":2,
      "avgActiveClients":6.0,
      "roundsSinceLastConfigChange":175
   },
   {
      "schedulerId":"UDScheduler #0",
      "threadsTerminated":368,
      "roundsFinished":75,
      "totalStepsConsumed":542,
      "estimatedThroughPutPerSec":3680.0,
      "confPrim":5,
      "confSteps":2,
      "avgActiveClients":6.0,
      "roundsSinceLastConfigChange":75
   }
]
}
```

The observations in the environment using scheduling rounds as time steps looked different, and can be seen in the following listing. These observations included information about several previous rounds, including the type and number of threads seen in different data structures in the environment. Request types were encoded using numbers from `[0.2, 0.4, 0.6, 0.8]`, where for example ByTI-requests were signified by `0.2`. Note that once again, indeterministic values like `previousRoundCPUTimeUsedNs` were not given to the agent and values were usually normalized, too.


```
 {
"reward":1.0,
"observation":[
   {
      "scheduler":"UDScheduler #0",
      "previousRoundPrimaries":[
         0.8,
         0.8,
         0.8,
         0.8,
         0.8
      ],
      "previousRoundStepsTaken":[
         2,
         2,
         2,
         2,
         2
      ],
      "previousRoundRoundsSeen":[
         1,
         1,
         1,
         1,
         1
      ],
      "threadsInSystem":[
         0.8,
         0.8,
         0.8,
         0.8,
         0.8,
         0.8
      ],
      "round":54,
      "previousRoundDurationNs":1483076,
      "previousRoundCPUTimeUsedNs":1351579,
      "previousRoundActiveSODLClients":5,
      "estimatedClients":5,
      "roundsSinceLastConfigChange":0,
      "currentConfig":{
         "primaryNumber":5,
         "steps":2,
```

```
            "maxThreads":16,
            "deterministic":true
         }
      },
      {
         "scheduler":"UDScheduler #0",
         "previousRoundPrimaries":[
            0.8,
            0.8,
            0.8,
            0.8,
            0.8
         ],
         "previousRoundStepsTaken":[
            2,
            2,
            2,
            2,
            2
         ],
         "previousRoundRoundsSeen":[
            1,
            1,
            1,
            1,
            1
         ],
         "threadsInSystem":[
            0.8,
            0.8,
            0.8,
            0.8,
            0.8,
            0.8
         ],
         "round":53,
         "previousRoundDurationNs":1786962,
         "previousRoundCPUTimeUsedNs":1373420,
         "previousRoundActiveSODLClients":5,
         "estimatedClients":5,
         "roundsSinceLastConfigChange":0,
         "currentConfig":{
            "primaryNumber":5,
            "steps":2,
            "maxThreads":16,
            "deterministic":true
         }
      },
      {
         "scheduler":"UDScheduler #0",
         "previousRoundPrimaries":[
            0.8,
```

```
            0.8,
            0.8,
            0.8,
            0.8
         ],
         "previousRoundStepsTaken":[
            2,
            2,
            2,
            2,
            2
         ],
         "previousRoundRoundsSeen":[
            1,
            1,
            1,
            1,
            1
         ],
         "threadsInSystem":[
            0.8,
            0.8,
            0.8,
            0.8,
            0.8,
            0.8
         ],
         "round":52,
         "previousRoundDurationNs":1893872,
         "previousRoundCPUTimeUsedNs":1481881,
         "previousRoundActiveSODLClients":5,
         "estimatedClients":5,
         "roundsSinceLastConfigChange":0,
         "currentConfig":{
            "primaryNumber":5,
            "steps":2,
            "maxThreads":16,
            "deterministic":true
         }
      }
   ]
}
```