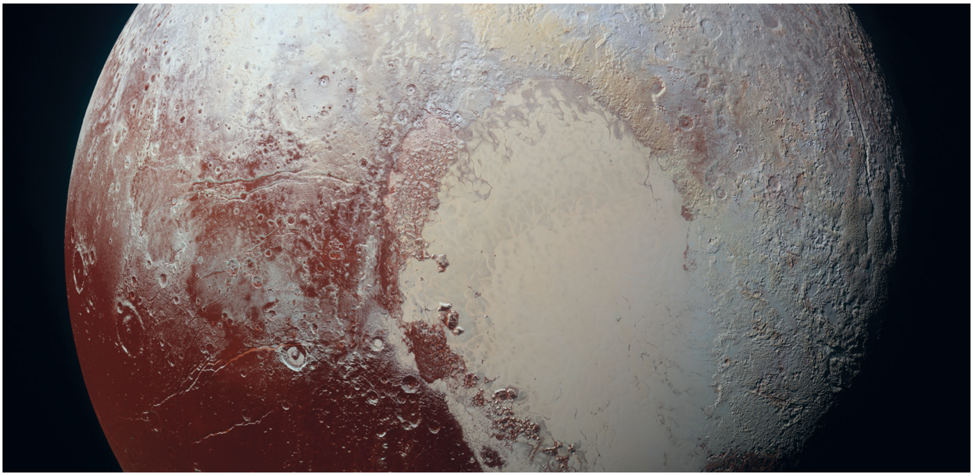
Foundations of computing Volume 11

Tim Jungnickel

On the Feasibility of Multi-Leader Replication in the Early Tiers



Universitätsverlag der TU Berlin

Technische Universität Berlin

Tim Jungnickel

On the Feasibility of Multi-Leader Replication in the Early Tiers

The scientific series *Foundations of computing* of the
Technische Universität Berlin is edited by:
Prof. Dr. Rolf Niedermeier,
Prof. Dr. Uwe Nestmann,
Prof. Dr. Stephan Kreutzer

Foundations of computing | 11

Tim Jungnickel

On the Feasibility of Multi-Leader Replication in the Early Tiers

Universitätsverlag der TU Berlin

Bibliographic information of the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at *http://dnb.dnb.de*.

Universitätsverlag der TU Berlin, 2018
http://verlag.tu-berlin.de

Fasanenstr. 88, 10623 Berlin
Tel.: +49 (0)30 314 76131 / Fax: -76133
E-Mail: *publikationen@ub.tu-berlin.de*

Zugl.: Berlin, Techn. Univ., Diss., 2018
Gutachter: Prof. Dr. Odej Kao
Gutachter: Prof. Dr. Uwe Nestmann
Gutachter: Prof. Dr. Gero Mühl (Universität Rostock)
Die Arbeit wurde am 30. Mai 2018 an der Fakultät IV unter Vorsitz von Prof. Dr. Florian Tschorsch erfolgreich verteidigt.

Print: docupoint GmbH
Layout/Typesetting: Tim Jungnickel

ISBN 978-3-7983-3001-6 (print)
ISBN 978-3-7983-3002-3 (online)

ISSN 2199-5249 (print)
ISSN 2199-5257 (online)

Published online on the institutional Repository of the Technische Universität Berlin:
DOI 10.14279/depositonce-7109
http://dx.doi.org/10.14279/depositonce-7109

ABSTRACT

In traditional service architectures that follow the service statelessness principle, the state is primarily held in the data tier. Here, service operators utilize tailored storage solutions to guarantee the required availability; even though failures can occur at any time. This centralized approach to store and process an application's state in the data tier implies that outages of the entire tier cannot be tolerated. An alternative approach, which is in focus of this thesis, is to decentralize the processing of state information and to use more stateful components in the early tiers.

The possibility to tolerate a temporary outage of an entire tier implies that the application's state can be manipulated by the remaining tiers without waiting for approval from the unavailable tier. This setup requires multi-leader replication, where every replica can accept writes and forwards the resulting changes to the other replicas.

This thesis explores the feasibility of using multi-leader replication to store and process state in a decentralized manner across multiple tiers. To this end, two replication mechanisms, namely Conflict-free Replicated Data Types and Operational Transformation, are under particular investigation. We use and extend both mechanisms to demonstrate that the aforementioned decentralization is worth considering when designing a service architecture.

The challenges that arise when following our approach go back to fundamental impossibility results in distributed systems research, i.e. the impossibility to achieve a fault-tolerant consensus mechanism in asynchronous systems and the inevitable trade-off between availability and consistency in the presence of failures. With this thesis, we contribute to close the exposed gaps of both results by providing usable alternatives for standard IT services. We exemplify the feasibility of our alternatives with a fully distributed IMAP service and a programming library that provides the necessary extension to utilize our approach in a variety of web-based applications.

All contributions of this thesis are based on both theory and practice. In particular, all extensions to the existing multi-leader replication mechanisms were proven to satisfy the necessary properties. Moreover, those extensions were also implemented as prototypical applications and evaluated against the corresponding de facto standard software from the industry.

ZUSAMMENFASSUNG

Basierend auf dem „service statelessness principle“ ist es üblich, Softwaredienste so zu entwerfen, dass der Zustand des Dienstes primär in einer gekapselten Datenschicht verarbeitet wird. Innerhalb der Datenschicht werden spezielle Lösungen verwendet, um die Verfügbarkeit der Daten sicherzustellen. Dieser zentralisierte Ansatz hat zur Folge, dass ein Ausfall oder eine temporäre Nichtverfügbarkeit der gesamten Datenschicht zwangsweise zur Nichtverfügbarkeit des gesamten Dienstes führt. Ein alternativer Ansatz, welcher in dieser Arbeit erforscht wird, ist die dezentralisierte Speicherung und Verarbeitung der Daten in den darüberliegenden Softwareschichten.

Um in diesem Ansatz einen Ausfall der gesamten Datenschicht zu kompensieren, ist es zwingend notwendig, dass die verbleibenden Schichten die eingehenden Anfragen ohne die Bestätigung durch die Datenschicht beantworten können. Hierfür wird eine Replikationsarchitektur benötigt, in der jedes Replikat die Anfragen direkt beantworten kann; die so genannte „multi-leader replication“.

In dieser Arbeit werden diese Replikationsarchitekturen verwendet, um den Zustand und die Daten eines Dienstes zu dezentralisieren und über mehrere Schichten zu replizieren. Hierbei werden zwei Mechanismen detaillierter betrachtet: „Conflict-free Replicated Data Types“ und „Operational Transformation“. Anschließend werden beide Mechanismen erweitert und hinsichtlich der Verwendbarkeit für den beschriebenen Ansatz geprüft. Als Ergebnis dieser Arbeit wird gezeigt, dass ein dezentralisierter Ansatz mit den vorgestellten Mechanismen in Betracht gezogen werden kann.

Die Herausforderungen, die bei der Anwendung dieses Ansatzes entstehen, basieren auf nachweislich unlösbaren Problemen aus der Forschung von Verteilten Systemen. Dazu gehört die Unlösbarkeit von Konsensus und die unausweichliche Abwägung zwischen Verfügbarkeit und Konsistenz in einem verteilten System mit Ausfällen. Diese Arbeit trägt dazu bei, die entstehenden Lücken, welche aus diesen fundamentalen Ergebnissen resultieren, zu schließen und die vorgeschlagenen Lösungen für reale IT Dienste anwendbar zu machen. Dieses wird anhand eines dezentralen IMAP Dienstes und einer Programmierbibliothek für Webanwendungen verdeutlicht.

Alle Bestandteile dieser Doktorarbeit verbinden Theorie und Praxis. Alle vorgeschlagenen Erweiterungen für bestehende Replikationssysteme werden in formalen Modellen verifiziert und prototypisch implementiert. Die Implementierungen werden außerdem mit vergleichbarer Standardsoftware, welche dem heutigen Stand der Technik entspricht, in praktischen Experimenten evaluiert.

ACKNOWLEDGMENTS

The Committee

I would like to express my sincere gratitude to my advisor Odej Kao for welcoming me in his research group in 2014 and giving me the opportunity to follow my ideas. On this way, Odej provided not only the resources, but also the necessary freedom and trust so that I was able to identify the questions that I wanted to answer in this thesis.

Furthermore, I would like to thank Uwe Nestmann for his guidance during, and before this dissertation project. The experience to be part of his research group as a student teaching assistant paved my way for this thesis. Thank you, Uwe, for providing an additional office, which I regularly used to keep in touch with you, the colleagues, and the research on the theory of distributed systems.

I also want to thank Gero Mühl for accepting to review this thesis and his valuable comments and feedback.

The Colleagues

I want to emphasize that my surroundings were highly influential in my daily work and that I am thankful for the enjoyable atmosphere that everyone of my colleagues created. I thank you all for sharing your knowledge and experience, and the countless discussions that we had during our lunch breaks and after our presentations.

CIT

Alexander Acker
Sören Becker
Jana Bechstein
Anne Grohnert
Anton Gulenko
Vincent Henning
Tobias Herb
Mareike Höger
Peter Janacik
Britta Kitanova
Andreas Kliem
Marc Körner
Björn Lohrmann
Sasho Nedelkoski
Thomas Renner
Florian Schmidt
Alexander Stanik
Lauritz Thamsen
Ilya Verbitskiy
Paul Völker
M. Wallschläger
Meike Zehlike

MTV

Benjamin Bisping
Paul Brodmann
David Karcher
Tobias Prehn
Kirstin Peters
Christina Rickmann
Christoph Wagner
A. Wilhelm-Weidner
Svea Wilkending

The Faculty

For almost 10 years I was able to benefit from the achievements of my faculty. This was made possible by the countless people that contributed to the development and growth, especially in teaching, research, and administration. Fortunately, I was able to make my own small contributions as well. I am thankful for the amount of appreciation that encountered to me, especially for my work in the faculty's boards. Among the many people that I met there, I particularly thank Hanna Wesner and my friends in the Freitagsrunde for their support and the exceptional work to make the university a better place.

The Students

I would like to thank the many students that worked with me during my research. In particular, I want to thank Ronny Bräunlich, Juan Cabello, Matthias Loibl, and Lennart Oldenburg for their inspirational minds and their contributions to this thesis. Many lines of code that were written for this thesis carry their signature for which I am indefinitely thankful.

The Family

Ultimately, I thank my family for their unquestionable love and support. My mother, Ina Weber, for raising me with the right values and the confidence that I succeed on the path that I chose. My father, Lutz Jungnickel, for his unconditional support and his counsel. Steffen Laubner, for the many skills that I have learned from him. My grandparents, who provided the necessary contrast to the life in the big city. And, most importantly, Maike and our kids Paul and Charlotte for their love and patience with me, especially during the most intense phases of my study.

Tim
Berlin, April 2018

This is for you,
Paul, Charlotte & Maike.

In loving memory of Werner Jungnickel.
1935 – 2017

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

Over the last years we have observed an ongoing centralization of the services we use on the Internet. This trend manifests in the increased popularity of services that aggregate the data of many users to deliver the necessary convenience to be successful. For example, the possibility to use a personal photo library on multiple devices like smartphones and tablets convinces users to upload their private data to a central instance like iCloud, Dropbox, or Google Drive. The same holds true for office applications or social networks.

Surprisingly, this trend can also be observed for services that are designed to run in a decentralized fashion, like the email system. In the original email system, everyone could participate in a network of mail servers and be responsible for the delivery and the reception of messages for their domain. Nowadays, only a minority of the people run their own mail server and huge mail providers are used instead. For example, two years ago Google's mail service GMail crossed the mark of 1 billion monthly active users [Mil16]. In the context of blockchains and the Bitcoin network we observe that many people buy their digital currency on centralized services like Coinbase that aggregate the wallets of users [Shi17]. In a sense, these centralized services can be seen as a bank, which is strange manifestation of the opposite of the original motivation of the Bitcoin network, i.e. to have a decentralized digital currency without centralized authorities.

The reason for this ongoing centralization is the convenience that is provided by centralized services. Decentralized services, however, are typically seen as difficult to maintain and prone to configuration errors or concurrency related bugs. On the other hand, centralized services require a certain confidence that the reliability expectations are fulfilled.

The trend of centralization can also be observed in the design and development of software services with respect to managing data and state information. As a guiding principle, i.e. the service statelessness principle [Erl05], processing state information is reduced to the necessary minimum. This typically results in application designs where data and state is held exclusively in the data tier, and the other tiers follow stateless designs. The data tier is in turn held at a single provider like Amazon Web Services or at the Google Cloud Platform where, consequently, convenience in maintainability is again traded against the required confidence in the reliability guarantees of those providers.

Unfortunately, the consequences of a failure of the single and centralized point where the state is processed can be severe. For example, an outage of Amazon's S3 system that happened in 2017 resulted in service disruption of major websites like GitHub, Slack, or Twitch for over four hours [Nic17]. During that time, no EC2 compute instances could be launched and other important storage systems like the Elastic Block Storage were unavailable [Ser17].

To this end, achieving more decentralization in application designs and thus more independence between the different tiers of an application's architecture is a desirable goal. The requirement to achieve this, is that designing and maintaining a decentralized system is as convenient as designing a centralized system. Therefore, an exploration and advancement of the existing decentralization mechanisms is necessary.

1.2 CHALLENGES AND PROBLEM STATEMENT

In today's commonly used architectures for applications of a scale that exceeds the capabilities of a single server, functionalities are encapsulated in independent software components which are in turn placed on multiple servers. One typically used alignment of these independent software components is the 3-tier architecture, which defines a hierarchical structure of *presentation*, *logic*, and *data* [Fow02].

While this approach to design an application is certainly beneficial, the requirement to operate such an application is that all tiers are working properly. A temporary interruption of any of the tiers would result in service disruption, or at least reduced functionality. In order to prevent this, certain measures are taken to increase the reliability of the application. We note that the critical part, which makes increasing reliability a nontrivial challenge, is in fact the application's state. Therefore, the commonly applied architectural approach is to process the application's state in the data tier only; making the components in the presentation and logic tier mostly stateless.

The stateful components in the data tier are in turn subject of special consideration to achieve the necessary *high availability*. The typical mechanisms that are applied to increase the reliability while achieving the needed performance are *replication* and *partitioning* [Kle16]. For example, databases like PostgreSQL or MySQL use single-leader replication to avoid relying on a single copy of data [Pos18; Ora18]. Other approaches allocate specialized hardware or duplicate the entire service (active replication), resulting in extraordinary high demands of resources [AT09; AT03].

We note that the underlying strategy is to avoid maintaining more state than necessary at any cost, which can be seen as a direct consequence of the service statelessness principle. In this thesis, however, we propose a different approach by purposely breaking this principle and allowing the maintenance of more state the the early tiers. Hence, in contrast to *avoiding* the problem of processing and storing more state than necessary, we are *addressing* the problem directly and propose the needed extensions to existing replication mechanisms in order to transfer our approach to real-life applications.

The major challenge behind our approach is that achieving more independent and autonomous tiers requires a more complex replication architecture, namely multi-leader replication. In this architecture, the state is replicated across multiple tiers, enabling components in the early tiers to respond to a client's request without waiting for data tier to process the state. The presence of conflicting updates is inevitable in this scenario, making conflict resolution mechanisms necessary for all kind of situations that occur with concurrent updates. Designing a complete and sound conflict resolution mechanism is a challenging and error-prone task that requires careful consideration.

The overarching problem is the manifestation of the CAP dilemma, which represents a fundamental and still unsolved problem in distributed systems research [Bre00; GL02]. By replicating the application state across multiple tiers, we effectively face the CAP consequences that designing an application that simultaneously guarantees high availability, consistency, and partition tolerance is impossible. The challenge in our approach is to mitigate this dilemma and to propose a practical solution for IT services.

In this thesis we aim to demonstrate the feasibility of storing and processing more state in the presentation and logic tier, and thus improving the reliability of applications. We use existing multi-leader replication mechanisms and extend them in a way that they can be applied in standard IT services. We carefully take the arising challenges both on a theoretical and practical level, i.e. all proposed extensions are verified and implemented.

To this end, we assert the following statement in this thesis: *In order to improve the reliability and performance of applications in a 3-tier architecture, replicating state across the presentation or logic tier is worth considering.*

1.3 OUTLINE OF THIS THESIS

We begin by providing the necessary background and the fundamental perspectives for this thesis in Chapter 2. This chapter includes an overview about the most important contributions in the field of distributed systems that are referenced in the later chapters. We look closer at the recent achievements in the field of software architectures, e.g. service-oriented architectures, microservices, layered-architectures, and the 3-tier architecture. Furthermore, we summarize the often used approaches *partitioning* and *replication* to scale-out applications and look closer at the CAP dilemma to understand the options for an application in presence of failures. Ultimately, we define the consistency models that are of particular interest in the rest of this thesis.

In Chapter 3 we present one of the two main explorations of this thesis: the opportunities and disadvantages of handling more state in the logic tier. Therefore, we present the necessary foundations of the used multi-leader replication mechanism, i.e. Conflict-free Replicated Data Types (CRDTs), before we exemplify our approach with the Internet Message Access Protocol (IMAP). As one of our main contributions, we present our proposal of an IMAP-CRDT and the associated verification of the necessary properties with the interactive theorem prover Isabelle/HOL. Moreover, we show the feasibility and applicability of our approach by presenting our research prototype *pluto* and an evaluation against the de facto standard IMAP server *Dovecot*. Subsequently, we discuss the related work and our contributions to the research community.

In Chapter 4 we present the second main exploration: handling more state in the presentation tier. Here, we summarize and use Operational Transformation (OT) as multi-leader replication mechanism. We present our generalization of OT by introducing a verified transformation function that enables simultaneous editing of JSON objects, which is highly relevant for modern web development. To demonstrate the applicability in the presentation tier, we present our prototypical application of a collaborative patient documentation system. Thereafter, we transfer the gained insights to a programming library, which we use to evaluate our extension in various collaborative editing scenarios against Google Docs and ShareDB, and confirm the limitations of OT.

We take the acquired insights from the previous chapters and discuss the transferability of our approach to other than the explored software services in Chapter 5. Since both of the used mechanisms for multi-leader replication have their advantages and disadvantages, we discuss the sweet spot and further explore possible application areas. Moreover, we discuss the implications and the possibilities that arise when further following our approach in a future work section.

Ultimately, we conclude and summarize this thesis in Chapter 6.

1.4 MAIN CONTRIBUTIONS AND PUBLICATIONS

The contributions to the scientific community can be summarized as follows:

1. We propose a novel approach for placing stateful software components with integrated multi-leader replication in a 3-tier architecture by breaking the widely applied service statelessness principle. The resulting decentralization yields new opportunities for deploying services in inter-cloud and hybrid-cloud setups up to planetary scale.

2. We present a new and verified Conflict-free Replicated Data Type that reflects the state of an IMAP server and supports all

write-commands from RFC 3501. With our datatype design we propose a novel approach to use CRDTs in standard IT services. Furthermore, with the implementation of our verification in Isabelle we contribute another example for a recently proposed CRDT verification framework to the Isabelle community.

3. We unveil the limitations of the currently used replication mechanism (*dsync*) in the de facto standard IMAP server *Dovecot*. The evaluation of our open-source prototype *pluto* includes a benchmark tool that generates write-intensive workloads, which can be used to stress test other IMAP servers, for example GMail or Microsoft Exchange. With our evaluation at planetary-scale we demonstrate how our approach can be used to significantly reduce the replication lag.

4. We contribute our extension of Operational Transformation to support simultaneous editing of JSON objects by introducing a mapping between the JSON structure and our verified transformation function for ordered n-ary trees. We further contribute an open-source example application that exemplifies the applicability of our extension in modern web development. With our evaluation we confirm the limitations of OT in real-time collaboration systems at large scale and compare our programming library to the existing open-source solution ShareDB.

5. With our extensions for two commonly used multi-leader replication mechanisms we further contribute to bridging the gap that is exposed by the CAP theorem, which represents an open and fundamental problem in distributed systems research, that has high influence to the IT industry.

We want to emphasize that all contributions of this thesis are a result of a scientific process that includes a theoretical analysis of the problem, an abstract solution, a formal verification of the necessary properties, a prototypical implementation, and a practical evaluation. Hence, we would like to add this interplay of theory and practice to solve a particular problem to the list of engineering-related contributions, together with all open-source prototypes that have been developed for this thesis.

Publications

Most parts of this thesis, and the corresponding contributions, have been published in the following list of peer-reviewed articles [JH16; JCR17; JO17; JB17; JOL17a] and technical reports [JH15; JOL17b]:

[JB17]	Tim Jungnickel and Ronny Bräunlich. 2017. formic: Building Collaborative Applications with Operational Transformation. In: *Conference on Distributed Applications and Interoperable Systems* (DAIS), 138–145.
[JCR17]	Tim Jungnickel, Juan Cabello, and Klemens Raile. 2017. HotPi: Open-Source Collaborative Patient Documentation. In: *ACM Conference on Computer-Supported Cooperative Work and Social Computing Companion* (CSCW), 219–222.
[JH15]	Tim Jungnickel and Tobias Herb. 2015. *TP1-valid Transformation Functions for Operations on ordered n-ary Trees.* *arxiv.org*
[JH16]	Tim Jungnickel and Tobias Herb. 2016. Simultaneous Editing of JSON Objects via Operational Transformation. In: *ACM Symposium on Applied Computing* (SAC), 812–815.
[JO17]	Tim Jungnickel and Lennart Oldenburg. 2017. pluto: The CRDT-Driven IMAP Server. In: *Workshop on Principles and Practice of Consistency for Distributed Data* (PaPoC), 1:1–1:5.
[JOL17a]	Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. 2017. Designing a Planetary-Scale IMAP Service with Conflict-free Replicated Data Types. In: *Conference on Principles of Distributed Systems* (OPODIS), 23:1–23:17.
[JOL17b]	Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. 2017. *The IMAP CmRDT.* Isabelle Archive of Formal Proofs. *isa-afp.org*

2 BACKGROUND

2.1 STATE IN SERVICE-ORIENTED ARCHITECTURES

At the time of writing this thesis, service-oriented architectures (SOA) were introduced more than 20 years ago by Schulze and Natiz in a research report from Gartner. Since then, it has been actively researched and various extensions and patterns emerged and ultimately SOA became the de facto standard architecture for enterprise applications. More recently, the term *microservices* has been introduced as a variant of SOA. Now, and almost 20 years after the introduction of SOA, architectures based on microservices are state-of-the-art and used in almost all major large-scale websites like Netflix, Amazon, or eBay [Fow14]. However, precisely defining the conceptual difference between microservice architectures and SOA is difficult, as microservices are often seen as a reinvention of the almost outdated SOA principles. In this section we introduce the needed definitions that have emerged from the past years of SOA related research. We want to emphasize that even though we base some of our contributions on the rather old SOA definitions, the concepts behind those definitions are more than ever relevant in modern IT architectures.

2.1.1 *Service Statelessness Principle*

Service-oriented architectures are typically defined as an architectural style that uses software *services* as the smallest building block of a bigger infrastructure. The services are usually seen as loosely coupled, coarse-grained, and autonomous [RGO12]. Services can interact with each other over *messages* at discoverable addresses called *endpoints*.

In the literature there is a distinction between *stateful* and *stateless* services [AT09]. A stateless service treats each request as an independent one, where the response is not related to any previous request. In contrast to that, a stateful service stores data beyond one request. This distinction will become important because both types of services have fundamentally different properties when it comes to scalability and fault tolerance.

In addition to the mentioned distinction between stateful and stateless services, Thomas Erl further defines the architectual principles of SOA [Erl05]. Among those principles is the *service statelessness principle*. According to this principle, services should minimize the usage of resources and limit itself to storing and processing state information only when it is absolutely necessary. We note that the distinction between stateful and stateless services is, in fact, a manifestation of the service statelessness principle, because following the idea of minimizing the processed state information leads to only two options: either delegating state management (stateless) or handle state explicitly (stateful).

2.1.2 *3-tier Architecture*

Apart from service-oriented architectures there is another architectural pattern called the *layered architecture* [Bus+96]. In the layered architecture, the software functionality is partitioned into horizontal subsystems that encapsulate certain features. One benefit of these encapsulations is that the functionality on one layer can evolve independently from other layers. For example, user interfaces tend to change at a higher rate than the application logic, therefore the layered structure allows to redesign the interface without modifying the logic.

The major challenge in a layered architecture is to define the right number of layers. While too few layers may result in inflexibility regarding the rate of change that is necessary, too many layers can fragment the architecture too much, which in turn results in unnecessary overhead and poor maintainability. To this end, there is a common approach to define three layers, i.e. the 3-layered architecture:

PRESENTATION LAYER: The presentation layer contains the user interface and all interface-related functionality. Typical functionality includes the invocation of requests to the other layers and the translation of the result of one request to a form that can be understood by the user.

APPLICATION LAYER: The application layer contains the logic of an application. This layer typically contains the functions that are responsible to compute a result for a request.

PERSISTENCE LAYER: The persistence layer typically contains all functionality to store and retrieve data. For example, this layer could include functions to search the data or reorganize the used data layout.

We note that the layered architecture can be seen as a logical separation of code or functionality. The presented definition makes no assumptions about how these layers should interact or how the layers could be placed on IT infrastructure. To this end, an opposing version of the layered architecture emerged: the *multitier architecture*.

The multitier architecture focuses on the physical components and where the code is deployed on [Fow02]. In combination with the layered architecture, the multitier architecture defines which layers should be placed on which group of physical machines. Consequently, it is possible to run a system with a 3-layered architecture on a single tier.

We note that there exist some confusion and misinterpretation of the term *tier*. In very strict definitions, the term tier refers to a single physical machine. We find that this strict definition is rather inconvenient. Following this logic, an application that can automatically scale to a higher number of nodes in case of an increased workload would result in an ever-changing-tier architecture and the separation into tiers would be pointless.

In this thesis we use the term tier as a group of physical machines that are responsible for a particular layer. Since we focus on systems of a certain scale, we neglect the possibility that multiple layers run on one tier. Consequently, there is no mismatch between the number of tiers and the number of layers in our definition[1].

In accordance with the 3-layered architecture is the 3-tier architecture. In the 3-tier architecture, the tiers can be interpreted as physical machines that are connected over a network to exchange messages. The three tiers are typically called *presentation* tier, *logic*

1 This is actually a very common interpretation. Some definitions go even further and make no difference between the terms *tier* and *layer*, because nowadays the separation of functionality on different nodes can be easily achieved with virtualization and containerized applications.

tier, and *data* tier. The functionality that is typically deployed in those tiers can be derived from the above-presented definitions of the three layers.

State in a 3-tier Architecture

We note that this encapsulation of functionality in smaller building blocks that are communicating over messages is closely related to SOA. The major restriction of 3-tier architectures compared to SOA is that the type and the number of services is predetermined.

In this thesis we take a closer look at the state that is handled in a 3-tier architecture. The distinction between stateless and stateful services, which we presented earlier, can be applied to the 3-tier architecture as well. This raises the question which of the tiers should handle state (stateful) and which tiers should avoid handling state information (stateless).

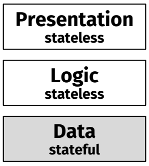


Fig. 2.1: State in a typical 3-tier architecture.

The answer to the aforementioned question is presented in Figure 2.1, where we visualize the state in a 3-tier architecture. It is relatively obvious that the data tier handles state explicitly and is a stateful component (colored gray). According to the service statelessness principle and the efforts of the last ten years of cloud engineering, the tiers above the data tier have typically a stateless design[2].

The reason for this alignment of stateful and stateless components in a 3-tier architecture is that there is a fundamental difference in terms of what stateful and stateless services can achieve with respect to scalability and fault tolerance. In essence, a stateless service can simply be restarted on another machine in case of a failure. Moreover, multiple instances of the stateless service in combination with a load-balancer can be used to increase the throughput and thus the scalability. In contrast to that, scaling a stateful service and providing the necessary fault tolerance is difficult. That is why we use the rest of this chapter to present the challenges that arise when dealing with stateful services.

2 We note that this is a quite oversimplified statement. In reality, application state in form of caches can be found on every tier, and therefore no tier is truly stateless.

2.1.3 *Fault Tolerance*

For stateless services it is relatively easy to achieve increased availability by supporting a *failover*. In case a server failure is detected, e.g. by a missing heartbeat, a failover is typically accomplished by routing new requests to an alternative resource, i.e. another server. In this case, the new server has taken over the *identity* of the failed server [AT03]. This is typically achieved by an update of the DNS record or by rewriting or reusing the IP [AB00], similar to Floating IPs in OpenStack or Elastic IPs in AWS.

For stateful services, however, a failover can only be accomplished if there exists a valid and working up-to-date copy of the state that can be used by a new server. To this end, there are three common approaches to preserve the state of a service:

MESSAGE LOGGING: All incoming requests to the service are temporarily stored on another machine. In case the server fails, the temporarily stored requests are then replayed to an alternative server in order to rebuild the state before the server fault.

CHECKPOINTING: An up-to-date copy of the state is stored periodically on a redundant server. In case the server fails, this copy (read *checkpoint*) is then loaded to an alternative server that can continue to serve the incoming requests.

ACTIVE REPLICATION: All incoming requests are duplicated and simultaneously processed by multiple servers. In this setup, the response is sent from only one server. In case this server fails, the other server takes over and answers incoming requests.

All of the above-mentioned approaches have their advantages and disadvantages. For example, message logging and active replication assume that the result of a request can be deterministically recreated. Any nondeterminism, for example the use of the system clock, would result in a different state and yields unexpected results if a failover is applied. This issue has been addressed in more than ten years of research on deterministic replay [Che+15; MCT08].

The checkpointing approach is widely applied in high-availability systems and has been investigated by researchers for over 20 years. The major disadvantage of this approach is that the maintenance of an up-to date copy, i.e. a checkpoint, binds compute resources and typically has a negative impact on the overall performance of the service. For example, Cully et al. included the periodic store of a checkpoint on hypervisor level, which enables the use of checkpointing for virtual machines that run unmodified software [Cul+08]. Their evaluation revealed that at a checkpoint rate of 20 times per second, the overhead is 52%, where the costs for the additional resources to store the checkpoint are excluded. Applying this approach on a larger scale is, at least at the current state of development, uneconomical.

Despite the need of deterministic operations, active replication is also widely applied. For example, AWS provides a fault tolerant storage called *Elastic Block Store* (EBS), where they achieve high-availability by replicating all write requests on a volume to three availability zones within one region [Ama18]. The availability zones within one region can be seen as isolated data center locations with a fast and reliable network connection. Due to the fast network connection, EBS can tolerate failures of a single data center, i.e. availability zone, without any noticeable performance loss. The major disadvantage is that this concept is rather expensive and cannot be applied outside a fast network, for example between different cloud providers (inter-cloud) or in collaboration with a local data center (hybrid-cloud).

We note that the described active replication mechanism is a relatively simple form of replication. In addition to the duplication of requests, there exists a broad spectrum of replication mechanisms and resulting consistency models. Those mechanisms are not only used to increase the reliability of a service, but also to increase the performance by enabling scalability up to a planetary level. In the next section we explore this spectrum of replication mechanisms in depth and present the necessary background in order to enable replication of stateful components in the logic and presentation tier.

2.2 SCALING WITH REPLICATION AND PARTITIONING

In the previous section we have presented a layered approach for software development, i.e. the 3-layered architecture. The components on each layer are typically placed on different machines which communicate over a network, i.e. a 3-tier architecture. Once the number of requests exceeds the capabilities of one of those machines, two mechanisms, which often go hand-in-hand, can be applied: *partitioning* and *replication*. In addition to achieving scalability, both mechanisms are often used to achieve high-availability, fault tolerance, and an improved response time.

2.2.1 *Partitioning*

The intuition of partitioning is to split the data into small subsets which are then assigned to different nodes. This mechanism is also known as *sharding*. To avoid terminological confusion, however, we prefer the term *partitioning*. In a 3-tier architecture that follows the service statelessness principle, the components in the data tier qualify for this approach. For the sake of simplicity, we assume a standard SQL database in the data tier and illustrate how partitioning can be used to achieve more beneficial properties.

For a database there are several ways to split the data into disjoint subsets. The two commonly used options are partitioning by hashes or by key range. In the first option, a hash function determines the partition of the key. This usually results in a uniform distribution of the data, for the price of a reduced efficiency of range queries, because multiple partitions must be requested. The second option, and the more intuitive one, arranges the partitions based on a key range. For example, the last names of the users in the range from A-D may form one partition. Queries for data within that range, e.g. "list all users with last name A", can be executed more efficiently because no other partition must be queried. In contrast to hashing based partitioning, key range partitioning is prone to hot spots and nonuniform distribution of the data across the partitions.

If a software service is designed to operate with multiple and independent partitions, it usually follows a *shared-nothing* architecture. In contrast to *shared-disk* or *shared-memory* architectures, each node used the CPU, memory, or disk independently. While systems that are designed in a *shared-nothing* architecture certainly provide many benefits, they are also prone to pitfalls. For example, McSharry et al. showed that a single threaded program can easily outperform a misaligned service on a cluster of over 100 CPU cores [MIM15].

2.2.2 *Replication*

Replication essentially means that a copy of the data is kept on multiple nodes, called replicas. The beneficial properties of replication typically include increased fault tolerance and improved performance. On the negative side, there is the coordination overhead to synchronize the shared data, which again depends on the kind of replication mechanism.

In a 3-tier architecture that follows the service statelessness principle, again, only the components in the data tier qualify for replication. This time, we assume a file system volume that can be mounted over the network to illustrate two examples where replication can be applied. The first option is to replicate the entire disk to tolerate a single disk failure, for example in a RAID-1 configuration. The second option is to use a service like NFS or GlusterFS to replicate the volume across multiple nodes. Here, failures of a whole node can be tolerated and the read-performance is significantly improved. Similar to partitioning, a naive application of replication can also be very harmful. In the next chapter we will present that the underlying network between two nodes of a network file system has crucial impact.

Types of Replication

In the literature there is a classification of three forms of replication mechanisms [Kle16]:

LEADERLESS: In leaderless replication, requests are simultaneously sent to multiple replicas. Once a certain quorum is reached, the request is accepted and a value is updated or returned. This replication mechanism is typically used in databases like Amazon's DynamoDB [DHJ+07], Apache Cassandra [LM10], or Basho's Riak [Bas18].

SINGLE-LEADER: In systems with single-leader replication, exactly one leader is responsible to answer write-requests. Thereafter, the write-requests are processed (either synchronously or asynchronously) at the followers, i.e. the replicas that are not the leader. The read-requests are typically answered from the followers directly. This is the standard replication mechanism that is used in most of the SQL-based databases such as

PostgreSQL [Pos18] or MySQL [Ora18], and also for selected nonrelational databases like MongoDB [Inc18].

MULTI-LEADER: In multi-leader replication systems, all replicas can accept and answer all kinds of requests. After one node has accepted one request, the request is sent to the other replicas. Hence, one replica is simultaneously leader and follower for the other leaders. This mechanism is supported for some databases via external tools, e.g. Tungsten Replicator for MySQL [Con14], and often used in online collaboration applications like Google Docs [DR18] or Etherpad [Fou18b].

We note that all of the above-mentioned systems can be implemented in different ways, which consequently implies different properties that are guaranteed. For example, there is an important difference whether the leader in a single-leader system waits until all followers applied an update (synchronous) or continues to process updates without waiting for an acknowledgment (asynchronous). We will address the implied *consistency models* for selected replication mechanisms in the end of this section.

In this thesis we focus on systems with multi-leader replication. We think that the idea that all nodes can accept requests is most appealing. However, this idea is also the reason why those systems are difficult to implement. In essence, there are two options: either all replicas have to agree whether to accept a write-request, or the system tolerates temporary divergence of the replicas and conflicts are allowed to occur. In the literature, those options are typically described as pessimistic- and optimistic replication. Where a pessimistic system waits and thus avoids conflicts, an optimistic system speculates and conflicts are solved after they have occurred [SS05]. This trade-off between waiting and speculating is a manifestation of the CAP dilemma, which we describe in the next subsection.

2.2.3 *CAP Dilemma and Partition Management*

The aforementioned dilemma has been summarized by Eric Brewer; generally known as Brewer's conjecture [Bre00]. In essence, the conjecture states that it is impossible for a distributed system to achieve the following three guarantees simultaneously: Consistency, Availability, and Partition tolerance. Unfortunately, there is no common definition on what those terms actually mean in this context. The term Consistency refers to what is now known as a model for *strong* consistency, which again lacks a formal definition. Intuitively spoken, strong consistency means that only one *consistent* state can be observed in a distributed system of replicas at any time. Availability in this context has a similarly vague definition. The typical definitions refer to a metric that can be observed on a scale from 0 to 100, as it is typically stated in service level agreements which claim an availability of the services of 99.99%. Other definitions state that every request that is received by a non-failing node must result in a response [GL02]. Ultimately, Partition tolerance can be interpreted as "a network partition is among the faults that are assumed to be possible in the system" [Kle15], even though there are again varying definitions [FB99; GL02].

The original conjecture, as visualized in Figure 2.2, provides the intuition that a system must choose to sacrifice one of the three guarantees, resulting in systems that are either AC, AP, or CP. Following that logic, a CP would rather wait and loose availability than risking inconsistent states of the replicas.

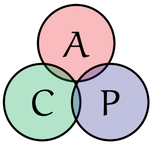


Fig. 2.2: The CAP conjecture.

One example for such system is a database management system, which typically avoids inconsistencies. An AP system, however, would rather loose the consistent view of the data than leaving requests unanswered. The *Domain Name System* (DNS) is one example for such systems due to the inherent inconsistent view that exists because of the used caches at multiple levels. An AC system sacrifices partition tolerance, which raises the question whether an application or a distributed system can choose to deny the possibility of partitions. Even if the system operates in a consistent and available way, if a partition occurs, the system is again faced with the two options *waiting* or *speculating*. In either case, one of the AC properties must be sacrificed.

Critique of the CAP Theorem

This imprecision in the definitions and the confusion about the consequences has led to several critiques about Brewer's conjecture [Kle15]. While the original result was sound and confirmed in a formal proof by Gilbert and Lynch [GL02], there exists a mismatch between the formal abstractions and real distributed systems. For example, Gilbert and Lynch consider consistency to be what is known as linearizability, whereas real systems may implement other consistency models that can be considered *strong* as well.

Daniel Abadi highlighted, that in the original conjecture the notion of latency is completely missing [Aba12]. According to his remarks, there is a finer trade-off between low latency, which can be seen as a special form of availability, and consistency in case no partition is currently present. To this end, he proposes an alternative to the initial conjecture which is called PACELC and it translates to: In case of a partition (P), the application must decide between availability (A) and consistency (C). In case no partition is currently present (read *else* or E), the application can choose between low latency (L) or consistency (C).

Martin Kleppmann further revealed the fine granular differences in the definitions between Brewer, Gilbert, and Lynch, which led to the widely present confusion regarding the CAP Theorem [Kle15]. For a better reasoning about the trade-offs between consistency guarantees and tolerance of network faults, Kleppmann proposes a *delay-sensitive* framework that categorizes the operations either as sensitive to network delay or as independent. Kleppmann further redefines the most commonly used definitions to further enhance the reasoning.

Brewer's Partition Management

Twelve years after the initial presentation of the conjecture, Brewer published an updated view of the CAP dilemma and illustrated how "the rules have changed" in the meantime [Bre12]. He states, that the intuition that one application has to choose between two of the three guarantees is misleading. In real systems, this choice is more fine granular and the choice can even change based on the current status of the network or other circumstances. To this end, Brewer suggested a novel approach to handle the occurrence of network partitions, which we call *partition management*.

In Figure 2.3 we show an illustration of Brewer's partition management. During normal operation, and in the absence of partitions, the application state evolves from update to update but in a consistent way, i.e. there is only one observable state. We note that in this phase, the application is both available and consistent. Once a partition occurs, Brewer suggests that the application switches to a partition mode, where multiple replicas can accept requests that

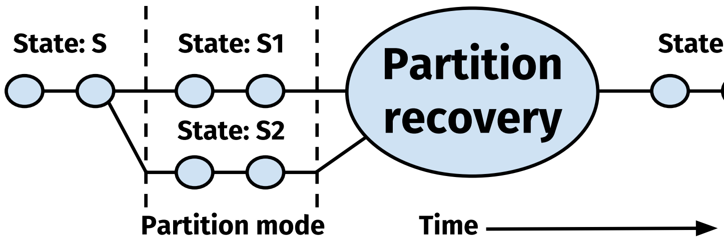


Figure 2.3: Evolving application state according to Brewer's partition management [Bre12].

individually update the state. As already outlined in the original CAP conjecture, this necessarily means that consistency must be sacrificed. Hence, at this point the application prefers availability over consistency. When the network has recovered, the partition mode ends. At this point, there is a minimum of two states that may contain different updates. In order to regain a consistent state, both states must be reconciled in a *partition recovery* process. After the recovery process has finished, the application can continue with only one observable state.

We note that in the above example, the application switches from strong consistency guarantees to weaker guarantees and back. The only thing that enables this agile adjustments is the *partition recovery* process. We further note, that the replicas can independently accept write requests in the presence of a partition and that the state is reconciled after the network has recovered. This is, effectively, a scenario where multi-leader replication is applied.

It is important to understand, that the consequence of Brewer's partition management is that applications, even if they are intended to guarantee a strong consistency model in the absence of partitions, must be designed to support multi-leader replication. This consequence further motivates the goals of this thesis, as we will explore the opportunities of implementing multi-leader application in the presentation and logic tier. Hence, we will explore the *feasibility* of Brewer's partition management in 3-tier architectures[3].

3 This was actually the working title of this thesis. Later we abandoned this title because of the ongoing misinterpretation of the CAP dilemma.

Brewer explicitly mentions two mechanisms to implement multi-leader replication: Operational Transformation [EG89], the mechanism behind collaboration platforms like Google Docs, and Conflict-free Replicated Data Types [Sha+11b]. In this thesis we will explore both mechanisms and ultimately show where both mechanisms have their sweet spot for Brewers's approach. We note that both mechanisms provide a consistency model called *causal consistency*, which we define and compare to other consistency models in the following and last subsection of this background chapter.

2.2.4 *(Causal) Consistency Models*

Consistency models in general are an actively researched topic in the distributed systems community and the first achievements go back to the seventies [Lam79]. In essence, a consistency model can be seen as a *contract* between the client and the application, which defines a relation between read and write requests [TS06]. In case of a replicated system, guaranteeing one model is a challenging task. That is why there exist a wide range of consistency models that are implemented in distributed data stores [VV16].

The strongest model to consider is called *linearizability* [HW90] (sometimes called strong consistency). The intuition behind linearizability is that a system appears as if there were only one copy of data. From an application's point of view, it is not distinguishable whether the data store with linearizability runs on one machine or in fact on multiple replicas. The guarantee that is implied by linearizability can be summarized as follows: "once a new value has been written or read, all subsequent reads see the value that was written, until it is overwritten again". One more intuitive definition of linearizability is that if each operation has a precise execution point somewhere between the invocation of the operation and the return of the result, then these points (or linearizability markers) must form a valid sequence of operations and a line that joins up these points must always move forward in time.

There are certain applications that require linearizability of operations. For example, if a distributed system of processes needs to agree on a single leader, e.g. a system with single-leader replication, distributed locking is used to ensure that only one process becomes the leader [Bur06]. In this case, acquiring and releasing a lock must be linearizable. The implementation of systems with linearizability, however, underlies the same dilemma which we presented earlier: the CAP dilemma. In fact, it has been shown that implementing a linearizable register with more complex atomic operations, such as compare-and-set, is equivalent to consensus [Her91], which is *theoretically* impossible in an asynchronous system where processes are allowed to crash [FLP85]. As a consequence, systems of a larger scale tend to implement more relaxed (or weak) consistency models than linearizability.

At the other end of the scale is a consistency model called *eventual consistency*. The intuition behind eventual consistency is that if no updates take place for a long time, replicas will gradually (and *eventually*) become consistent. This intuition has raised controversial discussions in the research community, because there are no time bounds for when the replicas become consistent [BEH14]. The fact that no real-world system ever stops getting requests makes this definition even more problematic. We will define eventual consistency more precisely in the next chapter.

Causal Consistency and Happened-before Relation

In this thesis we explore multi-leader replication mechanisms that guarantee a consistency model called *causal consistency* [Aha+95]. This consistency model is stronger than eventual consistency but weaker, and therefore easier to implement, than linearizability. The intuition of causal consistency is that all updates that are causally related are seen by all replicas in the same order. Updates that are not causally related, i.e. concurrent, can be seen in a different order on different machines [TS06].

In order to precisely define causal consistency, the notion of causality between two events must be defined. In distributed systems research, causality is expressed with the *happened-before* relation, originally introduced by Lamport in 1978 [Lam78].

Definition 1 (happened-before). The *happened-before* relation $\prec$ is defined as a strict partial order of events such that:

- If events a and b occur on the same process, $a \prec b$ if the occurrence of event a preceded the occurrence of event b.

- If event a is the sending of a message and event b is the reception of the message sent in event a, then $a \prec b$.

We note that the happened-before relation in Definition 1 creates a relation between events that are *potentially* causally related. With this relation, two events that are technically independent could be ordered as happened-before, e.g. two consecutively executed operations that operate on different objects. This imprecision is generally accepted in the development of distributed systems. There are, however, ongoing debates whether an application designer should blindly apply the happened-before relation (by using a causal-order broadcast middleware) or track the causality manually in the application [CS93; Bir94]. A causal-order broadcast middleware is typically implemented with the help of vector clocks.

3 ON STATEFUL LOGIC TIERS WITH CRDTS

3.1 CHAPTER OVERVIEW

This chapter is based on previous work by the author and co-authors of [JO17; JOL17b; JOL17a].

In this chapter we explore the feasibility of storing more state in the logic tier. In contrast to 3-tier architectures that follow the service statelessness principle, storing more state than necessary yields promising opportunities alongside with challenges that need to be addressed. We explore these opportunities and challenges and exemplify them with a new approach to handle the state of an IMAP service.

Among the opportunities are promising benefits for applications with respect to reduced response time, increased fault tolerance, and a scalability that can never be achieved with *traditional* approaches where state is primarily held in one component or one tier. As we will show with our later evaluation, the techniques we explore in this chapter qualify to design systems of planetary-scale; serving clients from different continents of this planet.

The challenges include the integration of multi-leader replication into the logic tier of systems that are traditionally not designed to be replicated. To this end, we utilize *Conflict-free Replicated Data Types* (CRDTs), which have been proposed as a method for avoiding conflicts [Sha+11b] per design. We set out to model, verify, implement, and evaluate a distributed IMAP service with non-trivial state based on CRDTs. IMAP is a simple and rather old standard—its beginnings date back to the mid-1980s—and as part of the email ecosystem is regularly proclaimed dead in favor of some supposedly more efficient communication service. Yet, email remains to be ubiquitous in all our lives and will stay so for the foreseeable future.

Even though the provided CRDT primitives [Sha+11a] are concise and simple, one can fail in numerous ways when constructing non-trivial system state based on these. We want to be sure of the correctness of our model and thus put effort into proving it correct. To this end, we extend the CRDT and network model framework by Gomes et al. [Gom+17b; Gom+17a], written in the interactive theorem prover Isabelle/HOL, to include our IMAP-CRDT. After being assured that state will always be consistent in our model, we adapt our prototype to adhere to the theoretical proof. This way, we achieve provable consistency guarantees in practice.

With the achieved multi-leader replication in the logic tier of an IMAP service, we demonstrate the benefits in a comprehensive evaluation against the industry standard IMAP software *Dovecot*. In addition to the increased fault tolerance, we show that the adapted system qualifies to run on planetary-scale. From the insights of the evaluation we derive a discussion on the setups where our approach can be beneficial.

The contributions that are addressed in this chapter include:

- We propose an IMAP-CRDT by modeling IMAP commands as operations on a CRDT.
- We verify the convergence of the IMAP-CRDT with the interactive theorem prover Isabelle/HOL.
- We propose an open-source prototype *pluto* that offers IMAP at planetary-scale with multi-leader replication based on CRDTs.
- We introduce a benchmark for IMAP services.
- We propose a Kubernetes-based deployment for planet-scale *Dovecot*.
- We explore response time performance and replication lag of planetary-scale IMAP services on public clouds by evaluating the developed prototype *pluto* against state-of-the-art *Dovecot* setups.

3.2 CONFLICT-FREE REPLICATED DATA TYPES

The theoretical concept of a Conflict-free Replicated Data Type has been formalized by Shapiro et al. in [Sha+11b]. In essence, CRDTs enable convergence of replicas without requiring a central coordination server or even a distributed coordination system based on consensus or locking. To achieve this goal, updates on an application's state based on CRDTs are designed to be conflict-free in the first place. With this property, CRDTs fall into the category of mechanisms that can be used for systems with multi-leader replication.

CRDTs offer a simple and theoretically sound approach to eventual consistency. In fact, the authors show that the implied consistency model, namely *strong eventual consistency*, is actually a more strong and more desirable model than eventual consistency. In this section we explain the fundamental definitions and properties. Later in this chapter, when we introduce our own IMAP-CRDT, we will refer to the definitions that we present here.

System Model

The authors of the original work on CRDTs, and the corresponding technical report, consider a distributed system of processes that communicate over an asynchronous network. Partitions of the network can occur and recover at any time, as well as processes can crash and recover. It is important to assume that the memory of a crashed process survives crashes, in order for the process to recover. Non-byzantine behavior of the processes is assumed.

Operation-based CRDTs

CRDTs come in two variants: *Convergent Replicated Data Types* (called CvRDT) and *Commutative Replicated Data Types* (called CmRDT). CvRDTs, often described as state-based CRDTs, ensure convergence by defining a *merge* function that is applied on two diverged states in order to obtain a consistent state again. The merge function calculates the *least upper bound* on a *join semi-lattice*, and therefore must be commutative, idempotent, and associative. A replica can update its local state and send the updated version to all other replicas which individually apply the merge function to regain a consistent state. The order in which the merge function is applied is irrelevant.

In this work we focus on the operation-based variant (CmRDTs), which we will explain in depth. In contrast to state-based ones, replicas exchange operations directly with minimal state information. A reliable causal-order broadcast ensures that operations ordered by the *happened-before* relation (see Definition 1) on the source replica are received and applied accordingly at all other replicas.

Updates that cannot be ordered by *happened-before* are considered *concurrent* and are required to commute. The design of a CmRDT is a challenging task, fortunately the technical report offers a variety of specifications for counters, sets, graphs, and even lists [Sha+11a].

Here, the definition of a CmRDT is composed of the following components:

1. The **payload** describes the type of state, e.g. a simple integer or a set. Furthermore, an **initial** state must be specified, which represents the initial value of the payload at every replica.

2. A **query** operation is an operation that does not modify the state. Typically, query operations are read-operations.

3. An **update** operation is an operation that modifies the state. The definition is further divided into an **atSource** and a **downstream** part. The atSource part contains preconditions that must hold for the state of the replica that is initiating the operation. Furthermore, certain information can be queried from the state of the local replica atSource. The downstream part is asynchronously executed at every replica, including the replica that initiated the operation. Typically, the downstream definitions are state changing functions.

As mentioned, CmRDTs require a reliable causal-order broadcast to ensure convergence. We note that an implementation of such a broadcast does not require consensus and can be achieved by use of vector clocks.

With the commutativity of concurrent updates and a reliable causal-order broadcast, Shapiro et al. showed that any two replicas that have seen the same set of operations have equivalent abstract states and therefore eventually converge [Sha+11b]. The authors formalize this notion of eventual convergence by introducing the *Causal History* as shown in the following definition:

Definition 2 (Causal History). The causal history of a replica x is defined as follows.

- Initially, $\mathcal{C}(x) = \varnothing$.

- After executing the downstream phase of operation f at replica x, $\mathcal{C}(f(x)) = \mathcal{C}(x) \cup \{f\}$

In essence, the causal history is the set of all operations that were executed at a replica. Later in this chapter we rename the causal history to the *event log* of a process.

With the notion of the causal history, the authors define *Eventual Convergence* as follows:

Definition 3 (Eventual Convergence). Two replicas x and y *converge eventually*, if the following conditions are met:

- Safety: $\mathcal{C}(x) = \mathcal{C}(y)$ implies that the abstract states of x and y are equivalent.
- Liveness: $f \in \mathcal{C}(x)$ implies that, eventually, $f \in \mathcal{C}(y)$.

We note that the safety property represents the already mentioned requirement that two replicas that have seen the same set of operations, not necessarily in the same order, reach the same abstract state. Here, two abstract states are equivalent if all *query* operations, i.e. read-operations, return the same values for all inputs. The liveness property, however, ensures that progress can always be made. In the above introduced system model, it is assumed that processes can crash and recover and network partitions eventually *heal*. For the rest of this chapter we focus on the safety property. In contrast to the liveness property, which is not related to the design of a CRDT, the safety property must be respected when operations are designed.

It is noteworthy that CmRDTs require only concurrent operations to commute. If an update operation would be commutative regardless it has *happened-before* another operation or not, then convergence is obviously achieved, because all operations can be reordered. We note that this requirement would be too strong to be practical and would exclude many valuable CRDT definitions. To this end, showing that only concurrent operations commute is less restrictive. However, a careful design is still necessary, because mistakes can easily be made.

We illustrate two examples of a CmRDT, namely a counter and the Observed-Remove Set, in the rest of this section.

Specification 1 Counter CmRDT [Sha+11b]

```
1: payload an integer $i \in \mathbb{Z}$        ▷ The payload is a single number
2:     initial $i \triangleq 0$        ▷ Initially, the value of the counter is 0
3: query value () : integer j
4:     let $j = i$        ▷ A side-effect free let instruction
5: update increase ()
6:     downstream ()
7:         $i \triangleq i + 1$
8: update decrease ()
9:     downstream ()
10:        $i \triangleq i - 1$
```

Two CmRDT Examples

The simplest CmRDT to consider is an integer counter. In essence, the counter offers operations to read the current value, as well as to increase and decrease this value by 1. We show the specification of this Counter CmRDT in Specification 1.

We note that all of the above mentioned components of a CmRDT, i.e. the payload, the initial state, query operations, and update operations, are defined in the referenced specification. The used presentation style is adopted from the original introduction of CRDTs from [Sha+11b]. We refer to the corresponding technical report [Sha+11a] for further details on CRDT descriptions.

The Counter CmRDT is rather special, because of its simplicity. For example, the **update** operations define no preconditions and therefore the **atSource** part is empty. Moreover, no parameters are ever given to any operation. The query operation *value* is a simple read operation that returns the current value of the counter at the replica where the query is executed. We note that the *let* construct is used to refer to a side-effect free function that is computed at the replica where the operation is executed.

For this CRDT, it is obvious that eventual convergence is guaranteed. The *increase* and *decrease* operations are commutative, regardless of whether they are executed concurrently or one *happened-before* the other. It is noteworthy that this particular CmRDT converges

Specification 2 Observed-Remove Set CmRDT [Sha+11b]

```
1:  payload a set S of pairs of elements and unique-tags
2:      initial $\varnothing$
3:  query lookup (element $e$) : boolean $b$
4:      let $b = (\exists u \,.\, (e, u) \in S)$    ▷ Checks if an element is in the set
5:  update add (element $e$)
6:      atSource ($e$)
7:          let $\alpha = unique()$    ▷ $unique()$ returns a unique value
8:      downstream ($e, \alpha$)
9:          $S \triangleq S \cup \{(e, \alpha)\}$
10: update remove (element $e$)
11:     atSource ($e$)
12:         pre $lookup(e)$
13:         let $R = \{(e, u) \mid \exists u \,.\, (e, u) \in S\}$ ▷ Compute the remove set
14:     downstream ($R$)
15:         pre $\forall (e, u) \in R \,.\, add(e, u)$ has been delivered
16:         $S \triangleq S \setminus R$    ▷ Remove R at the downstream replicas
```

even in case the operations are not *causally ordered*. So in this case, the requirement of a *causal-order broadcast* is unnecessary.

In contrast to that, the second CmRDT we introduce, the Observed-Remove Set (OR-Set), requires a *causal-order broadcast* to guarantee convergence. We show the Specification of the OR-Set in Specification 2.

The OR-Set's payload is a set that contains pairs of elements and unique-tags. The elements are the actual elements or *items* of the set, and the unique-tags can be seen as metadata that is required to track the state of the set. In addition to *lookup*, which is an intuitive *read* function on the set, there are two more update operations defined: *add* and *remove*.

The *add* operation inserts an element to the set by attaching a unique tag to it. The unique tag must be globally unique, so that two *add* operations for the same element from two different replicas can be distinguished. More concrete, if two replica concurrently adding an item i to the set, there would be two pairs with the element i in the set after the downstream operations are executed at every replica.

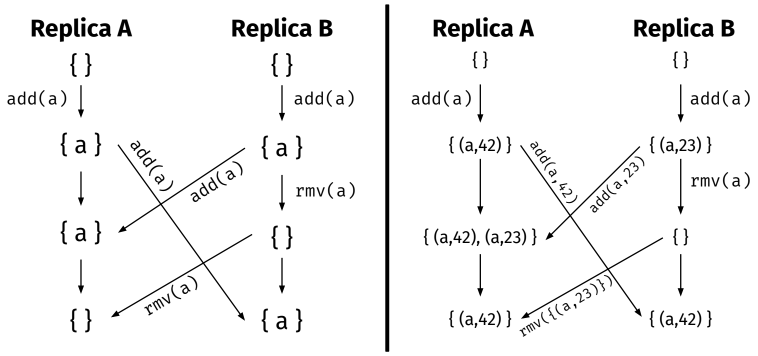


Figure 3.1: Diverging replicas when applying a naive exchange of the operations (left) compared to converging replicas of an OR-Set (right).

The *remove* operation, however, uses the tags to determine whether an item should be removed entirely, or concurrent *add* operations reinserted the item. We illustrate this puzzle in the left side of Figure 3.1. Without the tags, applying the operations may lead to diverged states; in this case $\varnothing$ on the left replica and $\{a\}$ on the other. We note that causality is not an issue here, because the shown communication is in accordance with the *happened-before* relation.

In the right side of Figure 3.1 we show how the OR-Set uses the tags to reference the corresponding *add* operations. In this example, after both replicas concurrently added the element a with two different tags, the remove set R is computed at replica B and transmitted to A. At replica A, the complement of R with respect to the state at replica A is computed, resulting in $\{(a, 42)\}$.

We note that the OR-Set requires a causal-order broadcast to achieve convergence. Without causal-order delivery, a *remove* operation may arrive before the corresponding add is delivered. This would most likely result in diverging states, because the replica that receives the operations in the aforementioned order would still contain the added element, in contrast to the replica that executed the operations in accordance with the *happened-before* relation.

The proof of the commutativity of the operations is relatively simple. Combinations of *add-add* or *remove-remove* are in all cases commutative. The mentioned *add-remove* puzzle can be proven to be commutative when the tags of the *add* operation are in fact globally unique and operations are applied in causal order.

Both of the here presented CRDTs were introduced with the technical report of the original CRDT publication [Sha+11a]. In addition to the counter and the OR-Set CRDT, the report includes many more datatypes, e.g. for ordered lists (RGA) [Roh+11]. More recently, even more versatile CRDTs, like the JSON-CRDT from Kleppmann and Beresford, were introduced [KB17]. We summarize this related work in the end of this chapter.

Strong Eventual Consistency

CRDTs imply a consistency model called *strong eventual consistency* (SEC). In contrast to eventual consistency, where it is only guaranteed that all replicas *eventually* reach the same state after all operations are exchanged and a certain amount of time has passed, SEC is in fact a *stronger* model.

Shapiro et al. compare both consistency models with the following definitions. In Definition 4 we show the formalized version of the above mentioned definition of eventual consistency according to [Sha+11b]. We note that the definition is closely related to the definition of eventual convergence in Definition 3. Eventual convergence is a property that is used to describe the desired behavior of two replicas. In contrast to that, eventual consistency is a property, or a *guarantee*, of a system. In essence, the here mentioned definition of eventual consistency states that eventual convergence holds for any two correct processes/replicas.

Definition 4 (Eventual Consistency (EC)). *Eventual Consistency* is guaranteed, if the following properties hold:

> **Eventual Delivery:** An update delivered at some correct replica is eventually delivered to all correct replicas.
>
> **Convergence:** Correct replicas that have delivered the same updates eventually reach equivalent states.
>
> **Termination:** All method executions terminate.

Several systems achieve eventual consistency by allowing updates on one replica immediately, only to discover later that there were conflicts with concurrent updates. Such systems resolve detected conflicts by performing a roll-back to a previous state. Shapiro et al. discovered, that CRDTs actually imply a stronger version of eventual consistency, hence SEC. In the above mentioned definition, convergence is reached *eventually* after replicas have seen the same set of updates. CRDTs ensure that replicas that have seen the same set of updates have in fact equivalent abstract states. To this end, Shapiro et al. redefine the convergence property by omitting the *eventually* in the definition, as shown in Definition 5 [Sha+11b].

Definition 5 (Strong Eventual Consistency (SEC)). *Strong Eventual Consistency* is guaranteed if Eventual Consistency is guaranteed and:

> **Strong Convergence:** Correct replicas that have delivered the same updates have equivalent states.

3.3 A CASE STUDY FOR IMAP

In this section we explore the feasibility of storing state in the logic tier; exemplified with IMAP. Therefore, we utilize CRDTs to enable replicated IMAP servers that synchronize the state without electing a leader. Hence, we propose an IMAP server that allows multi-leader replication by introducing the IMAP CmRDT. We put a special focus on the verification of our proposed CRDT to guarantee that convergence is achieved. Ultimately, we test our approach by evaluating our research prototype *pluto* against the de facto standard IMAP server *Dovecot* in the next section.

The decision to pick IMAP as example service is based on two reasons. The first reason is the popularity of IMAP in the sense that essentially every company of the world uses IMAP as part of their electronic mail ecosystem. The second reason is that IMAP is a comparable simple protocol with well defined control commands and a *managable* structure of the state.

Before we propose our CRDT-based solution that enables an IMAP service with multi-leader replication, we explore the currently existing solutions. Therefore, we present typical configurations of scaled out IMAP services exemplified with *Dovecot* in the following subsection.

3.3.1 *Dovecot and State-of-the-art Configurations*

The largest IMAP service we have today is Google's GMail. Google recently reported that Gmail exceeds the mark of one billion active users [Mil16]. In Germany, the biggest mail provider is the Deutsche Telekom with over 26 million active users[1]. Timo Sirainen, the primary author of *Dovecot*, reports that the IMAP Service of the Telekom runs on *Dovecot* and is one of the biggest German *Dovecot* installations.

Dovecot, as the de facto standard IMAP server, enables various ways of deployment. In the simplest configuration, *Dovecot* runs on a single machine; processing IMAP requests from the registered users and storing the mailboxes on the local disk. While this setup obviously has limitations in terms of scalability and maintainability, for example in case of a defective memory module the whole service needs to be stopped and no requests can be processed at this time, the performance of *Dovecot* is quite impressive.

Since this work focuses on systems of a certain scale, i.e. systems that are build to run on multiple machines and multiple tiers, we briefly discuss how a scaled out *Dovecot* system would look like. In order to achieve a higher scale, the system can be split up into multiple tiers and can be combined with partitioning (see Section 2.2.1), where each machine handles mailboxes of a different range of users. The typical configurations of a scaled out *Dovecot* system look a follows:

- A proxy redirects a user's requests to a random backend where the requests are processed and the mailboxes are stored on a shared file system like NFS. In this case, the system is not designed to use partitions, since the backends are chosen randomly and every backend is able to process requests for every user. In order to prevent race conditions with concurrent updates of the same mailbox, for example by accessing one mailbox with two different devices, a director service, which is an extension of the proxy, keeps track of the current connec-

1 An explanation for the popularity of the mail service of the Deutsche Telekom would be, that back in the days of dial-up modem internet connections, the Telekom was one of the first German internet providers. Together with the dial-up program, the *T-Online StartCenter*, they offered a free pop3 mail client and a first email address for their customers.

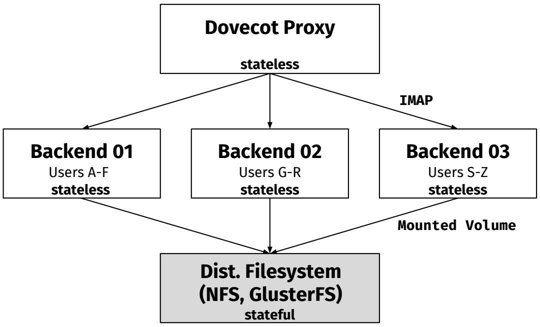


Figure 3.2: A *traditional* configuration of a scaled out *Dovecot* installation with enabled partitioning.

tions and assigns accesses to the same mailbox to the same backend.

- A proxy redirects users requests to a particular backend based on fixed rules. The assigned backend is used whenever possible. The mailboxes are stored on a shared file system. If a particular backend is unavailable, any other backend can continue handling the requests since the *critical* application state, i.e. the mailboxes, is stored outside of the unavailable backend. We illustrate this setup in Figure 3.2, where the components are arranged in the mentioned tiers and the state is stored on an NFS/SAN storage solution.

We note that in both cases the backends in the logic tier do not store *mission critical* state. Hence, both designs follow the service statelessness principle. In order to increase the performance of read-intensive requests, *Dovecot* can be configured to store caches (called index files) on the backend. The index files are automatically created on the first access of the mailbox and reduce the response time for further read requests. In the first of the two mentioned configurations, the system can only benefit from temporary caches, whereas in the second case the caches can be stored permanently on the backend.

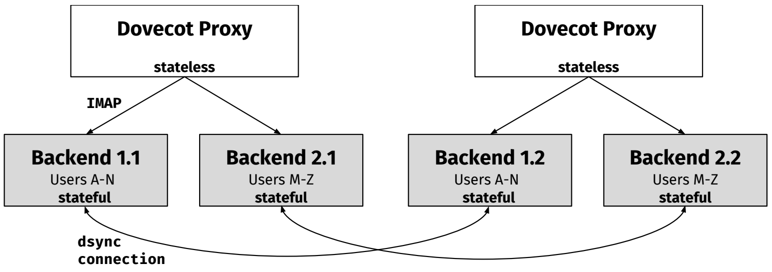


Figure 3.3: A *Dovecot-dsync* configuration with enabled partitioning and replication.

Dovecot dsync replication

In addition to partitioning, *Dovecot* provides a replication extension to further increase the performance and the tolerance against faults. The *dsync* extension enables a one- and two-way synchronization of the users' mailboxes, with a maximum number of two replicas. The typical configuration of a *dsync*-enabled *Dovecot* setup include pairs of backends that store the users mailboxes on the local hard drive. Hence, in this setup, no shared files system like NFS is used. We illustrate the *dsync* setup in Figure 3.3. Moreover, the figure shows a combination of partitioning and replication by dividing the users into two groups to illustrate that this configuration can be scaled out as well.

We note that a *dsync*-enabled setup actually stores *mission critical* state in the logic tier and provides a replication mechanism to synchronize between two notes. One could argue that *dsync* already solves the challenges that we illustrated in Section 1.2 and makes further research pointless. However, while *dsync* indeed solves the same problem that is in focus of this chapter, the generalization and the transferability of the underlying concepts are missing. In essence, the major disadvantages of the underlying mechanism of *dsync* are:

1. Only two replicas are allowed: *dsync* currently only supports pairs of backends to synchronize state. In contrast to that, in this thesis we investigate multi-leader replication in the logic tier with possible arbitrary many leaders/backends.

2. Highly application dependent: There are no background information how *dsync* is achieving the synchronization and if the concept can be generalized or applied to other services. In contrast to that, in this chapter we explore the feasibility of using CRDTs to ensure convergence of an application's state. CRDTs offer an extensive theoretical framework that can be applied to other services and, as we will point out in the rest of this section, to IMAP as well.

3. No *formal* convergence guarantee: While we could not find any violations against *dsync*'s promise to keep the mailboxes in sync, a formal verification of the guarantee is missing.

The mentioned disadvantages, or *challenges*, motivate a deeper look into the problem of guaranteeing convergence of the shared state among the backends. However, *dsync* qualifies as an excellent candidate for a comprehensive evaluation. In Section 3.5 we present the evaluation of our prototype and compare the response time performance as well as the replication lag of our prototype against *dsync*.

Object Storage for Mailboxes

We note that the above mentioned configurations and the illustration in Figure 3.2 propose a shared file system like NFS to store the state. This implies that the state is stored as folders and files on a volume that is ideally tolerant against single-disk failure. Even though there are highly sophisticated and high performing solutions for optimizing the data tier, e.g. by investing in a storage area network (SAN), it can easily become a bottleneck in the above mentioned configurations. The most obvious way to avoid that is to apply partitioning one more time. In this scenario the data tier is divided into multiple partitions where each partitions stores the mailboxes of the assigned group of users. Further applying replication on file system level is certainly possible in the same data center, but can lead to performance drops when applied beyond the boundaries of one data center.

A more recent approach to achieve performance on a planetary-scale is to use object storage. In such configurations, the backend stores the mailboxes not as files, like the maildir format, but as objects with a key. Modern highly available NoSQL database systems like Cassandra [LM10] offer scalability and replication between multiple data centers. The combination of *Dovecot* and enabled object storage on a Cassandra-like system is probably the most advanced configuration to consider. This configuration enables a scalable IMAP service with tolerance against a variety of possible faults and failures.

Unfortunately though, *Dovecot* extensions that allow mailboxes to be stored on an object store are only available as proprietary add-on. To the best of our knowledge, no open-source solution has yet been published. In contrast to the goal of this thesis, the object-storage solutions focus on solving all replication related issues in the data tier. As the goal of this chapter is to analyze the feasibility to store more *mission-critical* state in the logic tier, the mentioned object store setups are out of scope of this thesis, even though a comprehensive evaluation is certainly interesting. We discuss the derived research opportunities and future work in Chapter 5.

3.3.2 *State of an IMAP Server*

Today's email system is composed of a variety of interacting services and various email-specific protocols. Before we dive deeper into our protocol of interest, i.e. IMAP, we shortly summarize today's mail ecosystem and the necessary steps to send and receive emails. After that, we explore IMAP in depth and analyze the structure of the state that can be read and modified with IMAP commands.

The Electronic Mail Ecosystem

At the time of writing this thesis, electronic mail (or *email*) already has a history of about 50 years. The very first attempts to send messages between users of the same system goes back to 1960, where time-shared operating systems offered the functionality to store messages [Par08]. We note that at this time, there were no modern networks or similar complex *inter-computer* communication.

Since then, many protocols and standards have been established. The most noteworthy can be summarized as follows:

INTERNET MESSAGE FORMAT: The RFC 5322 describes a syntax for text messages that are sent between computer users [Res08]. It is the current revision of the earlier introduced RFCs 822 and 2822. The Internet Message Format describes the fields of an email message, including the header fields and the body.

SIMPLE MAIL TRANSFER PROTOCOL (SMTP): This protocol is used for the transmission of messages. It is a simple, human-readable protocol and is specified in RFC 821 [Pos82].

POST OFFICE PROTOCOL (POP): This protocol enables interaction with a remote mailbox that is not located on the client's machine. The set of features include a login and the download of new messages. The messages are typically deleted after they have been retrieved via POP, resulting in an empty mailbox on the server [Sie07]. The disadvantage is, that only one client can be active at a time, resulting in difficulties to use POP with multiple devices.

INTERNET MESSAGE ACCESS PROTOCOL (IMAP): This protocol is, in contrast to POP, designed to enable interaction with many devices. It is a text-based request/response protocol and widely used over the Internet [Cri03]. The main purpose of IMAP is to interact with a mailbox on a server. The IMAP commands are specified in RFC 3501.

When a user sends an email to another user, the above mentioned protocols perform an interplay with various services involved. We illustrate this interplay in Figure 3.4. The figure shows the involved services and protocols when sending and receiving an email.

A new email typically starts with a new window in the *Mail User Agend* (MUA) of a client. Examples include *Thunderbird* or *Outlook*. After the user presses the send button, the MUA initializes an SMTP-based communication with the users *Mail Transfer Agend* (MTA) and transmits the message. The sender's MTA transmits the message to the receiver's MTA over SMTP. The MX-records, as part of the *Domain Name System* (DNS), are used to identify the MTA that is responsible of handling the receiver's mailbox. After the email has been transmitted to the receiver's MTA, a *Message Delivery Agent* (MDA) is used to place the message in the receiver's mailbox. Thereafter, the receiver uses POP or most likely IMAP to retrieve

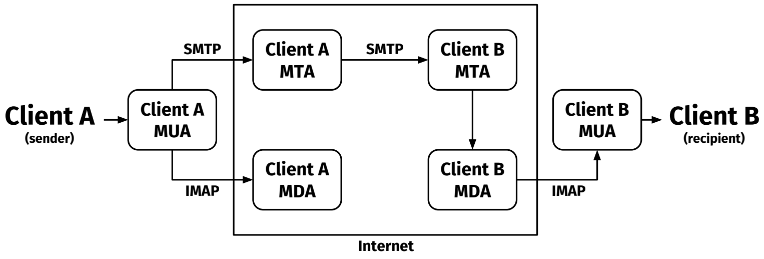


Figure 3.4: Interplay of various email-related protocols when sending a message from A to B.

the new message. *Dovecot* includes MDA functionality, but is mainly responsible to organize user's mailboxes and answers the client's IMAP requests, that are typically invoked by the client's MUA.

IMAP

An IMAP service manages mailboxes of registered users. Users are able to interact with their mailboxes by sending IMAP commands to the server. These commands are defined in the *IMAP4rev1* standard in RFC 3501 [Cri03].

In Figure 3.5 we show the *State and Flow Diagram* from RFC 3501, where we see the different states of an IMAP session. Initially, one client starts in the *Not Authenticated* state, where the client is able to login or authenticate in order to switch to the *Authenticated* state. In the *Authenticated* state, the client is able to interact with the mailboxes, for example by creating or deleting them. The client is able to select a particular mailbox and move to the *Selected* state. In the *Selected* state, the content of a particular folder can be modified, for example message flags can be altered and messages can be deleted. From the *Selected* state, the client is able to switch back to the *Authenticated* state or select another mailbox. In all states, the client is able to move to the *Logout* state by performing the corresponding command or in case something went wrong.

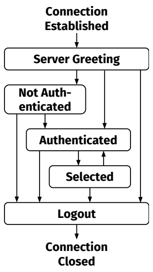


Fig. 3.5: State and Flow diagram from [Cri03].

The RFC 3501 defines a total of 25 client commands in the following allowed states:

Any state	`CAPABILITY`, `NOOP`, `LOGOUT`

Not Authenticated	`STARTTLS`, `AUTHENTICATE`, `LOGIN`
Authenticated	`SELECT`, `EXAMINE`, `CREATE`, `DELETE`, `RENAME`, `SUBSCRIBE`, `UNSUBSCRIBE`, `LIST`, `LSUB`, `STATUS`, `APPEND`
Selected	`CHECK`, `CLOSE`, `EXPUNGE`, `SEARCH`, `FETCH`, `STORE`, `COPY`, `UID`

After a client invoked one of the above mentioned commands, the server responds with a *status response* like `OK`, `NO`, `BAD`, `PREAUTH`, or `BYE`.

We illustrate an example of an IMAP session in Listing 3.1. In the beginning, the client initiates the communication preferably over a secured and reliable channel. Once the connection is established, the server prints the *Server Greeting*, as shown in line 3. The greeting includes the escape character, that is used to determine the hierarchy between folders, followed by the capabilities of the server. In line 5, the client invokes a `LOGIN` command by sending a tag, the `LOGIN` command, a username, and a password. In the following line, the server answers the request by sending the tag of the original request and a status response, in this case `OK`. The rest of the exemplified IMAP session follows the same communicational pattern. In this example, the client selects the *Inbox* and permanently deletes all messages with a *deleted* flag via `EXPUNGE`. Thereafter, the session is closed with a `LOGOUT` command.

For the rest of this chapter we use the typewriter font to refer to IMAP commands, e.g. `LOGIN` *or* `CREATE`.

We omit a detailed description of the IMAP commands and refer to the RFC 3501 [Cri03]. Later in this section, however, we will put a strong focus on the *consistency critical* commands and provide the necessary intuition.

Maildir and other Mailbox Formats

IMAP servers, like *Dovecot*, support various formats to represent the mailboxes of the users on hard disk. Typical formats are *mbox* and *Maildir*. The latter is generally preferred due to its use of individual files per mail message and thus, no locking is required when messages are appended. The messages are given unique file system names that include any potential standard flag.

From this structure we derive our notion of the state of an IMAP server. The most obvious components are the user's mailboxes. For the rest of this work, we write *mailbox folder* or simply *folder* when we refer to a mailbox in the account of one user. This notation avoids some confusion with the German parlance, because in the German language a mailbox typically refers to the account and a folder refers to a mailbox. The term *folder* is also more appealing, because most MUAs and web interfaces visualize the mailboxes similar to a file system where several folders, sub folders, and the items inside a folder, i.e. the messages, are shown.

We note that the folders within one account span a tree-like hierarchy. This is done by using an escape character inside a folder's name. Other than that, IMAP does not provide any more explicit commands to interact or modify the generated hierarchy. We also note that there is no explicit order between folders in the same hierarchy. Folders are accessed by the foldername which serves as a unique identifier.

The second obvious component of the state of an IMAP service are the actual messages. The messages are typically formatted in accordance with the Internet Message Format (RFC 5322 [Res08]), i.e. they contain a header with important metadata and a body

Listing 3.1: Example communication with an IMAP server.

```
Trying 192.168.23.42...
Connected to 192.168.23.42.
S: Escape character is '^]'.
S: * OK [CAPABILITY IMAP4REV1 STARTTLS AUTH=LOGIN]
C: A1 LOGIN username password
S: A1 OK [CAPABILITY IMAP4REV1] User username authenticated
C: A2 CREATE work
S: A2 OK CREATE completed
C: A3 SELECT Inbox
S: * 2 EXISTS
S: * 2 RECENT
S: A3 OK [READ-WRITE] SELECT completed
C: A4 EXPUNGE
S: * 8 EXPUNGE
S: A4 OK EXPUNGE completed
C: A5 LOGOUT
S: A5 OK LOGOUT completed
```

that contains the message's content. IMAP treats a message as a literal, which means that there are no commands that offer any modification of the message itself. For example, one cannot edit the content or the sender of a message inside a folder without deleting the message and reinserting the altered message.

We note that the IMAP commands that refer to a message, such as `STORE`, use a *message sequence number* to identify a message inside a folder. The message sequence number is a relative number from 1 to the number of messages in a folder. If a message is removed permanently from the folder, a message sequence number can be reassigned during the session.

The only additional state for a message that is introduced by IMAP are the message flags. The flags represent additional information about the message, e.g. a message may be marked as *recent* or *deleted*. Hence, the state contains an unordered list of flags for each message in the system. As mentioned, in Maildir the list of flags is part of the filename of a message.

In addition to the folders and the messages, the state contains a few less obvious information. For example, a user can subscribe and unsubscribe to a certain folder, resulting in an unordered list of subscribed folders that can be read with the command `LSUB`. The list of subscribed folders is typically used to define which folders should be checked regularly in order to fetch new messages.

3.3.3 *Modeling IMAP with CRDTs*

In our scenario, the main challenge is to model the above identified state of an IMAP server as a payload and the commands as operations on a CmRDT.

As mentioned in the above subsection, the application state can be summarized as follows: one user has a set of folders which contains a, possibly empty, set of messages. We identified a map that projects foldernames to the content of a folder to be best suited. Therefore, the content of a folder is a combination of metadata (tags) and messages. We model the map as a function where $\mathcal{N}$ is the set of foldernames, $\texttt{ID}$ is the set of tags, and $\mathcal{M}$ is a set of messages. We denote $\mathcal{P}(X)$ to be the power set of X:

$$u : \mathcal{N} \to \mathcal{P}(\texttt{ID}) \times \mathcal{P}(\mathcal{M})$$

Because a folder f contains arbitrary items, the result of $u(f)$ is a tuple of sets. The first set, denoted as $u(f)_1$, is the set of tags that represent metadata that should not be visible to a user. The second set, denoted as $u(f)_2$, represents the messages in the folder.

If both sets $u(f)_1$ and $u(f)_2$ are empty, the folder is interpreted as non-existent. Hence, we distinguish between a non-existent folder and an empty folder. A folder is empty, if $u(f)_2$ is empty but $u(f)_1$ is not empty, i.e. certain metadata is present. Initially, all folders are non-existent. Hence, the initial state can be described as a lambda abstraction that projects the tuple $(\varnothing, \varnothing)$ to every foldername in $\mathcal{N}$.

We note that this description of the initial state is chosen to simplify the formal reasoning, which we present later in this section. In practice, there are certain requirements that specific folders must exist, e.g. the INBOX. Hence, a more accurate but slightly more complicated initial state would include the mapping "INBOX $\mapsto (\{42\}, \varnothing)$" where the number 42 can be seen as metadata, but the folder is empty, i.e. $u(\text{INBOX})_2 = \varnothing$.

The flags of a message can be encoded into the message, similar to Maildir. To model the list of subscribed folders, we identified a simple set to be best suited, because there is no particular order between the selected folders.

Consistency Critical Commands

Next, we analyze the IMAP commands in more detail. To reduce the complexity of this work, we focus on the *consistency critical* commands, i.e. the commands that may change the state.

From the 25 IMAP commands we listed in the above subsection, the following commands can be considered as *consistency critical*:

Authenticated	`CREATE`, `DELETE`, `RENAME`, `SUBSCRIBE`, `UNSUBSCRIBE`, `APPEND`
Selected	`EXPUNGE`, `STORE`, `COPY`

We note that the commands that were not mentioned here are basically not consistency critical because they do not alter the state. Those commands can be considered as *read* commands, that would be designed as *query* operations on a CmRDT. For the rest of this

Command	Description
`CREATE`	Creates a folder with the given name n.
`DELETE`	Permanently removes the folder with the given name n.
`RENAME`	Changes the name of the folder from n_{old} to n_{new}.
`SUBSCRIBE`	Adds the foldername n to the list of subscribed folders.
`UNSUBSCRIBE`	Removes the foldername n from the list of subscribed folders.
`APPEND`	Appends the new message m to the folder n.
`EXPUNGE`	Permanently removes all messages with a `deleted` flag from a previously selected folder n.
`STORE`	Alters the flags of a messages m in folder n based on the flag command.
`COPY`	Copys a message m from folder n_{old} to n_{new}.

Table 3.1: The consistency critical IMAP commands with their arguments and descriptions based on RFC 3501.

work, we focus the consistency critical commands and therefore omit the modeling of the read commands.

In order to express the above mentioned consistency critical commands as operations on a CmRDT, we fist analyze the commands in detail based on the description in RFC 3501. We provide an overview of the commands, the arguments, and a short description in Table 3.1.

We note that the commands `CREATE`, `DELETE` and `RENAME` are rather self-explanatory. The only noteworthy exemption is the special handling of the INBOX folder, which cannot be deleted or renamed. The commands `SUBSCRIBE` and `UNSUBSCRIBE` are self-explanatory as well.

With the `APPEND` command, a user can add a message to a specific folder. In fact, this is the only command that adds new messages. In the RFC 3501 two optional arguments `flag list` and `date/time string` are mentioned. With the first argument, a user can assign a certain list of flags to the newly created message. The second argument can be used to assign a different date than the current date of the server. We note that most MUAs display the date that is part of the message's header to the user and the `date/time string` is typically ignored.

Specification 3 IMAP-CRDT (payload, *create*, and *delete*)

```
1: payload map $u : \mathcal{N} \to \mathcal{P}(\mathtt{ID}) \times \mathcal{P}(\mathcal{M})$
2:     initial $(\lambda x.(\varnothing, \varnothing))$
3: update create (foldername f)
4:     atSource
5:         let $\alpha = unique()$
6:     downstream $(f, \alpha)$
7:         $u(f) \mapsto (u(f)_1 \cup \{\alpha\}, u(f)_2)$
8: update delete (foldername f)
9:     atSource $(f)$
10:        let $R_1 = u(f)_1$
11:        let $R_2 = u(f)_2$
12:    downstream $(f, R_1, R_2)$
13:        $u(f) \mapsto (u(f)_1 \setminus R_1, u(f)_2 \setminus R_2)$
```

The `STORE` command provides the functionality to alter the message flags. This command is typically used to mark a message with a `deleted` flag. The sequence set can refer to a range of messages. In this case, the flags of multiple messages can be altered with one command. With `EXPUNGE`, all messages that have a *deleted* flag are permanently removed.

We note that the commands `EXPUNGE`, `STORE`, and `COPY` can only be executed in the *Selected* state. Hence, the foldername on which these commands operate is given implicitly. This will become important when we model those commands as operations on a CmRDT.

The IMAP CmRDT

We present the complete IMAP-CRDT in Specification 3, 4, and 5. We adhere to the presentation style that has been introduced by Shapiro et al. in [Sha+11b]. The definition of the payload is stated in line 1 and 2, as outlined in the beginning of this subsection. Next, we define the operations that represent the IMAP commands and begin with *create* and *delete* in Specification 3.

The desired result of *create* is to create an empty folder f. Therefore, a fresh and unique tag α is generated on the replica that initiates the operation **atSource**. Thereafter, the tag α is inserted into $u(f)_1$ and

Specification 4 IMAP-CRDT (*append*, *expunge*, and *store*)

```
14: update append (foldername $f$, message $m$)
15:     atSource ($m$)
16:         pre $m$ is globally unique
17:     downstream ($f, m$)
18:         $u(f) \mapsto (u(f)_1, u(f)_2 \cup \{m\})$
19: update expunge (foldername $f$, message $m$)
20:     atSource ($f, m$)
21:         pre $m \in u(f)_2$
22:         let $\alpha = unique()$
23:     downstream ($f, m, \alpha$)
24:         $u(f) \mapsto (u(f)_1 \cup \{\alpha\}, u(f)_2 \setminus \{m\})$
25: update store (foldername $f$, message $m_{\text{old}}$, message $m_{\text{new}}$)
26:     atSource ($f, m_{old}, m_{new}$)
27:         pre $m_{\text{old}} \in u(f)_2$
28:         pre $m_{\text{new}}$ is globally unique
29:     downstream ($f, m_{\text{old}}, m_{\text{new}}$)
30:         $u(f) \mapsto (u(f)_1, (u(f)_2 \setminus \{m_{\text{old}}\}) \cup \{m_{\text{new}}\})$
```

$u(f)_2$ remains untouched. This **downstream** part of the operation is executed at every replica. We denote an update of the map entry as $u(f) \mapsto (X, Y)$ where X and Y are the new sets that override the existing sets. We note that the map entries for the other foldernames remain unchanged.

We use the italic font to refer to an operation, e.g. create, to avoid confusion with the IMAP commands (in `typewriter`*).*

In contrast to *create*, the desired result of the *delete* operation is to make the folder non-existent. Hence, the content of $u(f)$ is removed at every replica. If we would define the downstream operation to be $u(f) \mapsto (\varnothing, \varnothing)$, then *create* and *delete* would no longer be commutative. Furthermore, the IMAP specification requires any *delete*(f) to be preceded by a *create*(f), aborting on IMAP protocol level if a client tries to remove a non-existing folder. This eliminates consistency issues when *delete*(f) and *create*(f) are issued concurrently. We note that the definitions of *create* and *delete* are very similar to the *add* and *remove* operations on the OR-Set, which has been introduced in [Sha+11b].

The operations *append*, *expunge* and *store* are defined in Specification 4. The *append* operation is very similar to the *create* operation, except that a message m is inserted into $u(f)_2$, and $u(f)_1$ remains unchanged. Another important difference is the atSource precondition. The IMAP specification states that each message is assigned a unique identifier called `UID`. We use this requirement to assure that no two *identical* messages are ever appended by different replicas, or even the same replica. We note that *identical* is not referring to the message's content. In practice, it is still possible to append two messages with identical content, although the `UID`s of the messages are in fact different. The existence of a unique message identifier can be assumed safely and is, in addition to the already mentioned part of the IMAP specification, common practice in the Maildir format, where messages can be identified by a unique filename within a folder.

The operation *store* is implemented in a similar fashion. The main purpose of *store* is to change the flags of a message m_{old}. We do not explicitly model the flags of a message. Instead, we insert the message m_{old} with updated flags as a new message m_{new} in $u(f)_2$ after deleting m_{old} from $u(f)_2$.

In contrast to the previous definitions, the *expunge* operation is rather counter intuitive. The deletion of a message, which has been marked with a `deleted` flag, is simply done by removing the message from $u(f)_2$. However, we decided that an additional tag must be inserted into $u(f)_1$ to avoid unexpected behavior in combination with concurrent *delete* operations. We illustrate this puzzle with the following example.

Two replicas r_1 and r_2 initially share the following state of a folder: $u(f) = (\varnothing, \{m_1\})$. The replica r_1 initiates a *delete* operation, resulting in an update of the local state at r_1 to be $u(f) = (\varnothing, \varnothing)$ and f is interpreted as non-existent, i.e. the complete folder is deleted. In the meantime, r_2 independently initiates an *expunge* operation that aims to delete m_1, resulting in the local state to be $u(f) = (\{t_{42}\}, \varnothing)$, i.e. an empty folder. At this point, it is unclear what result is actually desired, after the downstream operations are executed at both replicas. We decided, that the folder should be present as an empty folder at both replicas. Hence, according to the presented definitions, the resulting state is $u(f) = (\{t_{42}\}, \varnothing)$. In fact, our definition of the operations gives *create*, *append*, *store*, and *expunge* a precedence over *delete*, i.e. when manipulations of the folder f and a *delete*(f) are concurrently executed, the folder is never entirely deleted, only

Specification 5 IMAP-CRDT (*copy* and *rename*)

```
31: **update** *copy* (foldername $f_{old}$, message $m$, foldername $f_{new}$)
32:     **atSource** $(f_{old}, m, f_{new})$
33:         **pre** $m \in u(f_{old})_2$
34:         **pre** $u(f_{new}) \neq (\varnothing, \varnothing)$
35:         let $m'$ be a a globally unique copy of $m$
36:     **downstream** $(m', f_{new})$
37:         $u(f_{new}) \mapsto (u(f_{new})_1, u(f_{new})_2 \cup \{m'\})$
38: **update** *rename* (foldername $f_{old}$, foldername $f_{new}$)
39:     **atSource** $(f_{old}, f_{new})$
40:         **pre** $u(f_{old}) \neq (\varnothing, \varnothing)$ and $u(f_{new}) = (\varnothing, \varnothing)$
41:         let $R_1 = u(f_{old})_1$
42:         let $R_2 = u(f_{old})_2$
43:     **downstream** $(f_{old}, f_{new}, R_1, R_2)$
44:         $u(f_{new}) \mapsto (u(f_{new})_1 \cup R_1, u(f_{new})_2 \cup R_2)$
45:         $u(f_{old}) \mapsto (u(f_{old})_1 \setminus R_1, u(f_{old})_2 \setminus R_2)$
```

state visible at the initiation time of the *delete* operation is removed. Hence, we decided to pursue an *add-wins* semantic.

The modeling of the commands `RENAME` and `COPY` seems to be more laborious. Fortunately, both commands are rather similar to the already modeled operations. The `COPY` command can easily be modeled as a modified *append* operation, where the message must refer to a currently existing message in the selected folder. Moreover, the destination, i.e. the folder where the message should be copied to, must exist. Hence, in contrast to *append*, *copy* has three input parameters: a foldername that represents the folder that is currently selected, a message, and a foldername that represents the folder where the message should be copied to. The semantics, as shown in Specification 5, are, except for the additional preconditions, identical with *append*. We note that the `COPY` command does not modify the original message and that is why the content of the selected folder $u(f_{old})$ remains unchanged.

In Specification 5 we also present our modeling of the *rename* operation. In essence, *rename* is a combination of *create* and *delete*, except that the content of the folder is copied to the new foldername. As precondition we add that the source folder must exist, but it can be empty (see line 40). Moreover, the destination folder must not exist. We note that both requirements are related to the specification of RENAME in RFC 3501 and are technically not needed to achieve convergence of replicas. The remaining part of the definition follows the intuition of *create* and *delete*. We first store the content of the folder f_{old} at the source replica in R_1 and R_2. This content is then used to modify the map at all downstream replicas. We note that it is necessary to use unions and set complements to achieve convergence. Particularly, it would not be commutative if $u(f_{old})$ would be set to $(\emptyset, \emptyset)$ and $u(f_{new})$ would be set to (R_1, R_2). We discuss the design decisions and the implications for the application behavior in the end of this subsection.

The remaining commands SUBSCRIBE and UNSUBSCRIBE are easy to model as and Observed-Removed Set CRDT. As outlined in Section 3.2, the OR-Set already features common set semantics. In this case, the payload of the OR-Set would be foldernames that could be added or removed with the corresponding commands. In order to reduce the complexity of this work, we omit an explicit modeling of both commands.

Design Decisions and Discussion

The major problem of designing an IMAP-CRDT is the lack of well defined behavior in presence of concurrent updates. The desired behavior of single IMAP commands is perfectly described in RFC 3501, whereas the presence of concurrent updates can lead to unexpected state. Certain commands, that have reasonable consequences on the state of the local replica, may not even be allowed to be applied at another replica, because the remote state may have changed in the meantime. The guarantee of CRDTs, i.e. that replicas converge, only assures that no two replicas end up with different abstract states. With the IMAP-CRDT we present an approach that follows two design principles: (i) IMAP commands on a local replica must behave as expected according to RFC 3501 and (ii) concurrent updates may not result in any *damage* or obviously undesired state of the account. To this end, we decided for an *add-wins* strategy in order to avoid undesired loss of data.

The proposed *add-wins* strategy comes to the price of increased metadata that needs to be managed. In the presented definition we create a new tag for each deleted message of an *expunge* operation.

These tags are, in the current design, only removed by a *delete* operation, which is typically not executed as long as the user holds interest in the folder. To overcome this issue, some metadata could be deleted after a certain *stable* state has been reached. For example, Baquero et al. introduced the notion of log compaction through *causal stability information* in [BAS14]. An alternative decision would be to give *delete* precedence over the other operations. In this case, less metadata would be required to process state information. However, the application behavior in the presence of concurrent updates seems undesired. For example, in case of a concurrent *append* and *delete* operation on the same folder, the message that was added by the *append* operation would be deleted with the folder and be lost forever. We note that our IMAP-CRDT requires causal-order delivery and we omit this precondition in every update operation for the sake of simplicity.

3.3.4 *Mechanical Verification*

After presenting the design of our IMAP-CRDT, we aim to provide the certainty that the promised guarantees, i.e. the convergence of replicas, actually hold under realistic assumptions. To this end, we present an Isabelle/HOL formalization that is based on the recently published CRDT verification framework by Gomes et al. [Gom+17a]. The formalization includes a network model which uses the basic axioms of an *asynchronous unreliable causal broadcast*, which we will introduce throughout this subsection. The proof document and the Isabelle proof implementation[2] is published in the Archive of Formal Proofs [JOL17b].

System Model and Network Assumptions

With the verification of our IMAP-CRDT we show that *convergence* is guaranteed, i.e. two replicas have equivalent abstract states if both received the same set of operations. Hence, in order to reason about the convergence, we need to introduce the notion of a network first. We base our system model on the model that is common when working with CRDTs, which we already introduced in

2 https://www.isa-afp.org/entries/IMAP-CRDT.html (BSD License)

Section 3.2. A mechanical verification, however, requires a more detailed description in order to be approved by a theorem prover like Isabelle. Therefore, we reuse the relevant parts of the CRDT verification framework [Gom+17b] that are needed to verify that our IMAP-CRDT guarantees strong eventual consistency.

We start with a rather common definition of an asynchronous distributed system that consists of an unbounded number of processes that communicate over channels where each process is identified by a unique id. Asynchronous means that there are no bounds on the relative speed of message exchange and process execution. We can make no assumptions about the timing and messages can be delayed for an arbitrary time. Moreover, messages can be lost or processes can crash; therefore we consider our system *unreliable*. There are no further assumptions on the network topology.

Processes communicate with each other via messages. From the perspective of a process, there are only two types of messages: *broadcast* or *deliver*. A *broadcast* message has exactly one sender and is transmitted to all processes in the network. A *deliver* message can be seen as a message that was received from the network and delivered to the application. We note that the notion of a *broadcast* message is a common model in distributed systems. In practice, there are established ways to implement a broadcast network on top of unicast, e.g. with an overlay network over a fully connected graph or a gossip protocol. Each process stores a history of messages as an ordered list. Hence, messages inside the *event log* of a process are numbered and duplicates are excluded.

With the above mentioned components, Gomes et al. compose a definition of a *network* and make the following assumptions on the event log of a process:

1. For any *deliver* message in the event log of a process, there exists a process that has the corresponding *broadcast* message in the event log.

2. If a *broadcast* message is in the event log of a process, there exists a corresponding *deliver* message in the event log of the same process with a higher index.

3. Each message is globally unique.

We note that the introduced assumptions are typical for describing and modeling asynchronous distributed systems. The first assumption simply states that a message, that is delivered to somewhere, must have had a sender and the message was not created out of nowhere. The second assumption is also very standard. For easier reasoning it is more convenient to have the notion that a message is *delivered* locally, like it would be delivered to another process, but without actually using the network. The third assumption is important to distinguish messages in our system. In practice, this is typically achieved by adding a process identifier or a UUID to the message. Baquero et al. use the vector clocks, as a standard mechanism to track causality of messages, as message identifier, because a vector clock is globally unique [BAS14].

In fact, most CmRDTs require messages to be delivered in causal order. Hence, messages in the event log of the processes must be in harmony with the happened-before relation $\prec$. Together with the notion of the *broadcast* and *deliver* messages, Gomes et al. conducted the following rules for a message m_1 that happened-before m_2. They say $m_1 \prec m_2$ if any of the following rules hold.

1. m_1 and m_2 were broadcast messages by the same process, and m_1 was broadcast before m_2.
2. The process that broadcast m_2 had delivered m_1 before it broadcast m_2.
3. There exists some message m_3 such that both $m_1 \prec m_3$ and $m_3 \prec m_2$.

Causal-order delivery is typically achieved by using vector clocks. In case the vector clock of a received message indicates a missing message that happened-before, the delivery of the received message to the application is delayed until the missing message was received. This implementation is rather standard in distributed systems and we omit a detailed discussion. Later in this section we present related work that aims to reduce the introduced overhead of using and comparing vector clocks without loosing causality information.

Having defined a network that supports causal-order delivery; the only missing piece is the definition of the operations in the network. Therefore, the messages are further refined to be a pair of an identifier and an operation with its parameters. Furthermore, an initial state and an interpretation function of the operation must be defined. The interpretation function defines the semantics of a *downstream* operation on an arbitrary but fixed state. Because the IMAP-CRDT has further requirements to the operations, namely the *atSource* preconditions, the model only considers broadcast messages that are in accordance to these constrains.

Isabelle Proof

With the defined system model from the previous paragraphs we are able to refine our ultimate goal of this subsection: the implementation of a mechanically checked proof of the convergence. We find that having the certainty of an automatically checked proof is worth the additional work to convince the proof system. Especially the presence of concurrent updates makes it hard to reason about correctness properties without missing edge-cases or even more fundamental problems. Fortunately, Gomes et al. provided a sophisticated framework for verifying strong eventual consistency in Isabelle [Gom+17a]. Besides that we invested some work into developing our own system model in Isabelle[3], we chose to adapt to their framework and contribute the IMAP-CRDT verification as a further example to the Archive of Formal Proofs.

The framework of Gomes et al. is composed out of Isabelle *locales* that provide encapsulated definitions and properties of the above introduced model of a distributed system. In our proof, we instantiate these locales and implement the IMAP-CRDT in Isabelle. The already proven lemmas from the locales greatly simplify our proof and allow us to focus on the properties of interest. Before we present our proof, however, we state the main theorem that we want to verify.

Theorem 1. Having an *asynchronous unreliable causal broadcast network*, the abstract states of two processes that received and applied the same set of operations are equivalent, iff:

3 We were actually half-way through when we first identified some issues with the model. Fortunately, Peter Zeller and Mathias Weber from the Softech group at Universität Kaiserslautern helped us debugging the theories. Coincidentally, Gomes et al. published their framework five days prior to my Kaiserslautern visit; right on time to be used in this thesis.

1. All concurrent operations commute.
2. Applying an operation never fails.

The used framework offers a translation of this theorem into Isabelle and the authors show that the theorem holds in the above mentioned system model. This contribution significantly reduces the complexity of our proof and we are thankful that the framework was published right before we started investigating the correctness of our CRDT. Hence, the remaining task for us is to show that the concurrent operations of our IMAP-CRDT commute and that an apply operation never fails. However, proving both properties is a laborious task. In the following paragraphs we will give an overview over our Isabelle implementation. For further details we refer to the source code and the documentation [JOL17b].

Implementation of the IMAP-CRDT

As the first step, we define the IMAP-CRDT in Isabelle before proving any lemmas. Therefore, we start by defining the operations. In order to further reduce the complexity of the proof, we omit implementing the operations *copy* and *rename*. Since those operations are build on the primitives of *create* and *delete*, we expect no obstacles other than the additional implementation complexity.

```
datatype ('id, 'a) operation =
  Create "'id" "'a" |
  Delete "'id set" "'a" |
  Append "'id" "'a" |
  Expunge "'a" "'id" "'id" |
  Store "'a" "'id" "'id"
```

We note that we introduced two types `'a` and `'id`. In this case, the type `'a` represents the type of the foldernames and `'id` represents the metadata that is created by most of the operations. This metadata is, for example, heavily used in the definition of *create* to generate a globally unique tag.

The payload is, as in the original specification in Specification 3, defined as a map that projects foldernames to a tuple of identifier sets. Hence, the defined payload $u : \mathcal{N} \to \mathcal{P}(\mathtt{ID}) \times \mathcal{P}(\mathcal{M})$ is now translated into Isabelle with the help of a type synonym.

```
type_synonym ('id, 'a) state = "'a ⇒ ('id set × 'id set)"
```

At this point we note the first difference between the specification and the implementation. To reduce the complexity of the implementation, we no longer distinguish between the sets `ID` and $\mathcal{M}$. This makes the formalization slightly easier, because less type variables are introduced. While the original payload separates between metadata and messages, we treat messages the same way as the identifiers. This may seem counter intuitive, but the messages, i.e. the content of a folder, must be globally unique as well. One can see a mail message in our Isabelle implementation as a unique reference or a *pointer* that refers to the message's content stored on the hard drive. In fact, in our later implementation we use the filename of a message as unique identifier, but this is rather specific to the Maildir format. However, to avoid confusion with the use of the word *message*, we stick to metadata and filenames as content of a folder.

Next, we implement the operations in Isabelle. Therefore, we first introduce a function `op_elem` that is used to extract the foldername from an operation. For example, `op_elem` of a *create* operation would refer to the folder that is to be created. In contrast to that, `op-elem` of a *append* operation would refer to the folder where a filename should be inserted to.

```
definition op_elem :: "('id, 'a) operation ⇒ 'a" where
  "op_elem oper ≡ case oper of
    Create i e ⇒ e |
    Delete is e ⇒ e |
    Append i e ⇒ e |
    Expunge e mo i ⇒ e |
    Store e mo i ⇒ e"
```

After that, we implement the operations more or less directly from the Specification 3, 4, and 5 Here, Isabelle's update of the map, denoted as :=, translates to the $\mapsto$ arrow in the specification. We use a simple let-in to define all operations in one function. The only notable difference is the implementation of the *delete* operation that no longer handles two sets R_1 (the set of metadata that should be removed) and R_2 (the set of filenames that should be deleted), but only one set that is essentially a union of both sets. We will show later that the set that represents the *metadata* and the *filenames* are always disjoint.

```
definition interpret_op :: "('id, 'a) operation
  ⇒ ('id, 'a) state ⇀ ('id, 'a) state" where
  "interpret_op oper state ≡
```

```
    let metadata = fst (state (op_elem oper));
        files = snd (state (op_elem oper));
        after  = case oper of
          Create i e ⇒ (metadata ∪ {i}, files) |
          Delete is e ⇒ (metadata - is, files - is) |
          Append i e ⇒ (metadata, files ∪ {i}) |
          Expunge e mo i ⇒ (metadata ∪ {i}, files - {mo}) |
          Store e mo i ⇒ (metadata, insert i (files - {mo}))
    in  Some (state ((op_elem oper) := after))"
```

We define the preconditions of the *atSource* part in a separate function that we will later use to instantiate the system model. These *valid-behaviours* translate quite well from the specification.

```
definition valid_behaviours :: "('id, 'a) state ⇒
 'id × ('id, 'a) operation ⇒ bool" where
  "valid_behaviours state msg ≡
    case msg of
      (i, Create j e) ⇒ i = j |
      (i, Delete is e) ⇒ is = fst (state e) ∪ snd (state e) |
      (i, Append j e) ⇒ i = j |
      (i, Expunge e mo j) ⇒ i = j ∧ mo ∈ snd (state e) |
      (i, Store e mo j) ⇒ i = j ∧ mo ∈ snd (state e)"
```

At this point we note that we simply can use the fact that messages in our system are globally unique, as outlined in the list of assumptions on the network model, to guarantee that the metadata or messages of operations are globally unique. Therefore, we reuse a trick that we have seen in the implementation of Gomes et al. and simply set the globally unique message identifier as metadata or filename. This is realized by setting the message identifier `i` to the globally unique piece of metadata `j`. For *delete*, we require that the sender sends all current references to metadata and filenames when issuing a broadcast message. Moreover, *expunge* and *store* require that the message that should be altered or deleted must currently exist in the folder. We note that the specification lists more preconditions to the operations, e.g. *create* also requires that the folder that should be created must not already exist. We discovered, that this is a purely IMAP specific requirement and technically not needed to guarantee convergence of replicas. Hence, we omit these preconditions in the Isabelle implementation; making the proof result even stronger.

With the above introduced implementation we are ready to instantiate the system by using the locale from the framework and defining the initial state as a lambda function that projects everything to $(\emptyset, \emptyset)$.

```
locale imap = network_with_constrained_ops _
   interpret_op
   "λx. ({},{})"
   valid_behaviours
```

Commutativity of Concurrent Operations

Operation-based CRDTs require all concurrent operations to commute in order to ensure convergence. Therefore, we begin our verification by proving the commutativity of every combination of possible concurrent operations.

Gomes et al. base the definition of the commutativity of operation on the *Kleisli arrow composition*. In essence, for two operations x and y, the result of the composition of both operations, denoted as $\langle x\rangle \rhd \langle y\rangle$, is an operation that first applies x on some state, and then applies y on the result. The composed function fails if one of the operations x or y fails or is not defined. The Isabelle translation uses the *monadic bind operator* to express this composition:

```
definition kleisli ::
  "('b ⇒ 'b option) ⇒ ('b ⇒ 'b option) ⇒ ('b ⇒ 'b option)" where
  "f ▷ g ≡ λx. (f x ≫= (λy. g y))"
```

With the Kleisli arrow, we have a nice way to express the commutativity of operations, as we exemplify with two *create* operations in the following Isabelle code:

```
lemma (in imap) create_create_commute:
  shows "⟨Create i1 e1⟩ ▷ ⟨Create i2 e2⟩
        = ⟨Create i2 e2⟩ ▷ ⟨Create i1 e1⟩"
```

Initially, we used *nitpick*, Isabelle's counterexample generator, to identify corner cases in our implementation. We found out that most of the combinations are truly commutative, like the above showed *create* operations. This is, in fact, a stronger property than we actually need, because we only require *concurrent* updates to commute. Unfortunately, as we will point out in the following paragraphs, not all operations commute without adding extra assumptions.

```
lemma (in imap) create_delete_commute:
  assumes "i ∉ is"
  shows "⟨Create i e1⟩ ▷ ⟨Delete is e2⟩
        = ⟨Delete is e2⟩ ▷ ⟨Create i e1⟩"
```

Here, the *create* and *delete* operations only commute, if we require that the metadata that is inserted by *create* is not part of the remove-set of a *delete* operation, hence $i \notin is$. The necessity of this assumption implies that we need to put more effort into showing that this assumption holds in case both operations are concurrent. In fact, the remaining proof is mainly about showing that in this case, the *create* operation must have happened before *delete*, because otherwise we would invalidate the previously defined *valid-behaviours* and the *atSource* precondition of *create*. Before we continue to prove this happened-before relation, we show the remaining puzzles that we identified for all operations.

Alongside with the previous illustration of the assumption that $i \notin is$, which is actually needed for all combinations of operations with *delete*, we identified two more interesting corner-cases:

```
lemma (in imap) append_store_commute:
  assumes "i1 ≠ mo"
  shows "(⟨Append i1 e1⟩ ▷ ⟨Store e2 mo i2⟩)
        = (⟨Store e2 mo i2⟩ ▷ ⟨Append i1 e1⟩)"
```

```
lemma (in imap) store_store_commute:
  assumes "i1 ≠ mo2" and "i2 ≠ mo1"
  shows "(⟨Store e1 mo1 i1⟩ ▷ ⟨Store e2 mo2 i2⟩)
        = (⟨Store e2 mo2 i2⟩ ▷ ⟨Store e1 mo1 i1⟩)"
```

Both of the above shown Isabelle listings show an additional assumption that is introduced by combinations of *expunge* and *store*. In this case, an operation must not create a filename for a message that is referred to by a possibly concurrent *store* operation. More concrete, a message that is put into a folder with *append* cannot be the same message that is concurrently modified by *store*.

We can summarize the identified additional assumptions as *critical conditions* that need to be satisfied by all operations that follow the predefined *valid-behaviour*.

- The metadata of a *create* and *expunge* operation, or the filenames of an *append* and *store* operation, are never in the removed-set of a concurrent *delete* operation.
- The filename of an *append* operation is never the filename that is deleted by a concurrent *store* or *expunge* operation.
- The filename inserted by a *store* operation is never the filename that is deleted by a concurrent *store* or *expunge* operation.

The identified conditions obviously hold in our system, because an item that has been inserted by one operation cannot be deleted by a concurrent operation. It simply cannot be present at the time of the initiation of the concurrent operation.

With the above presented assumptions we were able to show that all combinations of operations, which condense to 15 individual cases, are in fact commutative. In all cases, the definitions of the interpretation function and the Kleisli arrow composition were sufficient to prove the lemmas.

Critical Conditions hold for Concurrent Operations

The remaining task is to prove that the identified critical conditions hold. For combinations with *delete*, the proof is similar to what Gomes et al. used to prove the correctness of the OR-Set. Hence, in order to show that $i \notin is$ and $i_1 \neq mo$ hold, we define a list of metadata, called `added_ids`, that represents a list of all metadata that was ever inserted by the delivered *create* and *expunge* operations in the event log of a process. Similar to that we define a list of filenames that are created by *append* and *store*.

```
definition (in imap) added_ids :: "('id × ('id, 'b) operation) event list
⇒ 'b ⇒ 'id list" where
  "added_ids es p ≡ List.map_filter (λx. case x of
    Deliver (i, Create j e) ⇒ if e = p then Some j else None |
    Deliver (i, Expunge e mo j) ⇒ if e = p then Some j else None |
    _ ⇒ None) es"
definition (in imap) added_files :: "('id × ('id, 'b) operation) event
list ⇒ 'b ⇒ 'id list" where
  "added_files es p ≡ List.map_filter (λx. case x of
    Deliver (i, Append j e) ⇒ if e = p then Some j else None |
    Deliver (i, Store e mo j) ⇒ if e = p then Some j else None |
    _ ⇒ None) es"
```

We can show that the event log of a process contains only a subset of the metadata and filenames of `added_ids` or `added_files`. This information is used to show that metadata and filenames that are referenced by one operation must be created by a preceding operation that *happened-before*. We exemplify this step with the following lemma for *append* and *store*. Here, `hb` translates to *happened-before*.

```
lemma (in imap) concurrent_append_store_independent_technical:
  assumes "i = mo"
    and "(i, Append i e) ∈ set (node_deliver_messages xs)"
    and "(r, Store e mo r) ∈ set (node_deliver_messages xs)"
  shows "hb (i, Append i e) (r, Store e mo r)"
```

In this lemma we show that if a *store* and *append* operation are in the event log of a process and *store* references to the filename of *append*, hence $i = mo$, then *append* must have happened before *store*. Moreover, we show that if there is such a reference, both operations work on the same folder:

```
lemma (in imap) append_store_ids_imply_messages_same:
  assumes "i = mo"
    and "(i, Append i e1) ∈ set (node_deliver_messages xs)"
    and "(r, Store e2 mo r) ∈ set (node_deliver_messages xs)"
  shows "e1 = e2"
```

Both of the above shown lemmas imply that the used assumption $i \neq mo$ is safe to be made for concurrent operations, which represents the remaining corollary that needs to be shown for this case:

```
corollary (in imap) concurrent_append_store_independent:
  assumes "¬ hb (i, Append i e1) (r, Store e2 mo r)"
    and "¬ hb (r, Store e2 mo r) (i, Append i e1)"
    and "(i, Append i e1) ∈ set (node_deliver_messages xs)"
    and "(r, Store e2 mo r) ∈ set (node_deliver_messages xs)"
  shows "i ≠ mo"
```

We note that this is just one example of the previously introduced *critical conditions* that must hold in order to show convergence. In fact, in addition to the presented lemmas for *append* and *store*, there are 7 more cases that were considered in our proof.

Convergence and Strong Eventual Consistency

At this point we are able to connect the remaining pieces in order to show the final theorem. As mentioned in the abstract description of Theorem 1, we must not only show the commutativity of concurrent operations, but also that applying an operation never fails. Fortunately, the latter is rather easy to show, since our IMAP-CRDT implementation in `interpret_op` never returns `None`. Hence, we can easily show the following lemma:

```
lemma (in imap) apply_operations_never_fails:
  assumes "xs prefix of i"
  shows "apply_operations xs ≠ None"
```

In the above and following code, `i` and `j` are two event logs, i.e. lists of operations of a process, that are fixed by the final locale that we instantiated. With "`xs prefix of i`" we show that the remaining properties hold for all prefixes of the event log. This will be particularly important when proving strong eventual consistency. Here, equivalent abstract states are guaranteed, if two processes have seen the same set of operations, which must also hold for every subset (or prefix).

The remaining task to show convergence is to conduct a lemma that proves concurrent operations to commute. With the above introduced lemmas, namely the commutativity of operations and the certainty that the identified *critical conditions* always hold, this lemma is relatively easy to show, even though all of the 15 combinations of operations must be considered separately.

```
lemma (in imap) concurrent_operations_commute:
  assumes "xs prefix of i"
  shows "hb.concurrent_ops_commute (node_deliver_messages xs)"
```

Ultimately, we show the convergence theorem. At this point, the used framework already offers the necessary steps to combine both of the above mentioned lemmas. In fact, the final proof of the theorems, i.e. the convergence, has not been modified by us and fully relies on the earlier introduced lemmas.

```
theorem (in imap) convergence:
  assumes "set (node_deliver_messages xs)
           = set (node_deliver_messages ys)"
    and "xs prefix of i"
    and "ys prefix of j"
  shows "apply_operations xs = apply_operations ys"
```

3.4 *pluto*: THE PLANETARY-SCALE IMAP SERVER

I would like pay tribute for the design and implementation of pluto to Lennart Oldenburg, who contributed phenomenal insights to this chapter.

In this section we present our prototype of a distributed IMAP server *pluto* that implements the IMAP-CRDT. We will put particular focus on the design decisions and the implied consequences for our approach. We omit concrete implementation details, because the source code, as well as additional documentation and installation instructions, is available in the project repository and licensed under GPLv3 or later[4].

3.4.1 *Selected Features and Limitations*

We begin the presentation of our prototype by defining the features of interest and its limitations. In contrast to software products that are designed to be used in the industry, our prototype mainly is a proof of concept that the conceptual benefits, i.e. using and replicating more application state in the logic tier, can be realized and pay off in practice. From this aspiration, we derive the necessary features that we need to evaluate our prototype against *Dovecot*, the de facto standard IMAP server software.

The very first decision includes the choice of the IMAP commands that our prototype must provide. Since we are mainly interested to analyze the opportunities and disadvantages of storing more state in the logic tier, we again focus on the consistency-critical commands, i.e. the write commands. That is why we omit the implementation of the *read* commands like `SEARCH` or `FETCH`, similar to the earlier presented choice were we omitted those commands when designing the IMAP-CRDT. Hence, our prototype currently supports the following IMAP commands:

- `CREATE`, `DELETE`, `APPEND`, `EXPUNGE`, `STORE`

In order to implement the aforementioned commands in a meaningful way, certain other commands are necessary as well. For example, `STORE` can only be executed if a folder has been selected by `SELECT`. For this reason, the following commands are also implemented in the prototype:

- `CAPABILITY`, `STARTTLS`, `SELECT`, `LIST`, `LOGOUT`

We note that these commands do not alter the folders or the messages and are therefore considered as *read* commands. We judge the aforementioned commands as sufficient to show that we are able to transfer our approach from theory to practice. Therefore, we omit implementing the remaining consistency-critical commands,

4 `https://github.com/go-pluto/pluto` licensed under GPLv3 or later.

i.e. `COPY` and `RENAME`, as well as the remaining read commands from RFC 3501.

While the aforementioned commands determine the set of features from client's perspective, we also identified the following important features from an operators point of view:

3-tier Architecture: The prototype must be composed of different individual components that can be arranged in a traditional 3-tier architecture.

Security and Encryption: Since we later conduct an evaluation based on public cloud infrastructure, all communication channels must be encrypted.

User Authentication: User information must be provided in the common ways, i.e. over a username/password file or a database like PostgreSQL.

Fault Tolerance: Connections to failed components that are not mission critical must be rerouted to working components; similar to a *failover*.

Configurability: The addresses of the replicas, as well as the partitions and the routing of users requests, must be configured with a simple configuration file.

We note that the above mentioned features are neither a precise nor a complete list of requirements. They rather provide an overview of what desired features must be considered in order to deliver a prototype that can withstand a fair evaluation. In addition to the mentioned features, the requirements for using our IMAP-CRDT must be considered as well. Hence, our prototype must be able to form a distributed system with arbitrary many replicas, in contrast to *Dovecot-dsync*, which only allows a maximum of two replicas.

As the only system specific requirement that is implied by using CmRDTs, our prototype must enforce a causal order of the messages and therefore offer a causal-order broadcast. We will address this challenge when we explain the design of our prototype in the following subsection.

In summary, we aim to provide a working prototype that offers reduced IMAP functionality but with a sophisticated architecture that enables configurable deployments and security, as we would expect it from a system of a certain scale.

3.4.2 *Architecture and Design*

The main point of our prototype is to demonstrate that it is feasible to store and replicate more state in the logic tier using our IMAP-CRDT. From this purpose, we derive a tailored architecture.

In Figure 3.6 we show the interplay of *pluto*'s components in a 3-tier architecture. In the presentation tier, we implement a *distributor* similar to a proxy. The workers in the logic tier are, in contrast to a *traditional* system that follows the service statelessness principle, stateful. That means, that mailbox accounts of a particular range of users are stored in this tier. In addition to that, the *storage* component of *pluto* can be seen as a replica of the state that is already stored on the worker.

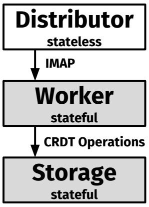


Fig. 3.6: The pluto architecture.

Next, we describe the particular tasks of these three components in depth:

DISTRIBUTOR: In any *pluto* deployment, an IMAP request enters the service at a stateless distributor node. Any request initiated via an unencrypted connection will get dropped, ensuring that authentication credentials transmitted as part of an ordinary IMAP session are only ever sent over a TLS connection. The distributor node handles a session as far as the authentication procedure was successful. For any further request, the worker node responsible for the particular partition of users is determined, and all traffic is proxied to this node. Should the determined worker node be unavailable, for whatever reasons, a *failover* to the storage node is performed, which accepts the proxied IMAP traffic in place of the worker node.

WORKER: As soon as requests of a regular IMAP session reach a worker node, the response is computed based on the state stored on the worker and immediately sent back to the client. In case the IMAP request changes the application's state, for example with a `CREATE` command, the **atSource** preconditions are checked and additional information are computed. Thereafter, the **downstream** operation is sent to all connected replicas and the storage. The fact that there may be more than one worker that answers IMAP requests for a particular user and the presence of the storage makes *pluto* a multi-leader replication system.

STORAGE: All mailbox accounts are securely stored in *pluto*'s storage component. Idealy, the storage runs on reliable and tailored hardware where the integrity of the data is the most important priority. The storage accepts **downstream** updates from the worker nodes and applies them on the local state. Only in case that all other worker nodes for a particular range of users are unavailable, the storage can accept incoming IMAP requests; acting as the last working copy of the state.

The separation into the three components, and especially the decision to make the worker component stateful, enables interesting options to configure the *pluto* system. Some of these configurations and the promising opportunities are already discussed in the preliminaries is Chapter 2, without the concrete impact on an IMAP service. However, with our prototype we can explore these options and see whether the opportunities are worth considering.

The easiest configuration to consider is to use these three components without any further replication or partitioning on three different machines. In this setup, which is also represented in Figure 3.6, the system would benefit from improved response times for *write* requests, because the response can be immediately computed based on the state that is stored in the worker without passing the request to the storage. Moreover, this configuration also provides increased fault tolerance in contrast to a traditional *Dovecot* deployment, because a temporary connection loss to the storage can be tolerated.

The most advanced configuration to consider is when the workers form a distributed system and the users are partitioned (or sharded) into different ranges. We illustrate this configuration in Figure 3.7, where we show a *pluto* system where the mailbox accounts are divided into two groups, which are then replicated over three replicas. In this setup, every replica of the worker can continue to operate independently in case of a network disruption or partitioned replicas. The verified guarantees of the IMAP-CRDT imply, that all replicas eventually reach the same state when partitions have healed and messages can be exchanged. We note that the storage in the mid-

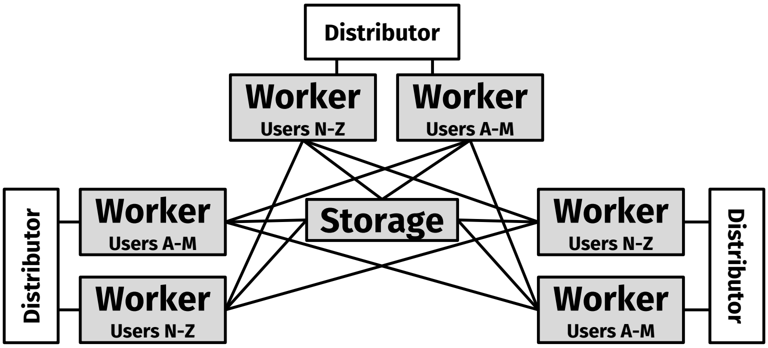


Figure 3.7: A more advanced *pluto* configuration with partitioning and replication.

dle of Figure 3.7 ideally has no impact on the performance and only represents a safe location where the data is persisted on more trustworthy hardware.

We will discuss the benefits of the enabled multi-leader replication in the logic tier in the end of this chapter, after we evaluated our prototype in different configurations.

3.4.3 *Implementation Decisions*

As the major difference to other IMAP server software, we implemented the two required components, the IMAP-CRDT and a reliable causal-order broadcast of update messages, as parts of *pluto*. The implementation of the IMAP-CRDT is slightly adjusted in order to achieve a better performance. In our implementation of the IMAP-CRDT, we assign each user an OR-Set, called *structure*, that represents the user's abstract mailbox state. The main difference to our theoretical model in Specification 3 is, that the map $u(f)$ for a mailbox folder f is modeled as a set of *value-tag* pairs for which the *value* element is always set to f. As an example, we consider a mailbox folder `uni`, on which an *append* operation was executed. Assume, that the state according to Specification 3 looks like $u(\texttt{uni}) \mapsto (\{\alpha\},\{m\})$. We can infer that the *create* operation for `uni` created tag α in $u(\texttt{uni})_1$ and the *append* operation put m into $u(\texttt{uni})_2$. In our *structure* OR-Set, this is represented as

$\{(\texttt{uni}, \alpha), (\texttt{uni}, m)\}$. Thus, in *pluto* we do not distinguish between metadata and message tags, similar to the choice we have made in the Isabelle verification. Any update to *structure* is followed by a file system `sync` operation on an associated log file on stable storage. This ensures that nodes can precisely reconstruct the internal representation of the user's mailboxes in case they have crashed or were restarted.

An update on a source replica triggers a message to all downstream replicas in order to reproduce it on their state. In *pluto*, worker and storage nodes are grouped into subnets that exchange updates for a particular partition of users. Considering a planetary-scale deployment with workers in Europe, the US, Asia, and the storage in Australia, the subnet for a worker `eu1` in Europe might contain `us1`, `asia1`, and `storage`. Each downstream message from `eu1` is sent to all other nodes in its subnet.

As the IMAP-CRDT is based on the operation-based OR-Set, we require these messages to be part of a reliable causal-order broadcast, ensuring that they are delivered to the application exactly once and with no causally-preceding ones missing. To this end, we maintain vector clocks [Mat88; Fid88] for each subnet. Send queue, receive queue, and vector clock are again `sync`'ed into associated files on any update. To reduce replication lag, we do not send messages individually but transfer the current send log as a whole in a defined interval.

The use and the implementation of vector clocks may leaves room for improvement, as it is surely costly to compute and compare a vector of n clocks for each operation on n replicas. While the concepts of vector clocks is relatively simple and commonly applied in distributed systems, there are certain optimizations to achieve a better performing causal communication. We discuss alternative approaches in the related work part of this chapter.

For the internal communication between the components, we use the Google's open-source implementation for remote procedure calls gRPC [gRP18]. Hence, we gain a sophisticated communication layer that enables fast and reliable interaction of the components.

3.5 EVALUATION

In the previous sections we introduced our IMAP-CRDT, verified its convergence guarantees, and presented our prototypical implementation of a distributed and CRDT-driven IMAP server *pluto*. The remaining open question is, whether our prototype can play of the conceptual benefits compared to the existing approaches or not. To this end, we conduct several experiments that compare our prototype *pluto* against the de facto standard IMAP server *Dovecot*.

As we pointed out in Section 2.2.2, enabling multi-leader replication in the logic tier mainly serves two purposes: to increase the performance and to increase the fault tolerance. For now, we leave the analyzation of the fault tolerance out of scope of this evaluation. A detailed discussion of what faults and failures can be tolerated and the implied design decisions is presented later in this chapter. The impact of the multi-leader replication on the performance, however, will be analyzed in depth in this section.

We find that the impact on the performance can be best analyzed in scenarios where there is high latency between the replicas. The best practical example for these scenarios are geo-replicated systems, i.e. systems that serve clients from different continents. In such systems, replication is used to reduce the response times for requests by placing a replica at a location near the client.

In this section we aim to evaluate our prototype against *Dovecot* in a geo-replicated setting. Therefore, we try different setups and identify the pitfalls of geo-replication, i.e. we show how naive applied replication mechanisms fail magnificently. Thereafter, we introduce our IMAP-Benchmark that generates write-intensive workloads. Moreover, we present the test bed that is based on the most advanced infrastructure at the time of developing the prototype. Ultimately, we conduct multiple IMAP experiments at planetary-scale and present our results.

3.5.1 *The Pitfalls of Geo-Replication*

Before we begin the evaluation of our prototype, we need to identify the best setup of the reference system, i.e. *Dovecot*, to create a *fair* comparison. As mentioned, we focus on geo-replicated systems.

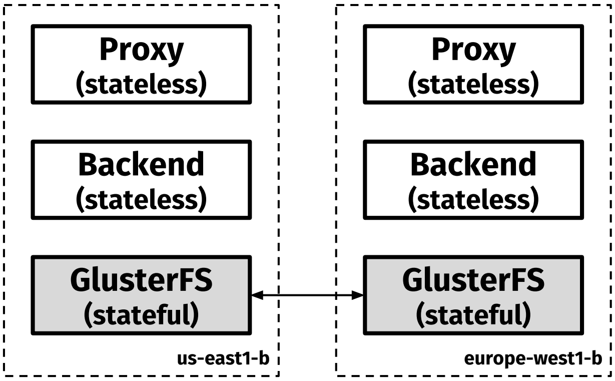


Figure 3.8: The naive approach to geo-replicate *Dovecot*.

Hence, it is our task to find the best setup for a *Dovecot*-powered IMAP service that aims to serve clients from rather distant locations.

In Section 3.3.1 we introduced different configurations for *Dovecot*, as well as the extension *dsync*, that enables a 2-way replication. For now, we focus on the *traditional Dovecot* configurations. However, we evaluate *dsync* in depth later in this section.

The most standard configuration of *Dovecot*, that does not involve any advanced replication, is composed of a proxy that redirects the clients' requests to a *Dovecot* backend which stores the mailbox accounts in a shared file system like GlusterFS. Such system, which we already introduced in a previous section, is generally not designed to achieve low response-times for users with a huge distance to the service, e.g. from another continent. However, the advantage of this configuration is, that the system is easy to maintain, since all components are most likely in the same data center. Client requests from different continents would be routed through the Internet and may suffer from comparable high latencies. We note that this setup is actually not the worst to consider. Later in this section we will use this system as our *baseline* for the other experiments.

From this simple setup we can derive ostensible optimizations to improve the response-time performance. Now, the most obvious answer is to replicate components of the system and place them near the clients. One obvious solution would be to replicate the entire setup. We illustrate this setup in Figure 3.8.

In the figure we see that the stateless components, i.e. the proxy and the backend, are deployed in two different regions. This is is easy to achieve, because no mission critical state is held in these tiers. The critical part of the application state is, in fact, held in the data tier, i.e. the GlusterFS. One could easily configure GlusterFS in a way that the file system is synchronized between both regions.

While the aforementioned configuration seems to make sense, our tests reveal that this is actually the worst configuration to consider. The reason is, that GlusterFS, as well as any other shared file system that is not particularly designed for this purpose, uses locking to prevent write conflicts. The locking in combination with high latency between the replicas results in unacceptable response times.

In order to illustrate this pitfall, we conducted an easy experiment where we increase the distance (and therefore the latency) between the backends and GlusterFS and see how much the response time increases. We note that the conducted experiment, which we will describe in the following paragraphs, does not accurately reflect the setup presented in Figure 3.8. In the later experiments, however, we will deploy a system as shown in the figure and confirm the results of this easy experiment.

Experiment Infrastructure

To confirm the above introduced pitfall, we deployed a *Dovecot* in the suggested 3-tier architecture on Amazon's EC2. The installation runs on three t2.micro instances with a proxy, a backend, and a GlusterFS volume that is mounted on the backend node. All connections between the instances are secured via TLS by default; allowing to deploy such setup over multiple clouds and off-site locations.

Furthermore, one t2.micro instance was configured to provide a PostgreSQL database holding the user table, and one t2.micro instance was set up to act as the client machine; executing all tests against the deployment. With this deployment we conducted three experimental setups:

- *Dovecot Ire*: All instances are deployed on machines in the Ireland region of AWS.
- *Dovecot Ire/Lon*: The GlusterFS instance operates in AWS London. All other instances are deployed in AWS Ireland.

- *GMail*: The large-scale IMAP service run by Google. Reachable at `imap.gmail.com`.

The *Dovecot Ire* setup is what we would call the *comfort zone* of *Dovecot*, since there is almost no transmission delay between the instances. In the *Ire/Lon* setup, we have a small but noticeable transmission delay of about 12 ms R/T to the data tier. This setup is generally interesting, because physically separated data and logic tiers make a deployment less prone to outages of a single infrastructure provider. In these scenarios, a complete outage of the worker nodes, like in major AWS outages in 2011 and 2017 [Ser17], can be parried by allocating instances on a different cloud provider. We use the Gmail service as a real-world reference for a production email service, even though we have no insights into GMail's internal infrastructure.

Experiment and Results

We conducted a very simple set of experiments where we injected 1000 consecutive IMAP commands of the same IMAP user in each experimental setup. For each command we measured the round-trip response time. We present average response time, standard deviation, and median for each setup in Table 3.2.

We observe that *Dovecot* delivered a very solid performance in its *comfort zone* setup (Dovecot Ire) with respect to its low standard deviation; as shown in Table 3.2. For the Ire/Lon setup, we observe the expected increase of the response times of *Dovecot*. We note that due to the structure of our experiment, Dovecot is unable to show its optimizations such as index files, to improve performance of IMAP *read* commands like `EXAMINE`, `SEARCH`, and `FETCH`. We expect Dovecot to perform well in case read commands were ever evaluated in terms of response time.

The response time of Google's GMail service is generally not comparable to the other setups, since GMail is a production service that is offered to millions of users. However, the numbers present what a realistic response time performance we can expect from an IMAP service that runs in production.

Table 3.2: Response time per IMAP command in milliseconds.

IMAP		Dovecot Ire	Dovecot Ire/Lon	GMail
`APPEND`	Average	34.66	445.96	553.99
	Std. Dev.	10.07	70.84	56.58
	Median	31.84	439.29	541.52
`CREATE`	Average	19.46	592.22	349.98
	Std. Dev.	2.37	30.01	72.72
	Median	18.83	601.62	337.15
`DELETE`	Average	106.49	1990.20	361.81
	Std. Dev.	18.86	93.65	56.34
	Median	107.06	1977.10	352.99
`STORE`	Average	19.43	267.63	126.62
	Std. Dev.	10.77	53.95	10.88
	Median	15.36	259.49	125.04

Discussion

The results reveal that the used *standard* 3-tier architecture setup of *Dovecot* is very sensitive to increased latency between the backend and the storage. In Table 3.2 we see that the introduced round-trip transmission delay of 12ms between the Ireland and London data center leads to an increase of the response time of more than one order of magnitude. For example, the average response time for a `CREATE` command increased from 19ms to 592ms.

The reason for this unexpected high increase is that one IMAP command that is processed in the backend may leads to multiple operations on the file system. Hence, in contrast to one client's IMAP request that may suffers from high latency, we actually multiplied the number of requests that all suffer from high latency, which ultimately sum up to unacceptable response times.

From this experiment we can derive two lessons we have learned. The first lesson is that one should carefully choose the layer where state is stored and replication is applied. The second lesson is that the replication mechanism must be carefully chosen. The distributed locking, which is used by most of the distributed file systems, and the implied strong consistency guarantee may work for replication within a data center where transmission delays are negligible, but is dangerous when applied across data centers.

In the rest of this chapter we will evaluate our alternative approach where more state is stored in the logic tier. Therefore, we evaluate our prototype against *dsync*—two systems that imply a more relaxed consistency model and replication without distributed locking or consensus.

3.5.2 *Benchmark*

In order to evaluate *pluto* and *Dovecot*, we need a way to apply a large amount of state-changing IMAP commands to our deployments. We are interested in the state-changing ("write") commands of RFC 3501, because only these manipulate mailbox state and trigger downstream messages that need to be applied at other replicas. "Read" commands, in turn, are answered authoritatively on the replica where they were received on; without inter-replica communication. Only state-changing commands potentially unearth consistency issues by generating edge cases. Thus, we require an IMAP benchmark that is able to generate large write-intensive workloads; involving the write commands that are implemented in both services: `CREATE`, `DELETE`, `APPEND`, `EXPUNGE`, and `STORE`.

At the time of conducting the experiments, and to the best of our knowledge, there is no such tool or data set available, and thus we implemented an IMAP benchmark ourselves[5] that generates arbitrary amounts of random data, hence write-intensive workloads. Each workload is composed of small and randomly generated sequences of IMAP commands, called *sessions*. One session always contains well-matched IMAP commands, e.g. a mailbox folder is created before a message is appended to it. The messages that are appended to a folder, and possibly deleted within the same session, contain random strings and different mail headers. The length of a message varies from 10 to 512 lines, resulting in message sizes from approximately 1 to 32 kilobyte. The session generation mechanism is deterministic and can be reproduced by configuring a benchmark with the same seed.

Before a session is executed, a user is chosen randomly from a provided file and logged in. Next, the session commands are applied

5 `https://github.com/go-pluto/benchmark` licensed under GPLv3 or later.

one after another; a successive one as soon as the previous one has finished and the time between sending and receiving a complete answer, i.e. the command's *response time*, has been stored. The workload's degree of parallelism, that is the number of concurrent active users, can be configured as well. All results are written to disk and optionally uploaded to a Google Cloud Storage (GCS) bucket.

With our IMAP benchmark we provide a tool that is able to measure the performance of an IMAP server with respect to the response time. We designed the benchmark in a way, that it complies with RFC 3501. Hence, the benchmark can be used to evaluate almost any IMAP server, including GMail or MS Exchange.

Design Details

As mentioned, a session is composed of randomly chosen IMAP commands that are well-matched. Because the size of the session, i.e. the minimal and maximal number of commands, can be configured by the user and contains a certain randomness, it is difficult to characterize the workload in terms of the distribution of commands.

The session generation, however, follows a certain pattern. For example, the very first command of a session must be a `CREATE` command in order to enable further commands like `DELETE`, `APPEND`, or `SELECT`. Similarly, a `STORE` command can only be executed if there exists a message to operate on. Hence, an `APPEND` command must be executed before `STORE`. With `SELECT`, a previously created folder can be selected to execute the commands `STORE` or `EXPUNGE`.

In the implementation we translated those dependencies between the commands into program code and assigned weights to each enabled command. The weights of the commands are used to create more meaningful session. For example, after a folder has been selected, the probability to select another folder is only 10%. This avoids sessions that mainly create some folders and switch between them without executing any commands that alter the folder's state. The assigned weights are currently not customizable, but may be in future versions.

We note that a session does not *clean up*, i.e. folders and messages that were created during a session are not deleted before the session is finished. Within a session, however, it is still possible that folders or messages are deleted. Hence, if several session are executed consecutively, the mailbox state, i.e. the mailbox size, grows monotonically, whereas within a single session the state will grow and shrink. This fact will become important when we compare the growth of the mailboxes of different replicas.

The session generation is deterministic, meaning that if the random number generator is initialized with the same seed, it will produce the same sessions. This feature is particularly important, because we only achieve a *fair* comparison between the evaluated systems if the used workload is identical.

We discovered that sessions that are executed concurrently on the account of one user may lead to race conditions which threat the comparability of the evaluated systems. Therefore, we decided to implement the session generation in a way that concurrent sessions on one replica never work on the same folders. This is achieved by adding a session specific identifier to the foldernames, in this case the `id` of the Goroutine that is responsible for this session. Concurrent session that are executed on different replicas, however, may work on the same folders and possibly on the same messages. This is, in fact, very unlikely in case the random number generator is initialized with differed seeds. In contrast to that, conflicts that are based on concurrent access to the mailbox can be purposely created by using the same seed at the same time on two or more different replicas.

Maildir Tools

While the IMAP benchmark provides response time measurements, replication lag data is at least as important because it tells us how well a service is able to disseminate and apply updates among its replicas. It complements the user-centric response time metrics by making the asynchronous replication part visible. Due to different replication mechanisms in the evaluated systems, we had to fall back on observing the Maildir file system in order to see when updates were applied.

The credit for the maildir tools and the used cloud infrastructure is due to Matthias Loibl, who contributed the amount of experience that enabled our evaluation.

We implemented a small utility[6] that periodically performs a disk usage calculation of a configured subset of the Maildirs present on a node (by running `'du -s'`). The Unix standard program `du` estimates the file space usage of a selected folder by recursively scanning all

6 `https://github.com/go-pluto/maildir_tools` licensed under GPLv3 or later.

files and sub folders. The results are logged to disk and uploaded to a GCS bucket at the end of the tool's run. For continuous monitoring, a duration histogram is exposed to Prometheus, a standard cloud monitoring tool [Pro18].

With our Maildir Tools we are able to monitor the increase of the size of the mailbox account while benchmarking the server. In addition to the current size of the mailbox, the current system time is logged. With these timestamps we are able to compare the growth of the account on different replicas. Here, the difference between the measured size of the mailbox for two *identical* timestamps on two different replicas represents the *replication lag* in seconds. We will explain this further later in this section when we use the computed difference to characterize and visualize the replication lag of the evaluated systems.

We note that the calculated differences have to be taken as estimations rather than precise values, as we rely on synchronized clocks for timestamp elicitation. We will see later that the clock drift between different replicas, even if the replicas are distributed over huge distances, are negligible, because the lag that is introduced by the synchronization exceeds the noticeable clock drifts by multiple orders of magnitude.

3.5.3 *Infrastructure*

We now introduce the infrastructure setup used in our experiments later on. As guiding principle, we have chosen a *Cloud Native* [Fou18a] approach, featuring the most advanced cloud technologies available at the time of developing our prototype. Our infrastructure is mainly based on two products: Kubernetes [Kub18], an orchestration platform for containerized applications, and Prometheus [Pro18], a powerful monitoring tool.

We provisioned two identical Kubernetes clusters in the `us-east1-b` and `europe-west1-b` regions of the Google Cloud Platform. Each cluster consisted of six `n1-standard` nodes (1 vCPU, 3.75GB memory). We combined both clusters into a Kubernetes cloud federation, enabling cross-cluster service discovery and resource synchronization. For persisting data, we always allocated 100GB SSD volumes. In the following we will write `us` or `europe` to reference the respective regions. The measured round-trip transmission delay between both clusters was approximately 140ms. This value is surprisingly small, thanks to the good network infrastructure between the two Google data centers.

We decided to publish our configurations in our infrastructure repository[7], so that our setup can easily be recreated and reused for further experiments. Hence, all resources, including the container images of all evaluated systems, are publicly available.

Confirmation of the Pitfalls

In order to test our infrastructure, we decided to confirm our findings from Section 3.5.1, i.e. the pitfalls of geo-replication. Therefore, we installed *Dovecot* in both regions of our Kubernetes cluster and configured *GlusterFS* to replicate a shared volume between `europe` and `us`. In Figure 3.8 we show a visualization of this deployment. We note that all components, except the *GlusterFS*, are stateless and therefore easy to place in both regions.

We used our benchmark to generate a write-intensive workload. Since we already saw that 12ms of latency between the backend and *GlusterFS* increases the response time by an order of magnitude, we configured our benchmark with comparable low settings. More concrete, we decided to send 10 sessions and each session contains 15 to 40 IMAP commands. Moreover, we decided not to evaluate concurrent access to the service, that is why our benchmark only operates on a single mailbox account at a time.

We deployed our benchmark in `europe` and configured it to send the commands to the proxy in `europe`, as we would expect it from a geo-replicated setup.

To our surprise, the results were even worse than expected. The experiment took around 38 minutes from the first to the last injected command. This results in a throughput of 0.1 commands per second with an average of over 9 seconds response time per command. Thus, we see this as a confirmation of the introduced pitfall.

7 `https://github.com/go-pluto/infrastructure` licensed under GPLv3 or later.

3.5.4 *Single-Cluster Benchmark*

To evaluate our approach in realistic scenarios, we conducted a set of experiments with our benchmark on our experimental infrastructure. We started by defining our *baseline*, i.e. a reference experiment where we used a standard configuration of *Dovecot* without any replication. Thereafter, we conducted two experiments where we compared *pluto* against *Dovecot* with enabled replication. We will discuss the results of these experiments in the end of this section.

Baseline Experiment

As introduced in Section 3.3.1, we used a *Dovecot* in a traditional 3-tier architecture as reference setup. For the data tier, we deployed a *GlusterFS* with a replicated volume on two `n1-standard` nodes with 100GB SSDs in the `europe` region. The remaining *Dovecot* components, i.e. a proxy and three backends, were installed on our Kubernetes cluster in the same region as *GlusterFS*. We used three backend nodes to illustrate the possibility of partitioning. In this and all later experiments we maintained a total number of 120 active users in three static user partitions and the proxy was configured to redirect users to the backend that was responsible for their partition. In this setup, no replication was introduced besides the synchronized volume in the *GlusterFS* cluster.

We configured our IMAP benchmark to execute 5000 IMAP sessions with a session length between 15 and 40 commands. The degree of parallelism, i.e. the number of users that are concurrently executing sessions, was set to 20. We identified these 20 concurrent users to be best suited for our experiments, because the reference setup reached the best resource utilization at reasonable response times.

We executed our benchmark on our Kubernetes cluster in the `us` region to simulate a write-intensive load from a distant location. In other words, we used a workload that required geo-replication on a system that was not replicated. Thus, high response times but no replication lag were expected.

We call this experiment our *baseline*, because all geo-replicated setups must be able to outperform it. Otherwise, the effort of geo-replication and the introduction of a replication lag is pointless.

Table 3.3: The results of the single-cluster benchmark, showing the response time performance in milliseconds and the throughput in IMAP commands per second. The average and median replication lag is stated in kilobytes and the replication lag area is stated in megabyte*second.

			baseline	*dsync*	*pluto*
Response Time Performance	`CREATE`	Average	251.36	16.24	47.77
		Median	224.50	12.52	28.25
	`DELETE`	Average	602.05	30.81	48.85
		Median	539.38	27.84	28.30
	`APPEND`	Average	437.26	43.02	91.96
		Median	400.87	38.37	57.15
	`EXPUNGE`	Average	112.91	13.87	42.74
		Median	97.05	**6.18**	25.61
	`STORE`	Average	184.16	15.72	52.04
		Median	166.66	**11.93**	31.80
	Throughput		47.17	480.67	256.039
Replication Lag		Average		734.61	39.10
		Median		729.10	**34.44**
		Area		279.89	**17.13**

We show the measured response times in the *baseline* column of Table 3.3. The average and median response times in milliseconds are grouped by IMAP command. We judge the measured values as realistic for this setup.

Single-Cluster dsync versus pluto

In the remaining experiments we focused on the systems that offer multi-leader replication, namely *dsync* and *pluto*. We deployed a setup of one proxy (or director) and three backends (or workers) in the `europe` and `us` Kubernetes clusters. Both setups were connected over a Kubernetes federation and communicated over public IP addresses and TLS-encrypted channels. In the first experiment we replayed the settings from our *baseline* experiment, except that

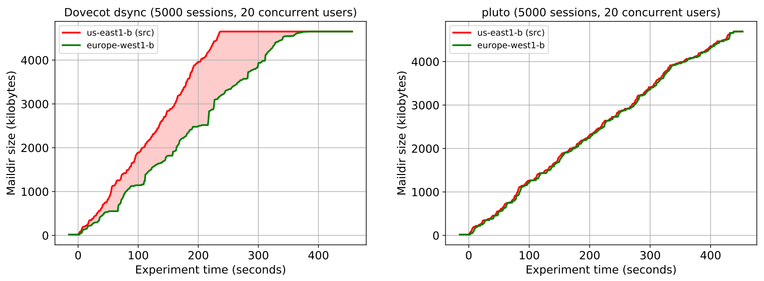


Figure 3.9: Replication lag diagram for *Dovecot dsync* (left) and our prototype *pluto* (right) for requests from `us` to `europe`.

the traffic from the `us` region was now directed to the respective proxy in the same region. In this scenario, the expected behavior is that both systems replicate the updated state from `us` to `europe` asynchronously. During the run we collected the response times and additionally tracked the size of the mailboxes for six selected sample users in both regions with our Maildir Tools. The tracking interval was set to one second, which we found to be the best trade-off between additional overhead by the `du` commands and unavoidable loss of precision. With the chosen interval a possible *micro clock drift* between `europe` and `us` has no significant influence to our results. Based on the collected values we identified the replication lag for both systems. We compare the results for *dsync* and *pluto* in the following two paragraphs.

The measured response times are given in the *dsync* column of Table 3.3. We judge the response times and the resulting throughput, i.e. the processed IMAP commands per second, as optimal for this setup. *Dovecot* is—not for nothing—the state of the art IMAP server software.

For analysis of the replication lag we compare the growth of the mailboxes in both regions for the selected sample users. In the left side of Figure 3.9 we illustrate the average growth for the selected sample users in what we call a *replication lag diagram*. On the x-axis we see the relative time of the experiment in seconds. The y-axis represents the size of the users' mailboxes in kilobytes. The red line represents the growth of the mailboxes in `us`, i.e. the region where the traffic was injected. The green line represents the growth of the replicated mailboxes in `europe`. In this replication lag diagram, a distance between both curves parallel to the x-axis represents the replication lag in seconds, i.e. the time until the `europe` replica catches up. A distance between both curves parallel to the y-axis represents the replication lag in kilobytes[8]. In order to quantify the replication lag, we think that it is feasible to compute the size of the red area between both curves. The computed area in *megabyte*second*, alongside with the average and median replication lag in kilobytes, is presented in the last 3 rows of Table 3.3.

In the *pluto* setup we additionally deployed the *storage* node (see Section 3.4) in a third region (`europe-west2-b`). Because we cannot directly compare the storage node to any *Dovecot* component, we used a more powerful node (`n1-standard-4`, 4vCPU, 15GB Memory) and set the resolution of our Maildir Tools for this node to 3 seconds to avoid any negative impact. The remaining parts of the *pluto* setup is almost identical to *dsync*, i.e. we have one director and three worker nodes with 100GB SSDs in each region.

The measured response times are stated in the *pluto* column of Table 3.3. We note that the response times are significantly higher than *Dovecot*'s, which we discuss in the end of this chapter.

The replication lag diagram for *pluto* is shown in the right part of Figure 3.9. We see that the difference between the curves is almost invisible, which indicates a very small replication lag. The quantified replication lag is shown in Table 3.3.

3.5.5 *Double-Cluster Benchmark*

For our final experiment we split the workload from the previous experiments and used our benchmark from both regions `us` and `europe`, i.e. we executed 2500 sessions from each region to simulate a workload that, in fact, requires geo-replication. The measured response times are stated in the *dsync*^{2} and *pluto*^{2} columns of Table 3.4.

In order to measure the replication lag, we also split the sample users and configured our benchmark in a way that the mailboxes

8 We note that these diagrams require a monotone growth of the mailboxes to be meaningful. Our benchmark generates *mostly* monotone growth, because `CREATE` and `APPEND` commands are more likely than `DELETE`. With the chosen resolution of our Maildir Tools, a declining mailbox size is almost invisible.

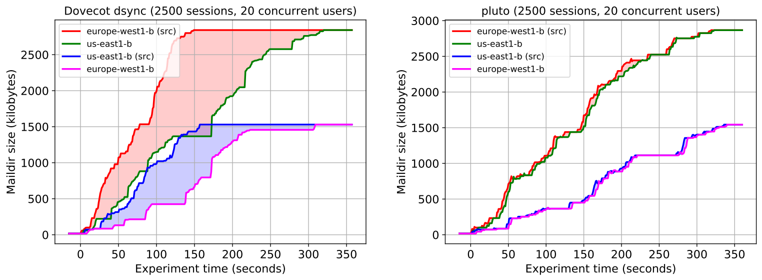


Figure 3.10: Replication lag for *dsync* (left) and *pluto* (right). The red areas represent the replication from `europe` to `us`, while the blue areas represent the opposite direction.

of the first half of the users are only accessed by the `us` benchmark, and the second half by the `europe` benchmark. The mailboxes of the remaining 114 users received commands from both regions. We present the replication lag diagram for both systems in Figure 3.10. The red areas represent the replication lag for synchronizing state from `europe` to `us`, and the blue areas represent the replication lag in the opposite direction.

3.6 DISCUSSION

Evaluation Results

The *baseline* experiment revealed that the absence of geo-replication can be costly with respect to response time and throughput, when the application is faced with traffic from distant regions. As we have seen with both compared systems, using multi-leader replication for traffic from different continents is convincing and necessary. The price for the introduced replication is the relaxation of consistency guarantees and the presence of a replication lag.

By comparing the response times and the achieved throughput of both systems, we clearly see that our prototype cannot keep up with *Dovecot* and that further optimizations are necessary. We acknowledge that throughput often is a performance metric that is placed emphasis on in large-scale services and *pluto* needs to improve in that direction. However, because *pluto* is a research prototype with

Table 3.4: The results of the double-cluster benchmark, showing the response time performance in milliseconds and the throughput in IMAP commands per second. The average and median replication lag is stated in kilobytes and the replication lag area is stated in megabyte*second.

			dsync^2		*pluto*^{2}	
			us	eu	us	eu
Response Time Performance	CREATE	Average	18.47	23.24	47.56	75.20
		Median	14.17	20.33	28.94	29.83
	DELETE	Average	32.46	37.03	47.12	74.61
		Median	29.46	34.31	29.16	29.89
	APPEND	Average	46.39	55.36	87.23	131.79
		Median	42.08	50.34	55.72	58.66
	EXPUNGE	Average	15.59	21.16	40.94	62.72
		Median	**9.44**	**18.91**	22.56	23.19
	STORE	Average	17.48	21.79	46.09	72.84
		Median	**13.83**	**19.64**	29.73	31.53
	Throughput		447.87	367.94	256.26	171.49
Replication Lag		Average	592.87	657.76	18.61	44.98
		Median	217.83	322.10	**6.1**	**34.33**
		Area	97.92	209.83	**5.83**	**14.32**

much less development time compared to the standard IMAP server *Dovecot*, we nevertheless are satisfied with its response time performance. We think that optimizations of the used index structures and file management can lead to improved response times and throughput.

With respect to the replication lag, however, our prototype clearly outperforms *dsync* and we judge our approach as successful. The replication based on the used op-based CRDT is cheap compared to the costly replication of *dsync*. An operation from one replica can almost instantly be delivered and applied on the other replicas without complex tracking of state information. The fact that our approach can be applied with an arbitrary number of replicas makes it even more interesting than *dsync*, where only a pair-wise replication is possible.

We note that our experiments only focus on write-intensive workloads and we purposely omitted the evaluation of read commands. Building an IMAP server that is able to compete with *Dovecot* in all facets is a challenging task and is, at least for now, not our primary focus. In our opinion, the improvement of our IMAP-CRDT and exploration of further standard IT services that can be modeled with CRDTs is a promising direction for future work.

Opportunities and the Price of Replication in the Logic Tier

While our evaluation mainly analyzes the response time and the replication lag, there are other benefits that arise when we add multi-leader replication in the logic tier. In addition to the possibility to reach planetary-scale across different continents, the increased fault tolerance can be utilized in systems that run within an single data center as well.

In contrast to *traditional* 3-tier configurations, e.g. our *baseline* setup, our prototype tolerates temporary connection losses to the data tier. If the connection to the storage is lost, a worker node can continue to operate without noticeable service disruption or a limitation of the application's features. After the connection is reestablished, the operations are transmitted to the storage and, due to the inherent guarantees of CRDTs, the worker and the storage eventually converge. The possibility to tolerate temporary disconnects between the worker and the storage enables a better maintainability, because components in one layer can be maintained while the remaining components can continue to serve clients' requests.

Moreover, the latency between the logic and the data tier has negligible influence on the performance of the prototype. The double-cluster benchmark revealed that we are able to achieve comparable good response times, even though we additionally maintained a storage node in contrast to our opponent *Dovecot-dsync*. Traditional configurations, however, are very sensitive to increased latency between the logic and the data tier, as we have demonstrated when we analyzed the pitfalls of geo-replication (see Section 3.5.1). This particular property of our prototype enables setups where the data tier can be placed on local infrastructure, e.g. a datacenter in the facilities a company, and the logic tier can in turn be placed on machines that run on a public cloud to achieve a good availability, similar to hybrid-cloud setups. In case of an outage of the machines of the cloud provider, requests can be rerouted through the stateless components, i.e. the director, and served by the storage on local infrastructure. In this scenario, full functionality is preserved. Clients requests, however, may suffer from a slightly increased response time and from a temporary loss of the updates that have not been transmitted to the storage. In any case, the possibility to tolerate outages of a cloud provider[9] is something that cannot be achieved in systems that are not designed to support this kind of replication.

The costs of our approach are the increased amount of metadata that needs to be processed and the overhead of the enforced causal communication. Both issues have been addressed in the recent literature, which we present in the next section as *related work*. It is, however, difficult to state whether the small throughput of *pluto* compared to *dsync* is a result of both aforementioned issues, or the lack of fine tuning and optimizations of our prototype. The fact that *pluto* and the underlying IMAP-CRDT is not limited to two replicas must be considered as well when comparing both systems.

9 Most cloud services like Amazon's AWS offer sophisticated solutions to tolerate outages of single data centers, for example changes on the an Elastic Block Storage (EBS) volume are automatically synced to three availability zones within milliseconds. Those specialized solutions, however, cannot be used outside AWS or in an hybrid/multi-cloud setup, resulting in unavailability of the service if one provider is unavailable.

Transferability to other IT Services

We would like to point out that we chose IMAP as the protocol for modeling with a custom CRDT not because it is better suited for this purpose than other protocols. We chose IMAP because of its widespread use and fundamental importance in everyday life—and because its relative simplicity. We judge the fact that the state of an IMAP server is based on relatively simple structures, namely its tree-like mailbox structure, as particularly advantageous for modeling the commands with operations on a CRDT. Hence, as long as the structural complexity of an application's state is manageable, our approach is promising.

The requirements for CmRDTs, i.e. the commutativity of concurrent operations, or the required properties of the *merge* function of CvRDTs, limit the type of the state that can be modeled. Fortunately, research on CRDTs has led to several useful data types that can be used in more complex applications. In addition to the counter, OR-Set, and the IMAP-CRDT that has been presented in the previous sections, there exist CRDTs for registers, lists, and for JSON objects. We expect that with the recently introduced JSON CRDT, the modeling of more IT services with CRDTs will become even easier [KB17]. We will summarize the recent achievements of the research around CRDTs in the following *related work* section.

While our approach of designing a CRDT for a specific application was successful, it is certainly questionable whether this way of realizing multi-leader replication in the logic tier is the most desirable one. Generally, we are convinced that our strategy to analyze the functionality, design the fitting payload and update operations, verify the necessary properties, and implement the service is a sophisticated and promising way of engineering. The development of further replicated IT-Services following this engineering requires domain knowledge and expertise in CRDT design.

In order to make this approach more accessible, we see the following two opportunities. First, an intuitive abstraction could help to convince developers to use CRDTs in their application. The JSON CRDT contributes in this direction, since the general structure of a JSON is well understood and widely used as de facto standard data exchange format of the web. Second, programming libraries that encapsulate the CRDT implementation and only offer a small but usable set of operations would reduce the barrier of adopting our approach to a minimum. Both directions are currently part of the ongoing research initiatives and industry programs, which we see as a confirmation that a transfer of our approach to other IT-services can be achieved.

3.7 RELATED WORK

Large-scale distributed systems replicating state in an available and partition-tolerant way have received academic attention since the advent of the Internet. Bayou [Ter+95] was one of the first distributed storage systems that enabled users to always submit updates and ensured eventual consistency when the network connection was available again. Inspired by the fundamental concepts captured in Amazon's Dynamo paper [DHJ+07], a new class of distributed data stores was proposed and developed, such as Cassandra [LM10] and Riak [Bas18]. Many of these new developments were based on the ideas of Google's Bigtable concept [Cha+08], which Google itself turned into Spanner [CDE+12], its planetary-scale, strongly consistent, and partition-tolerant distributed database. Their solution towards the CAP dilemma is to run Spanner on an expensive and highly sophisticated private network which ensures almost no downtimes [Bre17].

Regarding automatic resolution of conflicting writes in any distributed system, the choice is between discarding all but one update or merging all updates into one. The most common technique for the first approach is known as *last write wins* (LWW), where the update with the biggest timestamp is picked as winner and all others are lost. For example, DeCandia et al. [DHJ+07] describe an anomaly at Amazon where items in a shopping card ceeps reappearing due to poor conflict resolution. One well known merge-based resolution strategy is *Operational Transformation* [EG89], though mostly used for collaborative text editing and of decreasing performance with increasing number of operations [AN+11; DI16; JB17]. Conflict-free Replicated Data Types take a different approach as they avoid conflicts altogether due to their construction properties.

Related Work on CRDTs

CRDTs have been introduced as a theoretical framework by Shapiro et al. to achieve SEC in a distributed network of replicas [Sha+11b]. As mentioned in Section 3.2, there are two variants of CRDTs: *operation-based* and *state-based*. Operation-based CRDTs (also known as CmRDTs or op-based CRDTs) achieve convergence by requiring causal-order communication and commutativity of concurrent updates. In contrast to that, state-based CRDTs (or CvRDTs) require a merge function over a join-semilattice, which computes the least upper bound to reflect the combined state of diverged replicas. Since the introduction of CRDTs in 2011, researchers created a wide portfolio of data types for various purposes. The authors of the corresponding technical report introduced state- and op-based types for counters, registers, and other basic data types [Sha+11a]. Further achievements in the design of CRDTs include maps [BAL16], sets [Bie+12], lists (e.g. RGA [Roh+11], Treedoc [LPS09], WOOT [Ost+06a], Logoot [WUM10], LSEQ [Néd+13]), XML [MUW10], and the already mentioned JSON-CRDT [KB17]. With our IMAP-CRDT we contribute a verified op-based CRDT for a standard IT-service to this list.

The price of using CRDTs is the increased amount of metadata that needs to be processed in order to achieve convergence. For example, the *add* operation in an OR-set includes a unique tag α that is stored and eventually sent over the network. To address this issue, Baquero et al. introduced the notion of *pure* op-based CRDTs [BAS14]. The authors propose a CRDT design that uses a *tagged* causal-order broadcast that provides the functionality of the tag metadata on the communication layer. In essence, they propose to reuse the vector clocks as tags, which results in less metadata. Furthermore, the authors introduce the notion of *causal stability* which allows to further reduce the amount of handled tags after a certain stable state is reached. In our prototype *pluto* we also implemented the idea of using the vector clocks as tags for the operations on folder level. We think that transforming the IMAP-CRDT to a pure op-based version is certainly a promising improvement, but out of scope of this thesis.

State-based CRDTs are designed to send a current snapshot of the state from one replica to another in order to apply the merge function. This obviously results in disadvantages when the state grows to a certain size. To this end, Almeida et al. proposed a solution to only send the relevant parts of the state to the other replicas, namely delta-CRDTs [ASB18; ASB15].

Causal Communication and Causal Consistency

In addition to the commutativity of concurrent updates, CmRDTs require operations to be applied in . To this end, vector clocks are widely used in distributed systems to capture causality [Fid88; Mat88]. It is an inherent problem of vector clocks that the number of entries in the clock grows linear with the number of replicas, which makes the use of vector clocks for every replica prone to performance drops in geo-replicated systems [Bai+12]. To address this issue, researchers proposed messaging middlewares with message sequencers within the datacenters in order to reduce the overhead to a constant factor [BSS91; ALaR13; And+09; Lad+92; Ter+94]. The disadvantage of those sequencers is that the communication within one datacenter must be routed to the central sequencer unit, which results in limited parallelism while avoiding metadata explosion. In contrast to that, systems that track the causality explicitly by using vector clocks [Bai+13; Du+13; Llo+11; Llo+13] suffer from the aforementioned scalability problem.

The most advanced systems we have seen in the latest literature are GentleRain [Du+14], Cure [Akk+16], and Saturn [BRVR17]. The first two mentioned systems rely on a periodical background task called *global stabilization* that balances metadata overhead and delayed visibility of updates. The authors of Saturn introduce a tree-based dissemination of the metadata, which guarantees causality per architectual design. With their approach the authors were able to demonstrate that their system is able to achieve causal communication at planetary-scale with only 2% overhead compared to an eventually consistent system that sacrifices causality, i.e. accepts updates in any order. To further optimize the intra-datacenter communication, the authors suggest a second system called Eunomia [GBR17], which leverages Hybrid Clocks, a combination of logical and physical time [Kul+14]. We think that our prototype *pluto* would certainly benefit from a more advanced messaging middleware that guarantees causality without using the currently implemented causal communication based on vector-clocks for every replica. However, for the purpose of demonstrating that our approach can be transferred to a working prototype, we think that it is justified to omit further implementation optimizations of the messaging middleware.

Formal Verification of CRDTs

Many designers of CRDTs have elaborated on the correctness of their CRDTs with respect to the required properties to guarantee convergence. Alongside with the initial introduction [Sha+11b; Sha+11a], Shapiro et al. provided the very first formalization of the necessary properties and an abstract proof. Furthermore, the correctness of more complex types, e.g. the RGA CRDT for ordered lists, has been convincingly demonstrated in handwritten proofs [Att+16; Roh+11]. Although there is no reason to doubt the correctness of those proofs[10], machine-checked proofs deliver more convincing results. To this end, interactive theorem provers like Isabelle/HOL [NWP02] provide the certainty that the proof steps are reasonable. It is, however, dangerous to assume that machine-checked proofs are always correct. False or contradicting assumptions, as well as wrongly implemented definitions, can lead to faulty clues and incorrect proofs.

The only two approaches to mechanically verify the correctness of CRDTs are from Zeller et al. [ZBPH14] and Gomes et al. [Gom+17b]. The authors of both work introduce Isabelle frameworks for verifying the convergence of a CRDT in an abstract model of a distributed system. Here, Zeller et al. focus on state-based CRDTs and show the correctness of state-based counters, registers, and sets. In contrast to that, Gomes et al. introduce a framework for op-based CRDTs and propose a proof for a counter, the OR-Set, and the RGA. With our proof of the IMAP-CRDT we contributed an additional example for the framework of Gomes et al. to Isabelle's Archive of Formal Proofs and provided the certainty that our CRDT ensures convergence in an abstract but realistic model of an asynchronous distributed system [JOL17b].

10 In the next chapter we will see that there were, in fact, issues with published and well accepted results in the OT-research community. For CRDTs we could not find any reported violation of claimed properties.

Formal reasoning about the properties of distributed systems using mechanical verification tools is an active area of research. The first contributions in the area include the work of Charron-Bost et al. [CBDM11] on verifying a fault tolerant consensus mechanism with Isabelle. Coincidentally, our previous work together with the colleagues from the chair for *Models and Theory of Distributed Systems* on verifying the impossibility of crash-tolerant asynchronous consensus from Fisher, Lynch, and Patterson [FLP85; Bis+16] fits in this list of related work, even though it is not a contribution of this thesis.

One interesting byproduct of our Isabelle implementation of the IMAP-CRDT is that we are able to extract working source code from the Isabelle code [HN10]. Gomes et al. reported that they successfully extracted the implementation of an op-based counter as a distributed program that runs on n nodes with communication over TCP channels [Gom+17b].

Abstractions, Accessibility, and Applications

As outlined in the previous discussion, the development of intuitive abstractions and programming libraries are part of ongoing research initiatives. With Lasp [MVR15], Meiklejohn et al. proposed a programming model that is entirely build with always converging data types. Hence, from the developers perspective it is easy to build scaling application, because the used primitives and components are converging by design. The authors show that they were able to scale their prototype application to 1024 EC2 nodes without generating unmanageable overhead [Mei+17]. Lasp is, however, currently not as expressive as common languages like Python or Go and therefore unsuitable to implement multi-leader replication on the logic tier.

In the beginning of this section we already referred to databases that support multi-leader replication, like Apache Cassandra [LM10] or Amazon's Dynamo [DHJ+07]. These NoSQL databases leverage *last write wins* as strategy to solve conflicts. The applications that run on top of those databases are generally stateless and do not add additional replication to the logic tier.

Apart from LWW, we have seen Basho's Riak KV database [Bas18] which provides Riak-specific data types based on CRDTs. Applications that are build on top of Riak benefit from the scalability of

CRDTs and a more fine-tuned conflict avoidance strategy compared to the LWW approach. Unfortunately, Basho discontinued the work on Riak, but it remains available as an open-source project.

A more research-driven alternative to Riak is AntidoteDB [AB16]. The authors implement a transactional model around CRDTs and are, to our knowledge, the first who combine transactions and converging data types. Their approach towards a more desirable consistency model is called *just-right consistency*, and allows the application to decide whether to prefer performance and low latency or strong consistency on the fly [Sha+18]. AntidoteDB is written in Erlang and includes Cure [Akk+16] to provide causal-order communication.

We note that both Riak and AntidoteDB provide programming abstractions for CRDTs and promise to achieve planetary-scale. They *hide* all implementation details of the replication, e.g. the causal-communication layer, so that developers can benefit from the high scalability without explicitly dealing with the concurrency. To this end, both databases require a sophisticated setup to achieve the claimed promises. For example, they cluster the nodes within one data center in a ring topology where the partitions are aligned.

We think that both Riak and AntidoteDB are certainly promising tools to develop geo-replicated applications. However, there is a subtle difference to the approach we took in this chapter. In our approach, we promote the idea to achieve a more independent logic tier that can not only be used in geo-replicated setups, but also within one data center to tolerate temporary communication interruption to the data tier. Neither AntidoteDB or Riak are designed to be used in this configuration. It would, however, be interesting to see how an IMAP server based on those databases would compare against our prototype in a geo-replicated setting.

3.8 CHAPTER SUMMARY

In this chapter we presented one approach to explicitly handle more state in the logic tier and thus achieving a more distributed and less centralized setup compared to today's standard cloud-based services. To this end, we conducted an extensive case study for the Internet Message Access Protocol where we analyzed the feasibility of our approach in depth.

In order to realize convergence in a replicated system of backends with multiple leaders, we utilized Conflict-free Replicated Data Types and introduced our own IMAP-CRDT which maps all *consistency critical* IMAP commands to operations on an op-based CRDT. The required properties, i.e. the commutativity of concurrent operations and the termination of operations, were proven successfully with the interactive theorem prover Isabelle/HOL. At this point, we provided the certainty that an IMAP service can be designed to run in a setup with multi-leader replication and that all ever occurring conflicts can be automatically solved.

In order to transfer our *theoretical* approach into *practice*, we developed a prototype, namely *pluto*, which implements the IMAP-CRDT and a messaging middleware for causal communication. For our evaluation we deployed our prototype on a state-of-the-art cloud environment in a kubernetes federation spanning over multiple regions. Furthermore, we designed a reusable IMAP Benchmark which generates synthetic but reproducible *write-intensive* IMAP workloads.

In our experiments we focused on geo-replicated scenarios, where we assumed clients from distant locations; possibly from different continents. We first illustrated the pitfalls of geo replication, i.e. the things that could go wrong when replication is naively applied in the data tier. Thereafter, we conducted single- and double-cluster benchmarks of our prototype against *Dovecot-dsync*, the de facto standard IMAP server software.

As result, we were able to show that the replication lag can be significantly reduced with our approach. The response times were significantly better compared to a non-replicated system (*baseline*). However, *Dovecot* still delivered the highest throughput, which indicates that our prototype still needs improvements in order to compete with industry-grade software.

In our discussion we illustrated the opportunities (fault tolerance, hybrid-cloud deployments, geo-replication) as well as the disadvantages (causal communication overhead, metadata, expressiveness) of our approach. Ultimately, we discussed the transferability of our approach and how other IT-services could benefit from enabled multi-leader replication in the logic tier. To this end, we presented an extensive list of related work that address the outlined issues.

We judge that our approach, where we began with the system design and verification followed by the implementation and evaluation, turned out to be successful in this regard. The resulting prototype combines *theory* and *practice* by leveraging CRDTs in a standard IT service and is able to play off its conceptual advantages. We encourage fellow system designers to follow in this path to consider CRDTs for modeling state and update operations. Furthermore, we are happy to see that the contributions of this chapter were accepted in the research community [JO17; JOL17a; JOL17b] and hopefully create a lasting impact for the upcoming challenges.

4 ON STATEFUL PRESENTATION TIERS WITH OT

4.1 CHAPTER OVERVIEW

Parts of this chapter have been published in [JH15; JH16; JCR17; JB17] and in a previous thesis [Jun14]. I will highlight the contributions of the co-authors accordingly alongside the sections.

In the previous chapter we explored the feasibility to store and process more state in the logic tier. Consequently, this exploration is followed by the corresponding analysis for the presentation tier, which will be the focus of this chapter. Hence, we again follow our idea of an unconventional architecture that purposely breaks the widely applied service statelessness principle.

As mentioned in Section 2.1.2, the presentation tier typically runs the code that includes all interface-related functionality and the invocation of requests to the other layers. In order to reduce the complexity of this chapter, we focus on web-based applications. This restriction allows us to make more assumptions on the characteristics of the presentation tier, mainly because the interface-related code is executed in a web browser. Hence, in contrast to our exploration of a stateful logic tier in the previous chapter, the application state is now stored, processed, and replicated at the client's site; possibly on mobile devices with unreliable Internet connection.

We note that there already is a class of services that apply our approach, i.e. a stateful presentation tier with multi-leader replication, namely online collaboration services. These services include successful collaboration applications like Google Docs [DR18] or Etherpad [Fou18b]. We note that both mentioned tools utilize Operational Transformation (OT) as multi-leader replication mechanism. While the feasibility of a stateful presentation tier has already been proven successful for this class of services, the exploration in this chapter aims to make this approach more accessible for other services beside online collaboration applications.

In order to transfer the technological approach of online collaboration services to a broader range of web services, an extension of the underlying OT mechanism is necessary, i.e. the possibility to replicate mutable JSON objects with OT. Since JSON is the de facto standard data interchange format of the web, the need of this extension for our purpose is inevitable. Therefore, we contribute the needed extension, verify the convergence guarantees, provide a prototype of a patient documentation system to show the applicability of our extension, and introduce a library to provide the necessary transferability to other applications.

4.2 OPERATIONAL TRANSFORMATION

An alternative approach to CRDTs (see Section 3.2) is Operational Transformation (OT), which was first introduced by Ellis and Gibbs in 1989 in the context of collaborative groupware systems [EG89]. In such a groupware system, multiple collaborators share a document and independently update the content. In turns out that such systems implement a multi-leader replication mechanism, because each collaborator holds a replica of the shared document on a local computer and write-operations can be executed without waiting for the other replicas to approve. These collaboration systems recently received a lot of attention, for example Google Docs [DR18] or Etherpad [Fou18b] utilize OT to synchronizes changes to a shared document.

The intuition behind OT can be best explained with the following example. Two users u_1 and u_2 maintain their own replica of the character sequence `abc`. Both users simultaneously invoke edit operations on their local replica. The user u_1 inserts an `X` at position `0`, resulting in `Xabc`. The user u_2 deletes the character `b` at position `1`, resulting in `ac`. A naive interchange of the invoked edit operations would result in diverging replicas: u_1 results in `Xbc`, whereas u_2 results in `Xac`. With OT, remote operations are *transformed* based on previously executed local operations. Hence, u_1 needs to transform the position of the remote delete operation to respect the effect of the local insert operation, i.e. u_2's delete operation on position `1` needs to be transformed to a delete operation on position `2` to ensure convergence.

In order to achieve the above mentioned transformation, an OT system is composed of two components: a *transformation function* and a *control algorithm* [Sun02]. In essence, the transformation function defines how to transform one operation against another operation. The control algorithm, however, defines when and in which order two operations are transformed against each other. As visualized in Figure 4.1, there is a joining element between the transformation function and the control algorithm, i.e. the *transformation properties*. These properties can be seen as requirements to the transformation function in order to be compatible with the control algorithm. All of the aforementioned components will be explained in detail throughout the rest of this section.

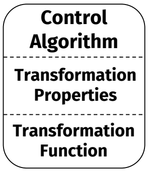


Fig. 4.1: The OT Architecture [Sun02].

Operations and Transformation Functions

In order to precisely define the mechanics of a transformation function, we reuse the above introduced example of a collaborative editing session. In the example we used two operations, namely *insert* and *delete*, on a sequence of characters, i.e. a list. That is why we use list operations to present the concepts of a transformation function. Hence, both operations require a precise definition in order to further elaborate the transformations.

For the rest of this chapter we use the notation of list operations as presented in Table 4.1. The notation is inspired by the programming language Python and covers the essential primitives to access and reason about lists. With the introduced notation we are able to precisely define the two operations in Definition 6 and 7. We write insert_L and delete_L to reference the operations on lists.

Definition 6 (insert_L). The operation insert_L has three parameters: an item i, a position k and a list L with $k \leqslant |L|$. As result, the item i is inserted into the list L at position k:

$$\text{insert}_L(i, k, L) \triangleq L[< k] + [i] + L[\geqslant k]$$

Definition 7 (delete_L). The operation delete_L has two parameters: a position k and a list L with $k < |L|$. As result, the item at position k is deleted from L:

$$\text{delete}_L(k, L) \triangleq L[< k] + L[> k]$$

We note that the presented definitions for operations on lists are rather standard. The remaining piece to fully describe the aforementioned scenario is, in fact, the transformation function. Therefore, we present a generic definition of a transformation function in Definition 8.

Table 4.1: Notation for lists based on the programming language Python.

Notation	Description
$[]$	**Empty List and Delimiters** We use $[$ and $]$ as delimiters for a list. Hence the list $[x, y, z]$ contains the items x, y and z. We denote the empty list as $[]$.
$\|L\|$	**Length of a List** We define the length of a list L as the number of items in the list, denoted as $\|L\|$.
$L[n]$	**List Access** Let L be an arbitrary list. We denote the access to the n^{th} element as $L[n]$. We note that $L[n]$ is only defined if $n < \|L\|$. We access the first element with $L[0]$.
$L[x, y]$	**Intervals** Let L be an arbitrary list. We write $L[x, y]$ for a new list that contains all elements from $L[x]$ to $L[y]$. We note that $L[x, y]$ can be the empty list if $x > y$, or if x and y referencing to non existing elements. If only y is referencing to a non existing element, $L[x, y]$ contains all elements from $L[x]$ to the last element of L.
$L_1 + L_2$	**List Concatenation** Let L_1 and L_2 be two arbitrary lists. We define the concatenation of L_1 and L_2, denoted as $L_1 + L_2$, as a new list that starts with L_1 and ends with L_2.
$L_1 \subseteq L_2$ $L_1 \subset L_2$	**Sublists and Strict Sublists** Let L_1 and L_2 be two arbitrary lists. We call L_1 a *sublist* of L_2, denoted as $L_1 \subseteq L_2$, if L_2 starts with L_1. If $L_1 \subseteq L_2$ and $\|L_1\| < \|L_2\|$, we call L_1 a *strict sublist* of L_2, denoted as $L_1 \subset L_2$.
$L[\leqslant x]$ $L[< x]$	**Head and Tail** Let L be an arbitrary list. We write $L[\leqslant x]$ for a new (sub)list that contains all elements from the first element to $L[x]$. We use the abbreviation $L[< x]$ for $L[\leqslant x - 1]$. The lists $L[\geqslant x]$ and $L[> x]$ are defined analogously.

Definition 8 (Transformation Function). A *Transformation Function* has a pair of operations (O_1, O_2) as input and returns a pair of transformed operations (O_1', O_2') where O_1' is the transformed version of O_1 with and O_2' is the transformed version of O_2.

For a transformation function called XFORM, we sometimes write XFORM^1 or XFORM^2 to reference the first or the second transformed operation, hence:

$$\text{XFORM}^1(O_1, O_2) = O_1' \text{ and } \text{XFORM}^2(O_1, O_2) = O_2'$$

The presented definition of the transformation function is based on the introduction of the Jupiter OT system [Nic+95]. The input of a transformation function can be seen as two independent update operations at two replicas. The operations are applied at the two replicas, which may result in an inconsistent state. In order to regain a consistent state of the replicas, the transformed versions of the operations are exchanged and applied at both replicas.

In Listing 4.1 we present one transformation function for list operations XFORM_L, which was initially introduced by Ellis and

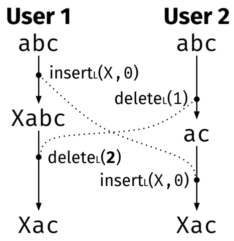


Figure 4.2: A transformation example.

Gibbs [EG89] and slightly improved by Ressel et al. [RNRG96]. In the listing we omit the last parameter of each operation since the list, also called *context*, of all operations is defined implicitly.

The transformation of the "abc" example, which we also visualize in Figure 4.2, would be processed in the lines 7 and 13 of Listing 4.1. Here, the $\text{insert}_L(X, 0)$ operation from u_1 would be transformed against the $\text{delete}_L(1)$ operation from u_2. We note that the position parameter of the insert_L operation is smaller than the delete_L operation, hence k1 < k2. According to the transformation function, the position parameter of the delete_L operation must be increased by one in order to include the effect of the concurrent insert_L operation. The insert_L operation does not need to be changed, since the effect is not influenced by delete_L. Ultimately, both replicas converge to the same state Xac.

The rest of the transformation function in Listing 4.1 covers the remaining cases for concurrent insert_L and delete_L operations. According to line 4 we need to use application dependent priorities to transform two insert_L operations with identical position parameters. This case is typically called an *insert-insert tie*, because both replicas independently insert an item at the same position. One typical example to solve this tie is by defining a total order among the replicas in order to prioritize operations. For delete_L operations with identical position parameters (see line 19), both replicas independently delete the same element from the list which result in a consistent state. Hence, both operations are transformed to no-op.

Listing 4.1: Pseudo code of the transformation function XFORM_L.

```
function XFORML(insertL(i1, k1), insertL(i2, k2)):
  if k1 < k2: return(insertL(i1, k1), insertL(i2, k2 + 1))
  if k1 > k2: return(insertL(i1, k1 + 1), insertL(i2, k2))
  if k1 == k2: # use application dependent priorities

function XFORML(insertL(i, k1), deleteL(k2)):
  if k1 < k2: return(insertL(i, k1), deleteL(k2 + 1))
  if k1 > k2: return(insertL(i, k1 - 1), deleteL(k2))
  if k1 == k2: return(insertL(i, k1), deleteL(k2 + 1))

function XFORML(deleteL(k1), insertL(i, k2)):
  if k1 < k2: return(deleteL(k1), insertL(i, k2 - 1))
  if k1 > k2: return(deleteL(k1 + 1), insertL(i, k2))
  if k1 == k2: return(deleteL(k1 + 1), insertL(i, k2))

function XFORML(deleteL(k1), deleteL(k2)):
  if k1 < k2: return(deleteL(k1), deleteL(k2 - 1))
  if k1 > k2: return(deleteL(k1 - 1), deleteL(k2))
  if k1 == k2: return(no-op, no-op)
```

Transformation Properties

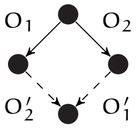


Fig. 4.3: An illustration of TP1.

Ressel et al. discovered two requirements for the transformation function in order to fulfill the promise that all cases lead to converging replicas, namely TP1 and TP2 [RNRG96]. The Transformation Property 1 (TP1) describes the already illustrated case where there are two concurrent operations that are transformed against each other. We illustrate this property in Figure 4.3, where we see two operations O_1 and O_2 and the corresponding transformed versions O'_1 and O'_2. Assuming that both replicas start in the same state, the consecutive execution of O_1 and O'_2 must lead to the same state as O_2 and O'_1; hence the illustrated diamond in Figure 4.3.

We present a more formal definitioWen of TP1 in Definition 9 where we reuse the already introduced *Kleisli arrow composition* from Section 3.3.4.

Definition 9 (Transformation Property 1). Let O_1 and O_2 be two operations. A transformation function XFORM satisfies the *Transformation Property 1* (TP1), if the following holds for $\text{XFORM}(O_1, O_2) = (O'_1, O'_2)$:

$$O_1 \triangleright O'_2 = O_2 \triangleright O'_1$$

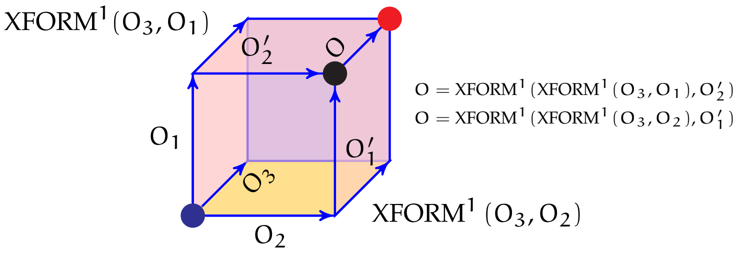

$XFORM^1(O_3, O_1)$
O_2'
O
O_1
O_1'
O_3
O_2
$XFORM^1(O_3, O_2)$
$O = XFORM^1(XFORM^1(O_3, O_1), O_2')$
$O = XFORM^1(XFORM^1(O_3, O_2), O_1')$

Figure 4.4: Illustration of TP2 [RNRG96].

In fact, the presented transformation function $XFORM_L$ in Listing 4.1 satisfies TP1. This has been proven by Imine et al. in [Imi+03] and has been confirmed in [Liu+14] and [SXA14]. However, designing a transformation function that satisfies TP1 is not trivial. Imine et al. found a counterexample in the first transformation function of Ellis and Gibbs [EG89], which has been corrected by Ressel et al. [RNRG96].

While TP1 suffices for two concurrent operations, Ressel et al. discovered that if there are three concurrent operations, a stronger property, namely TP2, is needed. We visualize this Transformation Property 2 in Figure 4.4. In the figure we see the three concurrent operations O_1, O_2, and O_3 spanning a 3-dimensional object. In essence, TP2 describes that there are two options to reach the red point from the black point, and that is by first transforming O_1 against O_3, or O_2 against O_3. According to TP2, the result, i.e. the final state at the red point, must be identical. We present an adapted version based on the Kleisli arrow composition in Definition 10.

Definition 10 (Transformation Property 2). Let O_1, O_2 and O_3 be arbitrary operations. A transformation function XFORM satisfies the *Transformation Property 2* (TP2), if the following holds for $XFORM(O_1, O_2) = (O_1', O_2')$:

$$\begin{aligned} & O_1 \triangleright O_2' \triangleright XFORM^1(XFORM^1(O_3, O_1), O_2') \\ = {} & O_2 \triangleright O_1' \triangleright XFORM^1(XFORM^1(O_3, O_2), O_1') \end{aligned}$$

It turned out that almost all proposed transformation functions for lists fail to satisfy TP2. In fact, the presented transformation function XFORM_L by Ressel et al. also breaks TP2, which has been shown with the help of a model-checker by Imine et al. [Imi+03]. We will discuss this further in Section 4.7, where we present the related work of our JSON extension. We note that the fact that TP2 is not satisfied by most of the transformation functions predetermines the set of control algorithms. That is why we focus on one particular control algorithm, namely *Wave* [DWL10], for the rest of this chapter.

Control Algorithms

The OT control algorithms can be categorized into two groups: either they require the transformation function to fulfill TP2 (dOPT [EG89], adOPTed [RNRG96], GOTO [SE98], SOCT2 [SCF97; SCF98]), or TP1 is sufficient (Jupiter [Nic+95], Wave [DWL10]). Those control algorithms that require TP2 can be seen as peer-to-peer algorithms, since there is no additional coordination required to achieve convergence. In contrast to that, the more popular control algorithms Jupiter and Wave use a central server that sequences the operations. Thus, one replica only needs to transform operations against the server's operation sequence and therefore TP1 is sufficient.

It seems that the need of a central server to sequence all operations is a major disadvantage compared to the peer-to-peer based OT systems. In web-based collaboration tools, however, the central server is generally not an issue because updates between clients must be sent over a server anyway, due to the absence of browser-to-browser communication[1]. As a consequence, the need of the central server is currently limiting the scalability of OT, which we will confirm in our evaluation in Section 4.6. In the following we briefly sketch the mechanics of Wave, because we will use this algorithm in our prototypes and for the later evaluation.

The mechanics of Wave can be best illustrated with an example where one replica and the server diverge by two operations each. This is visualized in left side of Figure 4.5, where we see the diverged states of the replica at the position $[2, 0]$ (visualized as red dot) and the server at $[0, 2]$ (visualized as blue dot). In Wave, the operations from the client that have not been sent to the server are divided

1 More recently, WebRTC is addressing this issue. Unfortunately, not all browser vendors currently support WebRTC. Nevertheless, we expect to see a shift in that regard, which we also discuss in Section 5.2.

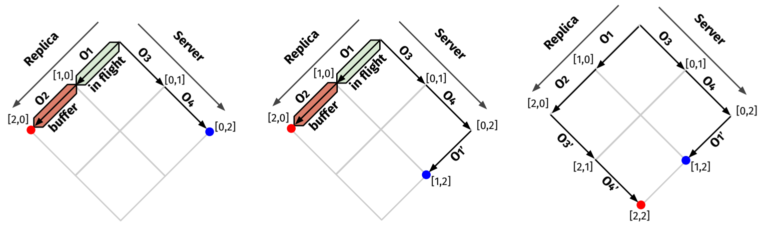


Figure 4.5: Three steps of transforming two operations of a diverged replica against the server's operations.

into an *in flight* operation, which represents the the first diverging operation, and the remaining operations, called buffer. Together, those operations form a *bridge*, which represents the sequence of operations that have not been sent to the server.

Because the transformation function cannot be directly applied to merge the two states at $[2,0]$ and $[0,2]$, the client first sends the in flight operation to the server. Based on the revision number of the in flight operation, the server computes the concurrent operations O3 and O4 and sends both back to the client, followed by an acknowledgment. At this point the server applies the transformation function on the in flight operation against the concurrent operations O3 and O4 in order to include the effects of those operations before the in flight operation is added to the server's history. We note that the server is now at a state where O1 is included in the server's history. The replica, however, consecutively transforms the two operations O3 and O4 against the bridge.

We illustrate these two steps in the middle and right side of Figure 4.5, where we see how the bridge evolves into the direction of the server. Again, we use the red and the blue dot to illustrate the current state of the replica and the server.

In the right side of Figure 4.5 we see that the replica is now only one operation ahead of the server, and that the transformed version of O2 is now the in flight operation. Since there are no further concurrent operations in the server's history, which can be identified by the revision number, the in flight operation can be transmitted to the server and applied immediately without further transformation. Ultimately, the server sends an acknowledgment to the client and broadcasts the operation to the other replicas. We note that the replica and the server reached the same state and are converged.

The stated control algorithm guarantees causal consistency, and therefore eventual consistency, because the partial order of operations that is sequenced by the server is in accordance with the happened-before relation from Definition 1. In order to provide a more precise description of the control algorithm, we present the pseudo code in the appendix of this thesis in Section A.1.

4.3 FROM TREE TRANSFORMATIONS TO JSON OPERATIONS

This section extends a transformation function which has been partly introduced in a previous thesis [Jun14]. The contributions of this thesis, however, include the transformation of replace_T *operations and the JSON mapping.*

As a main contribution of this chapter we present our extension of OT to support simultaneous editing of JSON objects. More concretely, we introduce a TP1-valid transformation function for ordered n-ary trees and present a mapping to the JSON components. Hence, in this section we precisely define our data model, the transformation, and our translation of operations on JSON objects to operations on n-ary trees.

4.3.1 *Tree Operations*

We consider ordered n-ary trees with the simplest set of operations insert_T, delete_T, and replace_T. An n-ary tree is recursively defined as a pair of a value and a list of trees. A leaf is defined as a pair of a value and an empty list. Thus, a tree cannot be empty and the smallest tree is a single leaf. As shown in Figure 4.6, we use a list of natural numbers (called *access path*) to access the tree at a specific position. For a tree T and an access path pos we write $T[\![pos]\!]$ to access the *subtree* at position pos. We define the operations insert_T, delete_T, and replace_T in Definition 11, 12, and 13.

Definition 11 (insert_T)**.** The operation insert_T has three input parameters: a tree t, a non empty access path pos and a tree $T = (v, L)$. As result, the tree t will be inserted into T at position pos. We define the operation recursively:

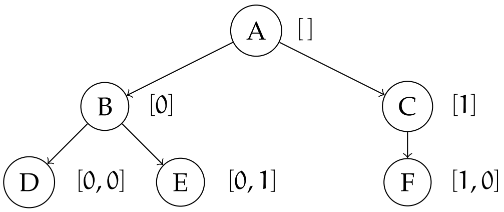


Figure 4.6: Tree representation and node access by access paths.

$$\text{insert}_T(t, [x], (v, L)) \triangleq (v, \text{insert}_L(t, x, L))$$

$$\text{insert}_T(t, [x] + xs, (v, L))$$
$$\triangleq (v, \text{insert}_L(\text{insert}_T(t, xs, L[x]), x, \text{delete}_L(x, L)))$$

Definition 12 (delete$_T$). The operation delete$_T$ has two input parameters: a non empty access path pos and a tree $T = (v, L)$. As result, the subtree at position pos will be deleted from T. We define the operation recursively:

$$\text{delete}_T(t, [x], (v, L)) \triangleq (v, \text{delete}_L(x, L))$$

$$\text{delete}_T(t, [x] + xs, (v, L))$$
$$\triangleq (v, \text{insert}_L(\text{delete}_T(xs, L[x]), x, \text{delete}_L(x, L)))$$

Definition 13 (replace$_T$). The operation replace$_T$ has three input parameters: a value v', an access path *pos* and a tree $T = (v, L)$. As result, the value at position *pos* in T is replaced by v'. We define the operation recursively:

$$\text{replace}_T(v', [\,], (v, L)) \triangleq (v', L)$$

$$\text{replace}_T(v', xs, (v, L))$$
$$\triangleq (v, \text{insert}_L(\text{replace}_T(v', xs[> 0], L[xs[0]]), xs[0], \text{delete}_L(xs[0], L)))$$

We note that the presented definitions of the tree operations are rather standard. In essence, the all operations replace the subtree below the root node with a modified version, where either a subtree is inserted, deleted, or a value is replaced. The only notable difference is that the replace$_T$ operation is also defined on an empty access path. In this case, the value of the root node is altered.

According to our definition of trees, the second last element of an access path determines the node where a subtree should be inserted into or where a subtree should be deleted from. The last element of an access path determines the position inside the list of subtrees of the node at the second last element. We simply use the operation insert_L to insert a tree into the list of subtrees and we use delete_L to delete a tree from the list of subtrees.

We notice that the definitions of insert_T, delete_T, and $\mathrm{replace}_T$ are insufficient if the access path directs to a non-existing node. Therefore, and for the rest of this chapter, we assume that we have a *valid* access path for the operations, i.e. the access path directs to a position where we can apply insert_T, delete_T, or $\mathrm{replace}_T$. This assumption can safely be made, because this validity can be checked when initiating the operation on a replica. With TP1, the transformation function assures that further transformations of the access path do not lead to undefined or inconsistent states. We define the validity of tree operations more precisely in Definition 14.

Definition 14 (Valid Tree Operations). Let O be a tree operation on the tree T. We call O a *valid tree operation* if:

- **case 1:** $O = \mathrm{insert}_T(t, pos, T)$, then:

  Up to the second last element, the position parameter pos directs to an existing subtree (v, L) in T and the last element of pos is a valid position parameter less or equal than $|L|$.

- **case 2:** $O = \mathrm{delete}_T(pos, T)$, then:

  The position parameter pos directs to an existing subtree in T.

- **case 3:** $O = \mathrm{replace}_T(v', pos, T)$, then:

  The position parameter pos directs to an existing node in T.

4.3.2 *A Transformation Function for n-ary Trees*

In order to develop a transformation function for n-ary tree operations, we introduce the definition of the transformation point and construct a transformation function that satisfies TP1. We maintain a very high level of detail, because the transformation of tree operations requires a precise definition of the transformed access paths.

Definition 15 (Transformation Point). Given two non empty lists l_1 and l_2 of natural numbers. The *Transformation Point* (TPt) is the index of the first difference of l_1 and l_2. If $l_1 \subseteq l_2$, the Transformation Point is the index of the last element of l_1, or vice versa.

If we consider two tree operations, the transformation point marks the point where a transformation may be necessary. We give two short examples of the transformation point:

$$\mathrm{TPt}\left([1,2,3],[1,2,4]\right) = 2 \qquad \mathrm{TPt}\left([1,0],[1,0,3,2]\right) = 1$$

With the definition of the transformation point we are able to determine whether two operations are effect dependent or effect independent, i.e. if a transformation is necessary or not. We provide a definition for the effect independent tree operations in Definition 16.

Definition 16 (Effect Independence of Tree Operations). Let pos_1 and pos_2 be the access paths of the operations O_1 and O_2 and tp be the transformation point of pos_1 and pos_2. The operations O_1 and O_2 are *effect independent tree operations*, denoted by $O_1 \parallel O_2$, iff:

1. $(|pos_1| > (tp+1)) \wedge (|pos_2| > (tp+1))$
2. $(pos_1[tp] > pos_2[tp]) \wedge (|pos_1| < |pos_2|)$
3. $(pos_1[tp] < pos_2[tp]) \wedge (|pos_1| > |pos_2|)$

The three cases of Definition 16 are visualized for two insert_T operations in Figure 4.7. The trees t_1 and t_2 are the subtrees which are inserted by the two insert_T operations O_1 and O_2. The effect of the operation O_1, that is the insertion of t_1, is visualized as a blue circle. The effect of the operation O_2 is visualized as a red circle. We note that the transformation point in all examples is 0. In the left tree we demonstrate the first case of Definition 16. Both trees t_1 and t_2 are inserted in nodes which are beyond the transformation point. The trees in the middle and in the right of Figure 4.7 represent the second and third case of Definition 16. In these cases one tree is inserted below a node left to the position where the other tree is

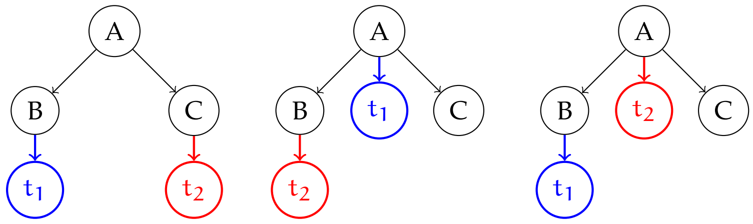


Figure 4.7: Demonstration of the cases of effect independent tree operations from Definition 16.

inserted. We note that the order of two effect independent operations does not matter[2].

In the transformation function for operation on lists in Listing 4.1 we have seen that the position parameters of the operations, i.e. the point where an item is inserted or deleted, are either increased or decreased by one, based on the effect of the concurrent operation. For tree operations the transformation of position parameters is slightly more difficult, because the position parameter is, in fact, a sequence of positions that needs to be transformed at a certain point, i.e. the transformation point. To achieve this we further define two operations, namely update^+ and update^- in Definition 17 and 18, that modify the access path at a particular position.

Definition 17 (update^+). The function update^+ has two input parameters: an access path pos and a number n. The result is a modified access path, where the n^{th} element of pos is *increased* by 1.

Definition 18 (update^-). The function update^- has two input parameters: an access path pos and a number n. The result is a modified access path, where the n^{th} element of pos is *decreased* by 1.

With the provided update operations we are able to define the transformation functions for all combinations of insert_T, delete_T, and replace_T. In order to reduce the complexity of this thesis, we exemplify the transformation for all combinations that include replace_T. The remaining transformation functions for combinations of insert_T and delete_T are stated in the appendix of this thesis in

2 We refer to our technical report where we prove this claim for all combinations of operations [JH15].

Listing 4.2: Pseudo code of the transformation of replace$_{\mathrm{T}}$ against insert$_{\mathrm{T}}$.

```
1  function XFORMT(replaceT(v, pos1), insertT(t, pos2)):
2    TP = TPt(pos1, pos2)
3
4    if effectIndependent(pos1, pos2) or pos1 == []:
5      return(replaceT(v, pos1), insertT(t, pos2))
6
7    if pos1[TP] > pos2[TP]:
8      return(replaceT(v, update+(pos1, TP)), insertT(t, pos2))
9
10   if pos1[TP] < pos2[TP]:
11     return(replaceT(v, pos1), insertT(t, pos2))
12
13   if pos1[TP] == pos2[TP]:
14     if len(pos1) < len(pos2):
15       return(replaceT(v, pos1), insertT(t, pos2))
16     else:
17       return(replaceT(v, update+(pos1, TP)), insertT(t, pos2))
```

Section A.2, and were subject of our work in [JH15] and my previous thesis [Jun14]. The transformation functions for those combination have been successfully proven to satisfy TP1.

In the following we introduce the transformation functions and provide the necessary intuition to understand the mechanics. We start with the transformation of replace$_{\mathrm{T}}$ against insert$_{\mathrm{T}}$ in Listing 4.2. In line 2, the transformation point for both operations is computed. We note that the transformation point according to Definition 15 is only defined, if the access paths are not empty. In order to address this issue, we catch this case alongside with combination that are in any case effect independent in line 4 and 5. In both cases, no further transformation necessary.

The more interesting case is a conflict which we illustrate in Figure 4.8. In the figure we see the desired effect of an insert$_{\mathrm{T}}$ operation to insert a subtree t at the left most position below the root node; visualized with with blue circle. The concurrent replace$_{\mathrm{T}}$ operation, however, aims to replace the value below the most right node; visualized as a bold red value. In this case, a transformation is necessary because after the insert$_{\mathrm{T}}$ operation is executed, the access path of the replace$_{\mathrm{T}}$ operation no longer directs to the desired node. This case is captured in line 7 and 8 of Listing 4.2. We use the update$^+$ function to perform the necessary increase of the position at the transformation point.

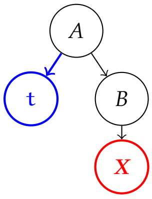


Fig. 4.8: *A conflict between an* insert_T *and* replace_T *operation.*

The same puzzle occurs if the situation mirrored, as visualized in Figure 4.9. Here, the replace_T operation modifies the value at position [0], whereas the insert_T operation aims to insert a new subtree t at position [1, 0]. It turns out that both operations are not in conflict and the order of execution is irrelevant. Hence, we capture this case in line `10` and `11` where no transformation is performed. In the remaining lines of Listing 4.2, the already shown case reoccurs in a more specialized form, where we additionally need to track the length of the access path in order to identify whether a transformation is necessary. With the introduced mechanics we show that the stated transformation function satisfies TP1.

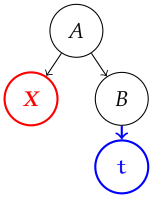


Fig. 4.9: *Two independent* insert_T *and* replace_T *operations.*

Lemma 1. *The transformation function for the transformation of* replace_T *against* insert_T *satisfies the Transformation Property 1 (TP1).*

Proof Sketch for Lemma 1. In order to verify that TP1 holds, we analyze the cases in Listing 4.2 separately. If the two operations are effect independent TP1 obviously holds, because no transformation is applied. The only *conflicts* that need to be solved occur if the insert_T operation inserts a subtree somewhere to the left along the access path of the replace_T operation. In these cases we need to show the following equation:

$$\begin{aligned} & \text{insert}_T(t, pos_2, \text{replace}_T(v', pos_1, T)) \\ = \; & \text{replace}_T(v', \text{update}^+(pos_1, tp), \text{insert}_T(t, pos_2, T)) \end{aligned}$$

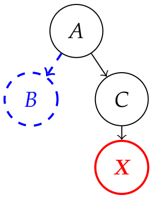


Fig. 4.10: *Two conflicting* delete_T *and* replace_T *operations.*

Fortunately, this case can be easily shown by unfolding the definitions of replace_T and insert_T, together with the fact that an insert_T operation on the tree T with the access path pos is equivalent to an insert_T operation on a trimmed version of pos and T. We show this equation by induction over the length of the trimmed access path. The detailed proof is provided in [JH15]. □

In the next case we consider the transformation of replace_T against delete_T. Fortunately, the mechanics are very similar to the already introduced transformation of replace_T against insert_T, so we can slightly reduce the level of detail. We show the transformation function of interest in Listing 4.3.

Similar to the transformation of replace_T against insert_T, we check whether the two operations are effect independent or if the access

Listing 4.3: Pseudo code of the transformation of replace$_T$ against delete$_T$.

```
1   function XFORMT(replaceT(v, pos1), deleteT(pos2)):
2     TP = TPt(pos1, pos2)
3
4     if effectIndependent(pos1, pos2) or pos1 == []:
5       return(replaceT(v, pos1), deleteT(pos2))
6
7     if pos1[TP] > pos2[TP]:
8       return(replaceT(v, update-(pos1, TP)), deleteT(pos2))
9
10    if pos1[TP] < pos2[TP]:
11      return(replaceT(v, pos1), deleteT(pos2))
12
13    if pos1[TP] == pos2[TP]:
14      if len(pos1) < len(pos2): # replace above a deleted node
15        return(replaceT(v, pos1), deleteT(pos2))
16      else: # replace a deleted node
17        return(noop, deleteT(pos2))
```

path is empty. As a consequence, no transformation is performed, as described in line 4 and 5.

The next two cases in Listing 4.3 are closely related to the outlined *conflicts* in the previous case. Here, both operations are in conflict if a delete$_T$ operation removes a subtree somewhere to the left alongside the access path of the replace$_T$ operation. We illustrate this case in Figure 4.10, where we use a dashed node to visualize the target of the delete$_T$ operation. We note that in the illustrated case a different order of execution leads to diverged states. Consequently, we use update$^-$ to perform the necessary shift of the access path of the replace$_T$ operation in line 8.

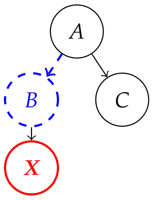


Fig. 4.11: A hierarchy conflict between a delete$_T$ *and* replace$_T$ *operation.*

We identified a more interesting case where the replace$_T$ operation aims to modify the value of a node that has been removed by a concurrent delete$_T$ operation, as visualized in Figure 4.11. In this case, the deletion of the parent node makes the replace$_T$ operation unnecessary. This is reflected in our transformation function by a transformation of replace$_T$ to no-op in line 17.

Lemma 2. *The transformation function for the transformation of* replace$_T$ *against* delete$_T$ *satisfies the Transformation Property 1 (TP1).*

Listing 4.4: Pseudo code of the transformation of replace$_T$ against replace$_T$.

```
1  function XFORMT(replaceT(v1, pos1), replaceT(v2, pos2)):
2
3    if pos1 == pos2:
4      # use application specific priorities
5    else:
6      return(replaceT(v1, pos1), replaceT(v2, pos2))
```

Proof Sketch for Lemma 2. The proof is equivalent to the proof of Lemma 1, only that we use update^{-} to correspond with concurrent delete$_T$ operations. All existing cases can be proven by unfolding the definitions of replace$_T$ and delete$_T$, together with the fact that a delete$_T$ operation on the tree T with the access path pos is equivalent to a delete$_T$ operation on a trimmed version of pos and T. A detailed version of the used lemmas is, again, shown in [JH15]. □

The remaining combination is the transformation of replace$_T$ against replace$_T$, which we introduce in Listing 4.4. Fortunately, this transformation function is rather easy because there exists only one case where a conflict occurs, namely when the access paths of both operations are identical. We note that in this *replace-replace tie* there is no solution that includes the effects of both operations. Hence, one operation must win against the other one. To this end, we simply propose application based decisions in line 4. Two typically applied solutions are the already mentioned *last write wins*, or a total ordering among the replicas, where one replica gets priority over another. The last write wins strategy can be easily implemented alongside with the Wave control algorithm, because the server sequences the operations and can easily determine which one was the last by comparing the revision numbers[3].

Lemma 3. *The transformation function for the transformation of* replace$_T$ *against* replace$_T$ *satisfies the Transformation Property 1 (TP1).*

3 We note that this is slightly different than the typically applied last write wins, which usually refers to a time stamp and relies on synchronized clocks.

Proof Sketch for Lemma 3. The transformation function obviously satisfies TP1, because at no point a transformation is applied. □

With the stated transformation functions in Listing 4.2, 4.3, and 4.4, together with the remaining transformation functions in the appendix in Section A.2, we are able to present the final theorem:

Theorem 2. The transformation function XFORM_T satisfies the Transformation Property 1 (TP1).

Proof. The claim follows directly from Lemma 1, 2, 3, and the proofs for the remaining combinations that are presented in [JH15]. □

We note that our transformation function XFORM_T enables simultaneous editing of ordered n-ary trees with support for insert, delete, and replace operations. We utilize those features when we present our mapping of JSON components to n-ary trees, in order to achieve these capabilities for JSON objects as well.

4.3.3 *Simultaneous Editing of JSON Objects*

The JavaScript Object Notation (JSON) is the de facto standard data interchange format of the web. While JSON has its origin as serialization format for JavaScript, many other programming languages provide serialization and deserialization mechanisms for JSON as well. Since 2014, the Internet Engineering Task Force defines the structure of the notation in RFC 7159 [Bra14].

As visualized in Figure 4.12, the data within JSON is structured in three components: an object (1), an array (2), and a value (3). An object (1) is a *unordered* set of key/value pairs. An array (2) is an *ordered* list of values and a value (3) is either an array, an object, or a simple type like a string or a number. We see that JSON has both hierarchical structure and ordered elements. For example, a hierarchy is created by nesting objects. One obvious example for ordered elements are the values within an array.

We note that our presented transformation function for ordered n-ary trees is capable of handling both hierarchy and the order of elements. Hence, we handle both challenges in one step. To do so, we map the JSON structure to the tree structure in order to achieve an OT synchronization of JSON objects.

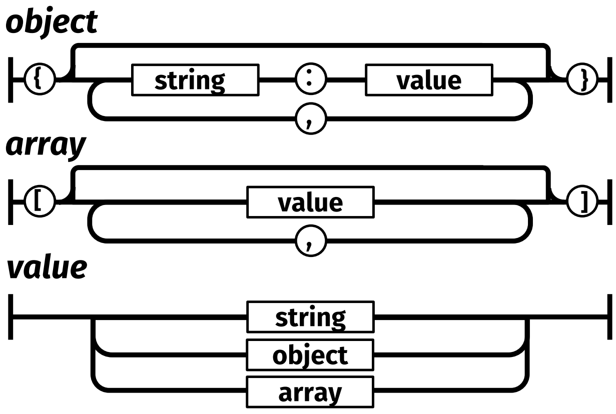


Figure 4.12: Simplified structure of a JSON object [Bra14].

In our mapping we introduce the following four node types with the corresponding rules:

- An object node, denoted as {}, is the parent of arbitrary many key nodes. Moreover, the root of the tree is an object node.
- A key node has exactly one value node as child.
- A value node either is a simple type, an object node, or an array node.
- An array node, denoted as [], is the parent of arbitrary many value nodes.

We visualize one example mapping in Figure 4.13, where we show one JSON object on the left side and the corresponding tree representation, that follows the above stated rules, on the right side of the figure. We note that this translation from a JSON object to an ordered n-ary tree is straightforward and rather simple. However, this mapping is unfortunately ambiguous, because a different order of `key1`, `key2`, and `key3` would lead to a different order inside the tree, even though this order is irrelevant for a JSON object. To address this issue, we additionally add the following rule:

- All key nodes below an object node must be ordered by their lexicographic order.

```
{
  "key1": [42, true],
  "key2": "string",
  "key3": {
    "key4": null,
    "key5": []
  }
}
```

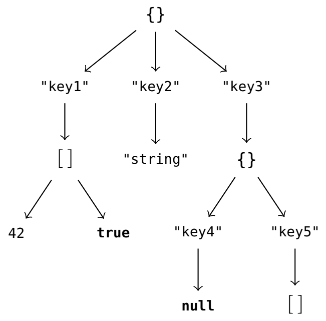


Figure 4.13: Tree representation of a JSON object.

We note that with the above stated rule the corresponding tree to a JSON object is unambiguous.

JSON Access Path

Next, we map operations on a JSON object to the already presented tree operations. Here, the challenge is to project our notion of an access path, which we used to navigate inside the tree, to a reasonable navigation inside a JSON object. Fortunately, there exists a similar mechanism to address this issue, called a JSON Pointer [Bry13]. Within such a pointer, a combination of keys and positions inside an array is used to navigate through the object. Hence, we propose a direct translation of a JSON Pointer to our access path. For example, the position parameter of an operation on the JSON object in Figure 4.13 that aims to insert an item at position `0` of the array with the key `"key5"` would be translated in the following way:

$$[\texttt{"key3","key5",0}] \rightarrow [\texttt{2,0,1,0,0}]$$

We note that this translation works flawlessly only if the keys inside an object are ordered. With the presented translation we are able to use the introduced tree operations insert_T, delete_T, and replace_T to modify the JSON object. Furthermore, we are able to allow simultaneous editing with OT, because our transformation function for those operations satisfies TP1.

Design Details and Discussion

While the above introduced mapping from the components of a JSON object to a tree, together with the translation of a JSON Pointer to an access path, seems straightforward, there are some important details and design decisions to consider. For example, the presence of concurrent operations may lead to ambiguous tree mappings when two replicas independently insert the same key below an object node. According to the JSON specification, the keys below an object node `SHOULD` (and not `MUST`) be unique [Bra14]. Hence, at this point we assume a more strict definition of JSON by requiring these keys to be unique. Fortunately, the specification allows implementation specific behavior in these cases, so with our assumption we are still in conformity with RFC 7159.

The above illustrated example can be seen as another *insert-insert tie*, where two operations concurrently try to insert an item at the same position. One example to enforce the uniqueness of keys is that one of the insert_T operations is transformed to a replace_T operation, and the other to no-op. In this case, one operation has a priority and the other operation will be rolled-back. We note that the same tie can be observed with two replace_T operations or combinations of replace_T and insert_T.

Another important design decision of our mapping is that we enforce an order of keys within an object, even though the order is irrelevant for the object itself. We see room for improvement here, because we currently maintain additional and unnecessary structure. An ideal transformation function would be based on the JSON structure entirely; including customized operations and precise definitions of operation validity. However, the fact that we can map JSON operations to our tree operations demonstrates that our transformation function XFORM_T is actually more expressive, which can be important when using other models than JSON. We judge the resulting overhead as justified, because, we gain the certainty that the transformation is *correct* with respect to TP1.

4.4 OPEN-SOURCE COLLABORATIVE PATIENT DOCUMENTATION

In the previous section we introduced an OT-based mechanism to enable simultaneous editing of replicated JSON objects. Here, the replicas can independently update the state of an object and the update operations are sent asynchronously (over a server) to the other replicas. Hence, we enabled a multi-leader replication architecture for mutable JSON objects. With our verified transformation function, we provided the certainty that the replicas will eventually converge to the same state. In order to show the applicability of our approach, we present a case study where we use our OT extension to handle more state in the presentation tier of a standard hospital IT service: a patient documentation system.

I would like to highlight the contributions of Juan Cabello, who designed and implemented the prototype.

4.4.1 *Paper-based Solutions*

Hospitals and clinics typically use complex enterprise software, like hospital information systems (HIS), to run their business operations. This software covers a broad variety of tasks, from treatment documentation to billing and resource planning. We observe many different software systems that are closely interconnected. These systems often lack a seamless integration, which results in frustration on the side of the medical and administrative staff.

During our observations at the Charité, the largest university hospital in Europe, we were able to confirm the mentioned lack of seamless integration, as we have seen many paper based workarounds for existing enterprise solutions. One example, which we found most interesting, was the use of a shared Word document for patient documentation. Here, the medical staff of a department with around 40 beds used printed copies of the shared document to organize treatment documentation as well as the daily schedule.

The used document is structured in a tabular layout where one row represents one currently admitted patient. The information per patient included the room & bed number, name, birth date, a short anamnesis, notes about the treatment, and a collection of the future tasks. These documents are typically printed out for each clinician in that department at the beginning of a shift. Throughout the day, each clinician independently annotates the printed document. At the end of the shift the annotations are used to update the shared Word document, which is, once finished, separately stored to capture the history. When a patient is discharged from the clinic, the information in the history are used for a report that is processed in the HIS. We note that these paper based solutions are still quite common in many hospitals, even though such workarounds create major issues, e.g. information loss or inconsistent data.

According to our interviews at the Charité, the main reason why the paper based solution is preferred over the documentation solution by the HIS is, that annotations on the paper are significantly faster and easier to make than using a desktop client. Hence, from a systems point of view, the paper solution has better *availability*. Moreover, the product that was used at the time did not offer any interaction mechanism for mobile devices. We find that our approach of handling the application state in the presentation tier, i.e. very close to the client, is promising in this case and might to be able to compete with the paper based solutions, because we expect similar availability and more convenience by tailoring our approach to mobile devices, such as smartphones or tablets. To this end, we identify the most important requirements to realize our approach in a working prototype.

4.4.2 *Requirements*

The development of a suitable patient documentation for the described purpose goes along with strict requirements of usability and compatibility with law. We identify[4] the most important requirements to be the following:

1. **Always Available:** Even in unstable network environments, the clinicians must be able to read, modify, and add information.

2. **Privacy Protection:** Sensitive information must not be disclosed to unauthorized parties.

3. **Cross-platform:** The system must support various platforms, especially mobile devices, such as tablets.

4 We note that the list of requirements is by no means complete. In this subsection we focus on the *relevant* requirements that relate to distributed systems challenges.

We note that the first requirement raises the already mentioned challenge of the CAP dilemma. We simply cannot guarantee a consistent view of the data alongside with partitioned networks and the availability requirement. Hence, we cannot avoid that doctors see different information on their mobile devices if network failures are present. This seems to be very critical in a hospital environment. However, with the *traditional* paper-based solutions, where every doctor annotates a personal copy of the patient's file, inconsistencies are inevitable and widely accepted. To automatically solve possible inconsistencies and to provide a high responsiveness if a working network connection is present, we use or JSON extension of OT.

The second requirement leads to strong encryption of the stored data and the communication. Depending on the regulations of the country, no external service provider like Google Docs can be used, since sensitive data must remain inside the hospital's network.

The third requirement ensures that existing devices, such as PCs, smartphones, and tablets, can be used without the need for a specific hardware.

4.4.3 *Application Design*

From the stated requirements we derive an application design based on the latest available open-source technologies. We note that the third requirement favors the development of a web-based application. In contrast to a native mobile application, a web-based application runs on various operating systems without further adaptation. Together with the first requirement, a single-page application is required. The single-page application runs, once loaded, completely autonomous in the browser. This suggestion is in accordance with the second requirement, since no external software service is required.

If one doctor updates a patient's record, the update is immediately present on the doctor's device and the information will be propagated to the other devices as soon as possible. If the network connection is unavailable, the updates will be queued. In order to regain a consistent state after a patient's record has been updated, we apply the previously introduced JSON extension for OT.

We note that the fact that every doctor has a replica of the data which can be accessed and modified directly on the mobile device is, in fact, a multi-leader replication scenario. Moreover, in this case it is necessary to hold the application state alongside with the single-page application at the doctor's device, which can be seen as a *stateful* presentation tier.

4.4.4 *Prototype*

Based on the stated application design, we implemented a prototype, called *HotPi*, that aims to be an alternative to the paper based solutions, which are still used in many hospitals. In our implementation we decided to follow the latest trends in web development at the time of writing this thesis in order to demonstrate that we do not rely on outdated libraries. Therefore, we use MERN [Has18] as technology stack for our single-page application.

The MERN stack is a software bundle that is based on the programming language JavaScript and comprises four building blocks from which the name derives: (1) MongoDB for the database, (2) Express.js for the server application, (3) React.js with Redux for the client application and (4) Node.js for the server platform. The main difference of a MERN application, compared to a *traditional* web application, is, that the user interface, together with the user interface related functionality, runs autonomously in the browser. This enables more responsive user interfaces, because updates of the interface, for example as a result of a click on a menu item, can be rendered without requesting additional style information from a server.

To implement our approach as a MERN application, we use the architecture that is visualized in Figure 4.14. Here, the client part of the Wave control algorithm is implemented as a React.js component that runs autonomously in the client's browser. We note that the application and the state is stored in the browser's *local storage*, which enables availability even in case no network connection is present. As long as the application is loaded in the local storage, updates on the state can be made, which will be exchanged when the network connection is reestablished.

The server part of the Wave algorithm is, consequently, implemented as Express.js application that runs on a dedicated backend in the logic tier. Fortunately, both server and client parts are written in the same language, namely JavaScript, which allowed us to reuse

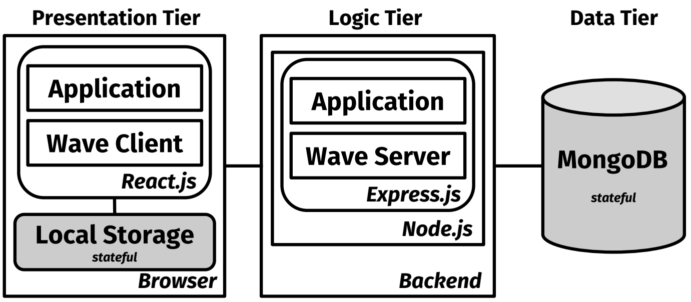


Figure 4.14: MERN-based architecture of our *HotPi* prototype with a stateful presentation tier.

parts of the code. All state information that are processed in the logic tier are persistently stored in MongoDB, which represents a stateful data tier. Hence, the used application design features are, as intended, a stateful presentation tier, a stateless logic tier, and a stateful data tier.

In our data model we use one JSON document per patient, where we store the essential information that are currently used in the paper-based solution. We utilize the insert and delete operations from our transformation function XFORM_T to add and remove characters to the description fields. This way we achieve the possibility to collaboratively edit patient information, similar to Google Docs [DR18].

As shown in Figure 4.15 and 4.16, our prototype features a comfortable way to annotate the patient's information with different notes. We structured the layout based on the paper document structure we have found at the hospital. Hence, doctors can add, delete, and modify notes for the anamnesis, the treatment documentation (or history), and the upcoming tasks. Particular information in the notes can be highlighted with different colors as well as different priorities can be chosen.

Since the underlying technology of our prototype is OT, the prototype supports both real-time collaboration and an offline mode. Hence, as long a network connection is available, multiple doctors can edit the notes simultaneously. If no network connection is available, the application remains fully functional and the information is updated when the device is online again.

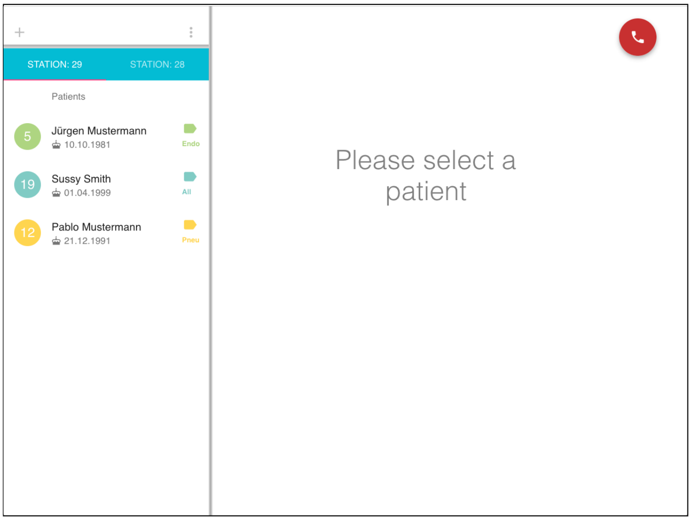
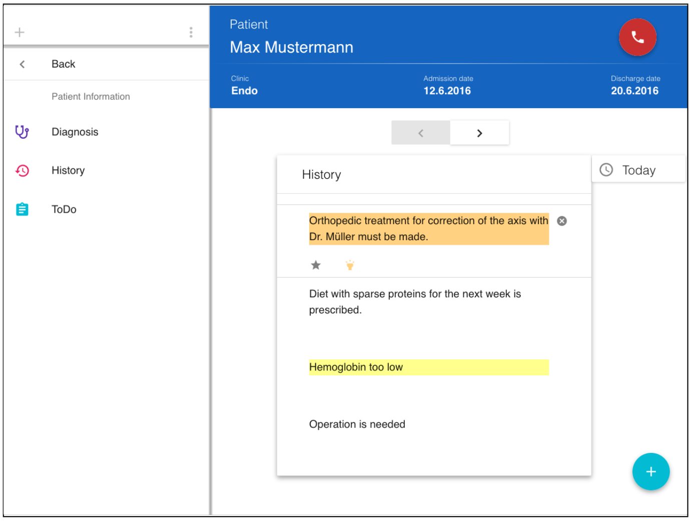

STATION: 29
STATION: 28
Patients
5 Jürgen Mustermann 10.10.1981 Endo
19 Sussy Smith 01.04.1999 All
12 Pablo Mustermann 21.12.1991 Pneu
Please select a patient

Figure 4.15: Patient selection view of the prototype.



Orthopedic treatment for correction of the axis with Dr. Müller must be made.

Diet with sparse proteins for the next week is prescribed.

Hemoglobin too low

Operation is needed

Figure 4.16: Active notes of a patient treatment.

We note that the prototype follows the information structure that we have seen at the Charité. Other hospitals are likely to use a different structure to process the patient information. Therefore, we have chosen an open-source license that allows an easy adaptation of the prototype[5].

4.4.5 *Discussion*

With our prototype, we demonstrate that our JSON extension of OT can be transferred from theory to practice, and that we are able to build applications that benefit from the integrated multi-leader replication. We strongly believe that the underlying technology, i.e. Operational Transformation and our JSON extension, is highly suitable to solve the problems that we observed with the paper-based solution in the clinical environment. The fact that we were able to combine modern web development based on MERN with OT demonstrates that there are no technology related restrictions for our approach.

However, we admit that transferring our prototype into a usable product requires a significant amount of work. Especially the compliance with the regulation in medical environments raises additional challenges that need to be addressed. For example, since the application state, and therefore sensitive patient information, is held on a mobile device, the access must be restricted and secured against unauthorized parties. These challenges require careful thought, which we omitted in the presentation of our prototype. However, we are convinced that these challenges can be solved and that our contribution, i.e. the extension of OT, can be actually used in future products.

4.5 *formic*: A LIBRARY FOR COLLABORATIVE APPLICATIONS

Based upon the insights that we have gained from our patient documentation prototype, we aim to generalize our used architecture and provide the necessary accessibility to develop new applications with our approach. To this end, we present *formic*, a programming library to build collaborative applications. Hence, we aim to provide

The design, implementation, and evaluation of formic has been carried out by Ronny Bräunlich, for which I am indefinitely thankful.

5 URL: `https://github.com/hotpi` licensed under GPLv3 or later.

the necessary tool to use and process more state in the presentation tier, without the need to explicitly handle the replication mechanism and the convergence of replicas. Since our library is designed to include our JSON extension to OT, we are able to use our library to evaluate the performance of our approach at high scale, which we present in Section 4.6. The source code of the library, as well as the documentation and detailed instructions, is available in the project repository[6].

4.5.1 *Selected Features and Challenges*

We begin with the features that should be included in the programming library. The major challenge in the library design, compared to the earlier presented prototype for a specific application, is, that the library must be able to handle arbitrary many clients that simultaneously edit arbitrary many objects. Hence, handling the degree of parallelism, as well as the necessary user convenience, is in the focus of the later derived library design. We summarize the selected features and the key design aspects in the following list:

3-tier Architecture: Applications that are build with our library can be designed to follow a 3-tier architecture, with the additional feature that the presentation tier is *stateful* and highly responsive.

Transparent Replication: The library provides the underlying OT replication mechanism and handles the communication between the components transparently. From the user perspective, the propagation of updates of the state in the presentation tier is transparent.

Expressiveness: Even complex application state must be expressible and accessible with our library. To this end, the library must offer list, tree, and JSON data structures and the corresponding operations as programming interface.

Partition-Capability: The data in the presentation tier must remain accessible even in case the network is disrupted and no communication to the logic and data tier is possible. After

6 URL: `https://github.com/rbraeunlich/formic` under Apache License 2.0.

the network partition has *healed*, the buffered updates are exchanged and the replicas regain a consistent state.

Configurability: The library must provide mechanisms to configure basic properties without modifying the source code. The modifiable properties include connection details (addresses, ports), the buffer size, the number of threads, and database credentials.

We note that according to the definition of the presentation layer in Section 2.1.2, the typical functionality is interface-related. To reduce complexity, we further refine the scope of the applications that benefit from our library to *web applications*. Hence, the interface-related code is executed in the browser, similar to what we used in our *HotPi* prototype. We think that this restriction of the scope is justified and that we do not lose any generality. Consequently, the above stated list of selected features can be supplemented by:

Client-Server Architecture: The library offers a client part that runs autonomously in the browser, and a corresponding server part that implements the Wave OT control algorithm.

We note that this restriction to web-based applications implies a restriction of the usable technologies, for example the client part of the library must somehow be based on JavaScript in order to achieve an autonomous interface. As a consequence, the above stated *Partition-Capability* could be restated to *Offline-Capability*, because a partitioned web-based interface is essentially an *offline* web application.

With the above stated list of selected features and architectural properties, we presented an outline of the capabilities of our library. From that outline we derive a tailored architecture, which we present in the following subsection.

4.5.2 *Library Design and Architecture*

As mentioned in the previous subsection, the client and the server part of the library follow different purposes. For example, the server part must be designed to handle a high degree of parallelism, whereas the client part focuses on the convenience for the user,

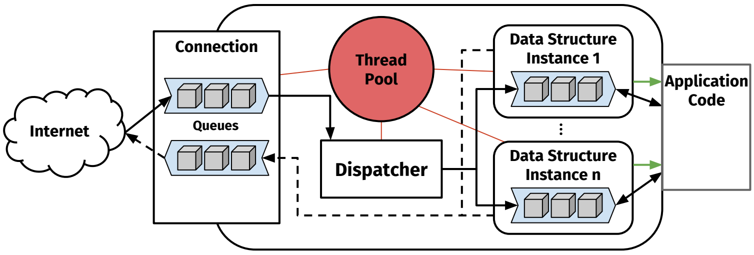


Figure 4.17: Client architecture of our *formic* prototype (from [Brä17]).

which includes the provided mechanisms to access and modify the shared data structures. Therefore, we derive a tailored architecture for the client and the server part, which fulfills the outlined demands.

We begin with the design of the client architecture, where we note a *one-to-many* relationship between the server and the used data structures. In this case, there is only one communication channel that needs to be maintained, i.e. the channel between the server and the client. In contrast, there can be arbitrary many data structures that are used by the client and where consistency needs to be maintained. To this end, we derive an architecture where we have one component that is responsible for the communication and one component for each maintained data structure instance. We visualize the client architecture in Figure 4.17, where we also show the interaction between the selected components. In the figure we visualize the incoming operations with regular arrows, the outgoing operations with dashed arrows, and synchronous calls with left right arrows. We note that the communication component consists of two message queues, one for incoming and one for outgoing messages. These queues are used to maintain the necessary *Offline-Capability*, so that the client can continue to work even if there is no connection to the server. In this case, the messages are buffered in the queue and sent out once the connection is reestablished. Similarly, we use message queues for each remote operation on a data structure instance so that we can utilize a thread pool to avoid the overhead of maintaining a single thread for each data structure instance and the bottleneck of using a single thread for the whole client. We visualize the thread

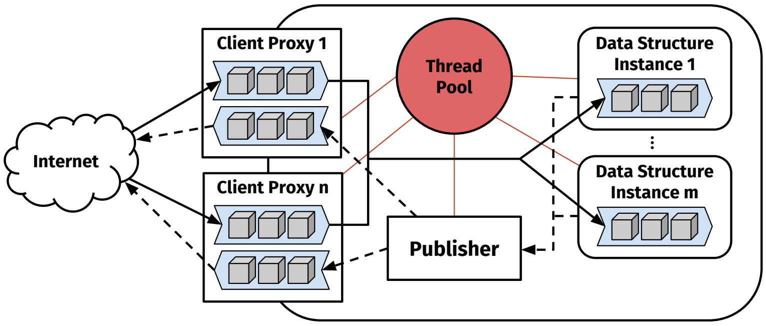


Figure 4.18: Server architecture of our *formic* prototype (from [Brä17]).

provisioning of the autonomous components with a thin red line to the thread pool.

In addition to the connection and data structure handling, the client architecture features a *dispatcher* component that distributes incoming messages to the corresponding queue of the data structure instance. Here, the incoming messages are dispatched and mapped to operations on the corresponding data structure instances.

An application interacts with the data structure instances by using the provided API, which exposes the operations on the data structures, e.g. insert, delete, and replace in case of the JSON data structure. In this case, the operations are immediately applied on the data structure instance and placed in the outgoing queue of the connection component, which asynchronously transmits the operations to the server. Since the local operation is immediately applied, we design and visualize the communication between the application and the data structure instance as a *synchronous call*. In case a data structure instance is modified by a remote operation, the application can be notified over a registered callback function, which we visualize with green arrows. With this mechanism, the application can be informed about changes in the background and can, for example, trigger an update of the user interface.

The corresponding server part of the library, which we visualize in Figure 4.18, follows a similar architectural pattern. The major difference is that the server needs to maintain connections to multiple clients with arbitrary many data structure instances. Hence, there exists an n to m relation between the connections and the accessed data objects. As shown in the figure, there is one *Client Proxy* component for each connected client. Here, the incoming and outgoing messages are buffered and asynchronously transmitted to the clients over separate connections. In contrast to the client architecture, we introduce an additional component to the server, namely the *publisher*. In this component, the server tracks the subscriptions of the clients to the data structure instances. Based on the stored subscriptions, the server assigns incoming and outgoing operations to the corresponding message queues. We note that the server architecture does not include any functionality to initiate an operation to a data structure instance. Consequently, there is no API compared to the client architecture.

The major challenge that needs to be addressed by the implementation of the server is that there is a potentially high degree of parallelism. That is why we again include a thread pool to achieve a better utilization of the server's resources.

4.5.3 *Library Prototype*

With *formic*, we present a prototypical implementation of the aforementioned architecture. Consequently, the library is divided into a server- and a client implementation, which slightly differ with respect to the used technologies. The main reason for this distinction is that both parts have a different purpose and therefore different needs.

For the communication between the server and the clients we decided to use WebSocket connections. WebSockets, in general, offer a two-way communication protocol which is especially useful in our case, because the clients can get notified by the server in case there are new remote operations [Fet11]. All messages between the server and the client are then serialized to JSON, which is common practice in modern web development. We note that WebSockets together with JSON formatted messages enables the possibility to easily replace one component, i.e. the server or the client part. Both technologies are rather standard and easy to use with other web frameworks.

For the server part, we decided to use the Akka Framework together with the Scala programming language to achieve high performance in the presence of high concurrency. The abstraction of Akka actors essentially allows the implementation of independent units, which can communicate to other units (or actors) via messages only. Hence, there is no complex method invocation possible, which allows to hide complex low-level details such as waiting or locking. This is particularly advantageous when we instantiate several data structures and client connections. The persistence of the data structures instances, i.e. the state, is realized with the Akka Persistence extension, which allows to recreate the state of an actor in case something crashed or needed to be restarted. The key idea is that all operations on a data structure are stored persistently—most likely in the data tier—and that the operations can be replayed upon reinstantiation of a actor, also known as *event-sourcing*.

For the client part, however, we use essentially the same technology stack, i.e. Akka and Scala, with the particular difference that we use ScalaJS to compile our implementation to JavaScript. This way, we achieve that the client part of *formic* can autonomously run in the browser.

Both the client and the server part implement the corresponding Wave OT control algorithm without exposing any transformation to the user. With that in mind, the underlying replication mechanism is completely hidden and an application designer can focus on the relevant business logic. We note that we allow a user-defined configuration of *formic*'s essential parameters, e.g. the buffer sizes, the thread pool size, the IP address & port of the server, and the credentials for the database in the data tier.

Ultimately, we would like to emphasize that with our library *formic*, we enabled a more comfortable way to develop applications that process and handle more state in the presentation tier. In conjunction with our JSON extension of OT, *formic* provides the necessary expressiveness and convenience to implement our approach in web-based applications.

4.6 EVALUATION

As a remaining step we evaluate our library in terms of performance in order to identify the boundaries for applications that store more state in the presentation tier. Therefore, we split our evaluation in two parts: (1) a comparison to a document editing scenario in

Google Docs, and (2) a comparison to the ShareDB library, which offers similar features.

4.6.1 *Large Scale Document Editing*

For this first part of the evaluation we reuse an experiment of Dang and Ignat [DI16], which was initially used to explore the performance of Google Docs at large scale. In their experiment, real users have been simulated with Selenium, a widely accepted web-based testing tool [HK06]. The simulated users are divided into one Writer, one Reader, and up to 50 DummyWriters. The DummyWriters write random strings to a shared document. The Writer writes a specific string to the document and the Reader waits until the specific string is present and reports the delay. Dang and Ignat measured the delay with different numbers of DummyWriters and various type speeds (1-10 keystrokes per second).

For our evaluation of *formic*, we recreated Dang and Ignat's setup by installing the *formic* server and the Selenium users on several virtualized machines on a local OpenStack cluster (16 servers with 2 x Intel Xeon X5355 (2x4 cores), 32GB memory). Every used instance (2vCPU, 2GB Memory) ran up to five DummyWriters; each instantiating the Chrome browser and behaving like a human client. The Writer and the Reader were, as in the original experiment, always placed on the same instance to guarantee a consistent clock. The formic server was placed on a virtual instance with 4 vCPUs and 8GB memory.

We note that the original experiment design assumes operations that insert single characters to a document where no further formatting is used. Hence, we decided utilize *formic*'s list operations and the corresponding transformation function XFORM_L to achieve the required functionality of the experiment. The list operations can be seen as operations on a sequence of characters, which ultimately forms a document. A detailed evaluation of our JSON extension, however, will be in focus in the next subsection.

In Figure 4.19 we show the results of an experiment run where each user injects one character per second. On the x-axis we see the number of active users, i.e. DummyWriters, that insert characters. Along the y-axis we see the delay in seconds until the particular Reader has observed a special character from the Writer.

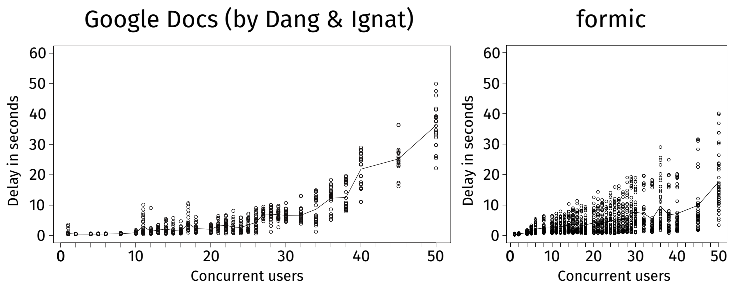


Figure 4.19: Collaborative editing with a type speed of one character per second on Google Docs (from [DI16]) compared to *formic*.

We note that the observed delays are relatively stable and below five seconds for less than 30 active users. Above that, the delays increase very fast, especially at a scale of 45 or 50 simultaneous users. The observation that can be made is, that our library *formic* struggles with simultaneous editing sessions at large scale. Fortunately, this behavior is not unique to *formic* and can also be observed for other OT-based collaboration systems. In the original experiment from Dang and Ignat, the authors reported a similar performance decrease in Google Docs at large scale, which we visualize in Figure 4.19.

In direct comparison of the results of Dang and Ignat to our measurements, our library is able to compete with Google Docs and even outperforms it at high scale. We find this surprising, since Google Docs can be seen as the de facto standard online collaboration tool. However, we admit that there is a certain bias in this comparison, because the features that are provided by Google Docs significantly exceed the capabilities that were offered by *formic* in this experiment. We note that the reason for the observable decrease of the performance is a consequence of the high degree of parallelism inside the server. This is, in fact, a bottleneck and our measurements can be interpreted as a confirmation of Dang and Ignat's result that OT systems, that rely on a server to sequence operations, are limited in performance at high scale. We further discuss the consequences in our discussion in Section 4.6.3, where we also motivate the scenarios in which high-scaling collaborative application are needed.

In the original experiment design, the authors also evaluated the performance for up to 10 keystrokes per second. In essence, all runs showed a similar growth of the delays with respect to the number of users. That is why we omit the presentation and comparison of the results of our measurement of *formic* in this section and refer to the appendix in Section A.3, where show our findings for the remaining type speeds.

4.6.2 *JSON operations and ShareDB*

In contrast to Google Docs, *formic* offers the OT mechanism in a way that web developers can enable simultaneous and collaborative editing of arbitrary objects, as long as the objects can be serialized into JSON. In order to evaluate the transformation of operations on JSON objects properly, we decided to compare the performance of *formic* to ShareDB [SG15] in an collaborative JSON editing scenario. ShareDB is a JavaScript project that offers similar features as *formic*. Both tools implement the Wave Control algorithm and offer a client and a server part to exchange operations on a replicated data structure.

For this run we modified Dang and Ignat's experiment design so that the DummyWriters are now invoking operations to a shared JSON object over a test website. To ensure comparability, we implemented an identical test website with *formic* and ShareDB and installed both systems on the same local cluster.

In Figure 4.20, we show our measurements of ShareDB and *formic* in a collaborative JSON editing scenario where we inject one modification per second from each DummyWriter. In general, we see that our library is not able to keep up with ShareDB. There are, however, two interesting observations to be made. First, we note that the performance of both libraries indicates a fast linear growth of the delay with respect to the number of concurrent users, which confirms the boundaries of such OT based systems at large scale. Second, the observed delays are still rather small compared to the measurements of Google Docs in Figure 4.19. Moreover, we find it interesting that the difference between *formic*'s delays in document and JSON editing, i.e. the difference between the right side of Figure 4.19 and the right side of Figure 4.20, is rather small, which indicates that our implementation has way more impact than the type of the data structure

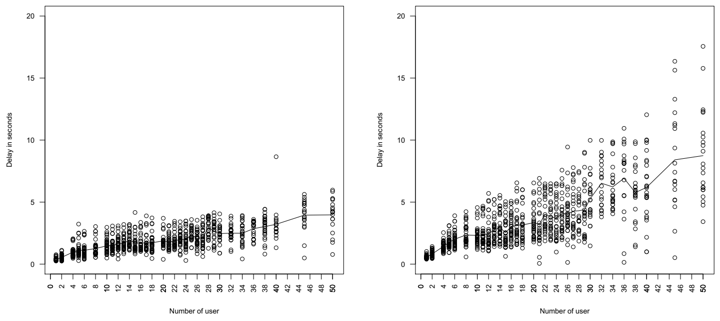


Figure 4.20: JSON editing with one modification per second for ShareDB (left) and *formic* (right).

and the corresponding transformation function. Surprisingly, the delays in the document editing scenario are a little higher than in the JSON editing scenario, which we expected to be the other way around. The only reasonable explanation that we could find is, that handling smaller objects in a hierarchy, as in the JSON experiment, is computationally less expensive than handling one big object, i.e. the document, in the preceding experiment.

We admit that our prototype leaves room for improvement when comparing it with ShareDB. Therefore, we discuss the superiority of ShareDB and the resulting opportunities to improve our prototype in the next subsection.

4.6.3 *Discussion*

With respect to text editing scenarios, we can confirm the finding of Dang and Ignat that the performance of OT in collaborative web applications is limited at large scale. However, the performance of *formic* is comparable with Google Docs. We note that the used local cluster for the evaluation of *formic* is relatively old. Hence, we would expect even better results with modern hardware. Unfortunately, Google provides no insight into the used infrastructure and the underlying OT implementation and it is therefore difficult to reason about the performance results of Google Docs.

In the JSON editing scenario, our library performs worse than the competitor ShareDB. We explain the difference in the performance by the used optimizations in ShareDB which are not implemented in *formic* yet. For example, multiple operations on the local replica can be combined before they are sent to the server. This reduces the amount of necessary communication and leads to faster response times. However, since we again compare a research prototype to an established tool[7], we are quite satisfied with *formic*'s performance and judge our our transformation-based approach to edit JSON objects as successful.

One major bottleneck in *formic* is the mapping of a JSON object to an ordered n-ary tree. The mapping enforces a total order in every layer of the tree, which is technically not necessary for every JSON component. For example, key/value pairs inside a JSON object do not require ordering, whereas elements inside an array must be ordered. This issue can be solved by introducing a more complex data model that is directly tailored to JSON. The most interesting solution would use a combination of different consistency control systems to best suit the JSON definition, e.g. a combination of the OR-Set CRDT [Sha+11b] and OT.

One particular strength of our library is that we base our transformation of JSON operations on a verified transformation function, namely XFORM_T. In contrast, ShareDB offers no proof that the underlying transformation works flawlessly for every edge case, even though we did not find any violation in our experiments. However, we find it curious that ShareDB's *delete* and *replace* operation on the JSON data type require to include the item that is deleted or replaced as a parameter [Gen11]. In *formic*, and ultimately in our transformation of tree operations, we only transform the access path and avoid transforming the state of an object itself. The authors of ShareDB also identified this to be an issue and proposed an updated version of their API, which, unfortunately, was never implemented in ShareDB [Gen12]. We have no insights why this idea was abandoned, but we think that our transformation function and our approach to translate JSON operations to tree operations actually solves this issue, because *formic*'s API is essentially identical to the idea that was proposed by the ShareDB authors. At this point,

7 ShareDB together with its predecessor ShareJS have over 6000 stars on GitHub, which indicates a significant interest of the open-source community in both tools.

we leave further investigation as future work, but we think that an integration of our transformation function in ShareDB is promising to solve the outlined issues.

Applicability, Feasibility, and Limitations

With our conducted experiments we confirmed both: the feasibility to store, process, and replicate more application state in the presentation tier, and the limitations of OT based system at large scale. The explored limitation, i.e. the growth of the replication lag with the number of concurrent users, raises the question whether a system similar to *formic* is applicable in general and what kind of applications must be excluded. We think, that the answer to this question depends on the intensity of concurrent access to the same object. The typical use-case for such systems include collaboration systems like, in fact, Google Docs, which works absolutely fine for most of the users, even though there are the outlined scalability drawbacks. However, even in such use-cases the limitations can be reached quite easily. For example, in 2013 a *Massive Open Online Course* asked all 40000 participants to register for the course by editing a Google Docs document. As a consequence of the lack of scalability, the course had to be canceled[8] [Jas13].

While this anecdote illustrates the limitations and where our approach should not be used, we think that the set of applications that can benefit from it is bigger than expected. One significant requirement to apply our approach in applications is that the used data is small enough to be processed in the presentation tier, i.e. on the client's device. Once that requirement is fulfilled, a variety of applications that enable modifications of a manageable number of users to a shared object can benefit from our approach. In addition to the already mentioned real-time collaboration tools, we think that mobile online games could benefit from our approach as well. In order to illustrate the opportunities, we implemented a browser-based battleship game that utilizes *formic* to allow immediate updates of the interface upon new interactions, while the operations on the battlefield are asynchronously sent to the server. Overall, with the experiment results and the outlined areas of applications we judge

The source code of the battleship game is freely available in the formic GitHub repository.

8 Ironically, it was a course on *The Fundamentals of Online Education*.

our approach to process and store more state in the presentation tier as successful.

We note that the required server that sequences the operations can be seen as a drawback, even though most of the communication in the web is still client-server based. As outlined in Section 4.2, a peer-to-peer based OT architecture requires a transformation function that satisfies TP2. Unfortunately, designing a TP2-valid transformation function is an error prone task, which we illustrate in the next section where we present the related work.

4.7 RELATED WORK

The initial idea of storing more state in the presentation tier emerged from early groupware and collaboration systems. In those systems, the collaborators expect high responsiveness of the shared documents, i.e. edit operations must be executed as fast as editing a document on the local hard drive. The only way to achieve this, especially in unreliable networks like the Internet, is if every collaborator maintains an own replica of the document that allows updates without waiting for confirmation from the other replicas [SS05]. Hence, collaborative systems require multi-leader replication (see Section 2.2.2) and face the consequences of the CAP dilemma [Bre00; GL02; Kle15]. That is why those challenges were addressed by several research groups in the distributed systems community.

Operational Transformation

OT has been introduced by Ellis and Gibbs in 1989 [EG89], followed by multiple decades of research around the mechanism and very valuable contributions from various groups; mostly in the *Computer Supported Cooperative Work* community. Prominent example applications that utilize OT are Google's document editing suite Google Docs [DR18] and the free competitor Etherpad [Fou18b].

The key idea of OT, as outlined in Section 4.2, is that local operations can be immediately applied on the state of a replica, and remote operations are transformed against the operations that were applied concurrently in order to include the effects of those. Ultimately, all replicas converge to the same state even though the operations were applied in different orders.

In order to achieve this, OT systems consist of control algorithms and transformation functions, which have been introduced by various researchers over the past 29 years. The different control algorithms include dOPT [EG89], Selective-undo [PK94], Jupiter [Nic+95], adOPTed [RNRG96], GOT [Sun+98], GOTO [SE98], SOCT2 [SCF97; SCF98], SOCT3/4 [Vid+00], SDT [LL04; LL08], COT [SS06; SS09], and Wave [DWL10]. Those algorithms can be categorized into two groups: either they require a central server that sequences the operations, or they work on a *peer-to-peer* basis. We note that the majority of web-based collaboration tools that utilize OT are based on those control algorithms that introduce a central server, e.g. Jupiter or Wave.

We followed this observation by tailoring our transformation function XFORM$_T$, and the corresponding mapping of JSON objects to n-ary trees, for web application. Therefore, we utilized Wave in *formic* and in our prototype of a collaborative patient documentation system. As we have illustrated in Section 4.2, the choice of the control algorithms predetermines the set of properties that must hold for the transformation function.

Transformation Functions for Complex Data Structures

The transformation functions that were introduced alongside the early groupware applications focused on collaborative text editing and were based on list operations [EG89; RNRG96; SE98; SCF97; LL04; Imi+03]. As a consequence, building more complex applications that exceed the capabilities of lists requires a new set of transformation functions that support arbitrary nested objects.

Davis et al. [DSL02] were, to our knowledge, the first that applied the OT approach on treelike structures. They extended operational transformation to support synchronous collaborative editing of documents written in dialects of SGML (Standard General Markup Language) such as XML and HTML. The authors introduced a set of structural operations with their associated transformation functions tailored for SGML's abstract data model *grove*. Their approach is followed by [Sun+06]; showing improvements in XML editing and implementations in collaborative business software. Ignat and Norrie introduced a similar approach by, namely treeOPT, where they apply the OT mechanism recursively on different document levels [IN03].

Oster et al. [Ost+06b] proposed a framework for supporting collaborative editing of XML documents. Their framework works similar to the Copy-Modify-Merge paradigm widely used in version control systems such as CVS. The synchronization of the replicated XML documents is based on Operational Transformation. They make also use of a positional addressing scheme of the XML elements; similar to our translation of JSON Pointers to access paths.

In contrast the aforementioned related work, we provide an alternative, more generic transformation function that is not shaped specifically for XML. Hence, with our transformation function we enable more general use-cases of hierarchical OT. From this, we were able to derive a transformation of operations on JSON objects. In addition, we presented our transformation functions in a programming language near notation, so that they can be easier implemented. We are, to the best of our knowledge, the first that introduced a verified transformation function that enables simultaneous editing of JSON objects.

Formal Verification

Ressel et al. identified two important properties of a transformation function that must be satisfied to achieve convergence in conjunction with certain control algorithms [RNRG96]. These properties are TP1 (Definition 9) and TP2 (Definition 10). It turned out, that server-based control algorithms work sufficiently if the transformation function satisfies TP1, without requiring TP2 [KK10]. Those systems are able to go without TP2, because the server sequences the operations and a transformation is only applied between two parties, i.e. the server and the client. Peer-to-peer control algorithms, however, require TP2 in addition to TP1.

Unfortunately, designing a TP2 valid transformation function is a challenging task which is prone to errors. In fact, most of the introduced transformation functions falsely claimed to satisfy TP2, even though they undertook a peer preview process. Imine et al. used the theorem prover SPIKE [BKR92] to identify counterexamples for the transformation functions that were introduced with dOPT, AdOPTED, SOCT2, and SDT [Imi+03]. In the same work, the authors introduced an own transformation function alongside with a SPIKE-based “proof” that TP2 is satisfied. Oster et al. later showed that the proof implementation of this transformation function was invalid because of incorrect assumptions [Ost+05].

As a consequence of the failed attempts to present a TP2 valid transformation function for collaborative editing, Randolph et al. further investigated the introduced OT systems and concluded that it is impossible to achieve TP2 in a meaningful transformation function [Ran+15]. Despite this impossibility result, the TTF OT system by Oster et al. provides, to the best of our knowledge, the only transformation function that claims TP2 validity for which no counterexample is known [Ost+06c]. The authors circumvent the conclusion of Randolph et al. by introducing a different notion of the transformation.

In contrast to the efforts of designing a TP2 valid transformation function, the work in this chapter focused on web systems which are mostly client-server based. Hence, we judge the requirement of a server to show the feasibility to store and process more data in the presentation tier as justified. Fortunately, proving the required TP1 validity is significantly easier. Even though the introduced transformation function by Ellis and Gibbs [EG89] had a tiny flaw [Sun+98; SCF98; RNRG96; Imi+03], all later introduced transformation functions that were mentioned in the previous paragraphs turned out to be TP1-valid. In addition to the machine-checked proofs by Imine et al., we confirmed this result in an Isabelle/HOL implementation for the transformation function that we presented in Listing 4.1 [JH15].

As a main contribution of this chapter, we introduced a TP1 valid transformation function for operations on n-ary trees. In the corresponding technical report, we provide a hybrid proof that is mostly hand-written but also relies on an Isabelle/HOL implementation of XFORM_L [JH15]. Oster et al. also presented a verification of their XML-based transformation function [Ost+06b]. They claim with respect to proving the correctness: “It is nearly impossible to do this by hand” and refer to an automated tool named VOTE to fulfill the challenge [Imi+06]. Compared to their work, we define the underlying model precisely for our purpose—the synchronization of changes on generic hierarchical objects. Moreover, we took the challenge of Oster et al. and managed to prove the correctness *by hand*. Our approach of verifying a tree-based transformation function is followed by Sinchuk et al., where the authors use Coq to verify that TP1 is satisfied [SCS16].

Libraries and Performance at Large Scale

In addition to the academic attention that OT received, we have seen various open-source projects that address the challenge to make OT more accessible, especially for web applications. The most interesting representative for us is ShareDB; an OT library based on JavaScript that allows an easy integration of live concurrent editing in web applications [SG15]. The library additionally supports concurrent editing of hierarchical JSON objects. As opposed to our generic approach, ShareDB only offers support for JSON-based hierarchical structures. Further exists, to our knowledge, no verification of the used transformation functions and no publication or abstracted documentation so that a reimplementation is rather difficult.

Another interesting open-source library is JOT (JSON Operational Transformation), which also supports simultaneous editing of JSON objects [Tau13]. Again, the authors omit an analysis of the necessary properties in order to guarantee convergence. With our library *formic*, we contributed an alternative to both mentioned libraries. Our library is based on verified transformation functions and is, moreover, capable of handling more generic tree structures than just JSON. Our library is also capable of offline-editing, which is currently not possible with ShareDB or JOT.

Dang and Ignat were, to our knowledge, the first who analyzed the performance of OT-based collaboration systems at large scale [DI16]. We followed their idea and used their experiment design to evaluate our library in direct comparison to ShareDB. Our evaluation is, to the best of our knowledge, the first in depth analysis of ShareDB at large scale.

Apart from OT, other multi-leader replication mechanism are currently used to store more state in the presentation tier. Most noteworthy are Conflict-Free Replicated Data Types [Sha+11b], which we extensively studied in the previous chapter (see Section 3.2). Several benchmarks have been conducted to show the suitability of CRDTs for document editing [BUS16; AN+11]. In recent work, Nédelec et al introduced a web-based collaborative editor CRATE that is based on CRDTs and enables collaboration without the need of a central server [NMM16]. Kleppmann and Beresford's JSON CRDT is very promising in that regard as well [KB17]. However, this CRDT has, to the best of our knowledge, not been utilized in a collaborative application. Most recently, Kleppmann et al. introduced an implementation of the JSON CRDT in form of the JavaScript library *automerge* [Kle+18]. The library also enables handling more state in the presentation tier and supports simultaneous editing of JSON objects. Unfortunately, the library was introduced after we conducted our experiments. Nevertheless, we think that an analysis of *automerge* at large scale is highly interesting, because we expect much better scalability properties compared to the server-based OT approach.

An alternative but noteworthy mechanism is Differential Synchronization by Neil Fraser [Fra09], which is a state based synchronization mechanism based on diffing and patching. An implementation of Jan Monschke demonstrates the applicability to JSON documents [Mon15]. So far, we have not seen much academic attention to it.

4.8 CHAPTER SUMMARY

In order to achieve our goal to show that storing and processing more state in the presentation tier is worth considering, a usable multi-leader replication mechanism for web-based applications is necessary. Therefore, we presented our extension of a replication mechanism that is extensively used in collaboration systems, namely Operational Transformation. Our extension enables simultaneous editing of JSON objects, which are omnipresent in today's web applications.

In order to provide the necessary confidence that the properties of our extension hold, i.e. the convergence of replicas, we introduced a transformation function for operations on ordered n-ary trees and verified the necessary convergence property TP1. The proof was carried out by hand for every combination of operations and parts of the verification were based on properties that have been proven in Isabelle/HOL. For the JSON extension we introduced a mapping between JSON objects and n-ary trees and a translation of JSON Pointers to paths inside the tree.

We showed the transferability of our *theoretical* approach to *practice* with our research prototype of a patient documentation system. This use-case was motivated by our observations at the Charité hospital in Berlin, where we have seen many paper-based documentation that could be replaced by our prototype. With our prototype we demonstrated that our JSON extension has the necessary versatility to be utilized in conjunction with modern web development in application domains other than groupware systems.

With *formic* we presented a library that provides our JSON extension together with a fully implemented OT system. This library can be utilized to implement new web applications that benefit from handling more state in the presentation tier without explicitly managing the replication or the propagation of operations.

We evaluated our library and our JSON extension of OT in an experiment design by Dang and Ignat [DI16] and compared our measurements of *formic* against the results of Google Docs in a collaborative editing scenario at large scale. As result, we were able to show that our library is able to keep up with Google Docs and even outperforms it at larger scales of 30 to 50 concurrent collaborators. More importantly, we were able to confirm that the performance drop at higher scale is an inherent artifact of server-based OT systems. In a direct comparison to ShareDB, a library that also offers an OT system for JSON objects, our experiments revealed that *formic* needs further improvement of the underlying engineering in order to keep up. Nevertheless, the fact that ShareDB also showed similar limitations at large scale is further evidence for the general limitations of server-based OT systems.

Ultimately, we judge our approach of utilizing end extending OT to store and process more state in the presentation tier as successful, as long as concurrent access to a replicated object is within the limits that were revealed by our evaluation. We see our prototype of a collaborative patient documentation system and our library that supports the development of new applications as both evidence and motivation that our approach is worth considering. Furthermore, we are happy to see that the contributions that were carried out in this chapter have been recognized in the scientific community [JH16; JH15; JCR17; JB17] and that we were able to discuss our ideas with the most influential researchers in the field.

5 OUTLOOK AND DISCUSSION

5.1 TRANSFERABILITY

The example applications that we developed in the previous chapters demonstrated that our approach is worth considering in particular application domains. While we analyzed those applications, i.e. an IMAP service and a patient documentation system, in depth, a discussion on how our approach can be adapted in other applications is necessary. This discussion, which we conduct in this section, focuses on what classes of applications are considerable for our approach and how the arising challenges must be addressed.

IT Services with a stateful Logic Tier

In Chapter 3 we analyzed the feasibility of using CRDTs as multi-leader replication mechanism for an IMAP service with a stateful logic tier. In contrast to the OT-based approach in Chapter 4, the fact that CRDTs require no central server to sequence the operations is more appealing and predisposed to be applied in the logic tier. Hence, we are convinced that CRDTs represent the most fruitful alternative to transfer our approach to other IT services with a stateful logic tier.

Based on the knowledge that we have gained when we designed the IMAP-CRDT, we judge the limitations of CRDTs in terms of expressiveness and capability to preserve invariants as the biggest obstacle. Hence, the biggest challenge when transferring our approach to other applications would be to define meaningful operations on a state representation that is composed of CRDT primitives, e.g. counters, sets, and registers. Unfortunately, this definition requires careful consideration of the application-dependent requirements and the desired results in the presence of concurrent updates. As mentioned in our presentation of the related work in Section 3.7, there are ongoing research projects that aim to simplify the development of CRDT-based applications by providing usable data type implementations and update propagation mechanisms [AB16; MVR15; Kle+18].

The mentioned JSON CRDT by Kleppmann and Beresford [KB17] represents, in our opinion, the most noteworthy contribution to further follow the idea of handling more state in the logic tier. The versatility of JSON—especially the capability to nest objects and order items in an array—is most beneficial when mapping the structure of an application's state to a CRDT. Nevertheless, even the JSON CRDT is not directly capable of enforcing application based invariants, e.g. that there are only a certain number of values or that the sum of the values never exceeds a certain threshold.

In essence, we judge that using CRDTs to model an application's state is generally easier if the application can tolerate a certain deviation from the defined invariants and the expected behavior of the operations, e.g. in our IMAP service *pluto* we allow a deleted folder to reappear if there were concurrent `APPEND` or `STORE` operations. If an application is less tolerant in that regard, applying our approach and designing operations on an application specific CRDT would be significantly more difficult and more likely to fail. However, we think that a general classification whether our approach can be applied with reasonable effort would be, at least at the current state of CRDT-related research, too imprecise to be meaningful. Nevertheless, we think that further investigation of the required properties to apply our approach is certainly valuable and we leave it open for future work.

Byzantine Fault Tolerance

The system model of CRDTs (see Section 3.2) assumes non-byzantine behavior of the replicas in order to guarantee convergence. We note that this assumption also influences the set of applications that can benefit from our approach. In general, we expect the applications in the logic tier to be robust and under control of the application's administrator. Nevertheless, an administrator must be aware of the possible *attacks* on the convergence guarantees and the resulting damage for the application.

Such attacks include sending wrong causality information, e.g. wrong vector clocks. If a replica purposely sends the wrong vector clocks, it is possible to dominate other replicas and revert the changes by imitating that two operations happened concurrently, even though the remote operation happened-before the local operation. In the opposite case, one replica can slow down other replicas by sending vector clocks *from the future* to feign that there are lost messages that need to be retransmitted. While the outlined attacks seem to be highly critical for our approach, it is questionable whether *traditional* approaches that bundle the application state in the data tier are better in that regard. For example, in a database management system with single-leader replication (see Section 2.2.2), a follower could easily disrupt the system by exposing itself as a new leader, resulting in divergence between the old leader and the imitation.

In essence, the possibility of our approach to decentralize the state across multiple replicas in the logic tier requires the same trust in the operators and administrators as needed when the state is processed in a centralized fashion within a single data center. For applications that require byzantine fault tolerance (BFT), there are few solutions that were recently introduced in the scientific community. For example, Shoker et al. introduced a combination of eventual consistent and strong consistent systems that can be used to validate that the results of the eventual consistent system are correct [SYB17]. Another option that gained a huge amount of attention in the last years are blockchains [Nak08]. Here, a blockchain similar to ethereum [Woo14] could be used to implement the state changing operations as smart contract. Blockchains, however, typically introduce a new set of challenges that need to be addressed [Und16]. The most important drawback, currently, is the required energy that is necessary to maintain the network and to solve the underlying consensus problem[1].

Transferability of Stateful Presentation Tiers

When it comes to storing and processing more state in the presentation tier, we think that our approach is easier to transfer to

1 By the time of writing this thesis, the amount of energy per year that is used to maintain the Bitcoin network is estimated to be 59 TWh, which is equivalent to the energy consumption of Colombia; a country with a population of over 49 million [Dig18]. While the required energy to maintain the network is that high, the throughput of Bitcoin is estimated to be between 2 and 3 transactions per second [Lux18], which is surprisingly low compared to Paypal (over 190) or Visa (over 1600) [Tod18].

other applications compared to our efforts for a stateful logic tier. The main reason is, that the de facto standard device to interpret the code in the presentation tier is, in fact, a web browser. With this unification, the set of the necessary technology to transfer our approach to other applications is reduced to JavaScript libraries that interact with the browser's capabilities. Moreover, client-server based multi-leader replication mechanisms are also applicable, because this architecture is still the predominant one for web-based applications. Hence, in addition to CRDTs, which we utilized in our idea of a stateful logic tier, OT systems can also be considered.

The major challenge when transferring our approach to other applications is to express the structure of the application's state as a datatype that is supported by a multi-leader replication mechanism. To this end, we judge our JSON extension of OT as promising in that regard. Moreover, the JSON CRDT [KB17] and the implementation for web applications *automerge* [Kle+18] can be seen as the CRDT-based pendant to our OT extension and *formic*.

The limitations of the transferability in terms of scalability have been analyzed in detail in Section 4.6. In essence, applications that require highly concurrent write access on the same object with more than 30 simultaneous users per second are suffering from the OT-inherent increase of the delay at large scale. Hence, applying our approach to those applications necessarily requires further investigation of the best suited replication mechanism. We think that CRDTs are promising in that regard, but a detailed analysis of the JSON CRDT for collaborative applications at large scale has, to the best of our knowledge, not yet been conducted and is certainly interesting for future work.

5.2 PERSPECTIVES

While the discussion in the previous section mainly focused on how existing applications can benefit from more state in the early tiers, we find it similar interesting to discuss the perspective for future applications that implement our approach.

5.2.1 *Decentralized Off-Cloud Services*

In Chapter 3 we introduced our idea of storing and replicating more state in the logic tier, mainly because to increase the reliability and to enable better performance at very large scale. The underlying multi-leader replication architecture allows replicas to stay available and responsive, even in the presence of network partitions. While our evaluation and the discussion in Section 3.5 and 3.6 already includes rather unconventional use-cases for our approach, for example geo-replication or hybrid-cloud setups, we think that the enabled possibilities are worth exploring.

One interesting idea that emerged from the blockchain community is the notion of a decentralized web or Web3 [Fou18c], where popular services that are currently operated in a centralized fashion, e.g. search engines like Google, social networks like Facebook, or communication services like WhatsApp, are replaced by fully distributed ones that run on shared infrastructure, i.e. a blockchain. On a closer look, the main purpose of the blockchain technology is to maintain a fault-tolerant consensus on particular transactions that can be executed by every participant and initiated by every node that contributes to the network. As a major benefit, everyone can participate in the network without requiring to *trust* the other participants.

Unfortunately, because the multi-leader replication mechanism that we explored in this thesis assume non-byzantine behavior, applying our approach requires a certain *trust* in the operators of the replicas. In contrast to blockchains, however, no expensive consensus mechanism is necessary as the replicas converge by design. Hence, if a group of operators share a certain mutual trust, designing and operating a decentralized service similar to Web3 is possible with our approach; without the significant blockchain-related overhead

In order to illustrate this idea, we reuse our IMAP example, which we extensively analyzed in Section 3.3. In this example, we assume a small group[2] of users that typically trust a smaller number of administrators that host the group's IMAP service on a public cloud. In contrast to this rather traditional approach, the group could use

2 This could be a small organization or a loosely coupled group of students that, for example, aim to improve the study programs at the university. In both cases, we assume that they need a reliable mail infrastructure.

a distributed IMAP service similar to *pluto* that is hosted by every member of the group on separate low-budged commodity infrastructure with Internet access. This infrastructure possibly includes smartphones, IoT-Devices like the Raspberry Pi, or wireless routers. We note that this decentralized deployment of an IMAP service on edge devices is only possible because of the utilized multi-leader replication that supports an arbitrary number of replicas without requiring expensive coordination like consensus.

As mentioned, the absence of BFT must be balanced with the amount of trust that is necessary between the members of the group. However, in this case, where the users are considered to be part of a group and share a common goal, we believe that this balance is possible to make. Moreover, further improvements to require less trust are certainly possible, for example by requiring operations to be signed by the replicas with a private key. We think that the idea of decentralizing software services to infrastructure beside public clouds and without requiring expensive consensus mechanisms, like a blockchain, is worth investigating. The use of multi-leader replication mechanisms outside the data tier is promising in that regard and the in depth analysis that we carried out in this thesis could contribute to the design of off-cloud services.

5.2.2 *Collaborative Web*

While our idea of decentralized off-cloud services mainly focused on our approach towards a stateful logic tier, further exploring the possibilities of a stateful presentation tier is promising as well. We note that the applications that are already utilizing stateful presentation tiers and multi-leader replication are, in fact, *collaborative* applications, i.e. applications where multiple users simultaneously edit the same document or object. With our JSON OT extension and *formic* we showed how web applications, e.g. patient documentation systems or games, can implement our approach and benefit from improved reliability. Nevertheless, we admit that both introduced use-cases are, in fact, pretty similar to what would be considered a collaborative application.

This observation raises the question whether web applications where only one user interacts with certain objects are excluded from our approach. One example for such applications are web mail clients, where only one user is supposed to interact with the mailboxes over the web interface. As a consequence, those web interfaces are not designed to be *collaborative* in any way.

On the other side, there is a growing number of Internet devices, such as mobile phones, tablets, or smart watches, that utilize web browsers or browser engines like WebKit [App18] to access and edit information on the web. Following the predictions of Gartner, there will be over 12 billion Internet-ready devices in the consumer category by 2020 [Gar18]. The amount of devices that a single person uses to access web services inevitably leads to situations where a user is in collaboration with him- or herself. This can be best illustrated with the aforementioned web mail client example:

In this example, we assume that a user starts to compose a new email message on a smartphone in a rural area without sufficient mobile Internet access. After the first few paragraphs, the user switches to a desktop computer with cable Internet access to write the last paragraphs. At this point, the paragraphs written on the smartphone are not synchronized with the desktop computer and therefore not accessible. We note that at this point a multi-leader replication mechanism is necessary to provide the certainty that the changes on the two devices eventually converge to a single, meaningful email. If the user interface, i.e. the presentation tier, would be designed to be *collaborative* in the first place, the situation could be handled like a shared document in Google Docs, where all changes are exchanged and merged when the devices reconnect with the server. In traditional interface designs, however, the user would end up with two separate mails that must be manually merged.

We find the perspective to design a web application to be collaborative, even though it is not an online collaboration application, highly interesting. This resulting *Collaborative Web*, a term that is currently not used for this purpose, could be seen as a next step in the evolution of the services we use. However, we admit that this perspective is highly visionary at this point and by no means an inevitable consequence. Nevertheless, the contributions made in this thesis are certainly valuable to further explore this idea.

6 CONCLUSION

In this thesis we analyzed the feasibility to store and process more state in the logic and presentation tier. During the exploration of our assertion that such novel placements of stateful components in a 3-tier architecture are worth considering, we identified and addressed the emerged challenges that originate from fundamental and unsolved problems in distributed systems research. Among others, the CAP dilemma represents a major restriction of the possibilities of distributed and decentralized services, especially with respect to the consistency of the state. With the contributions that were carried out with this thesis, we conduced to close the exposed gap of the CAP dilemma and demonstrated how standard IT services can benefit from our approach.

In order to substantiate the assertion of this thesis, we presented the necessary background in Chapter 2, followed by an exploration how multi-leader replication systems, and especially CRDTs, can be utilized to overcome the impossibility to keep a service with a stateful logic tier both available and consistent in the presence of failures. In Chapter 3, we exemplified our exploration with an IMAP service and therefore introduced a verified IMAP-CRDT and a research prototype to demonstrate that our approach can be transferred from theory to practice.

We are confident that the initial exploration of the feasibility of using CRDTs as multi-leader replication mechanism of an IMAP service can be considered successful. Along the way, we made two important contributions: a verified IMAP-CRDT design and the evaluation of our prototype, where we showed that the replication lag can be significantly reduced compared to *dsync*, the replication tool of the de facto standard IMAP server *Dovecot*.

We considered IMAP as the example to show the benefits of modeling standard IT services with CRDTs. Offering multi-leader replication without the need of manual conflict resolution enables not only the possibility of planet-scale distributed applications, but also more reliability in the presence of failures. To emphasize this further, this work convinced us that really any stateful IT service should be examined for applicability of multi-leader replication. Relying on strongly-consistent operations and fault-free infrastructure can get risky as state becomes ever more shared and clients distributed. CRDTs, combined with formal verification, offer the means to achieve confidence in relaxed consistency. Thus, considering this approach when designing and even upgrading large-scale IT services can be a matter of securing viability of a particular service—even in a single data center deployment.

Our exploration of the feasibility of storing and processing more state in the logic tier is followed by the corresponding analysis for the presentation tier in Chapter 4. Here, we explored how OT—another multi-leader replication mechanism—can be utilized to enable more reliable and autonomous user interfaces of web applications. On this way, we identified an essential obstacle that makes it unnecessary difficult to apply OT in modern web applications, i.e. the absence of the possibility to transform operations on JSON objects. To this end, we contributed a verified transformation function for operations on ordered n-ary trees, which we utilized to introduce our JSON extension of OT. The further contributions that resulted from the exploration in this chapter include a prototype application where we showed how our extension can be used in a collaborative patient documentation system, and our programming library *formic*.

With *formic*, we presented an open-source library that simplifies the development of web-based collaborative applications by providing a fully working OT system with implemented transformation functions for operations on lists, trees, and JSON objects. The conducted experiment demonstrated that our library is able to compete against Google Docs, the most successful collaborative application that utilizes OT. Moreover, we were able to reveal how much the underlying client-server architecture limits the performance of JSON-based OT systems at large scale.

In summary, this thesis contributed new insights on the challenges that arise when following an unconventional but yet promising application design that uses stateful components in the early tiers. The arising challenges were addressed with novel extensions to existing multi-leader replication mechanisms that can be reused, not only for the purpose that was in focus of this thesis, but also when applying those mechanisms in other domains. Our approach, where we began with the system design and verification, followed by the implementation and evaluation, turned out to be successful in this regard. The resulting prototypes combine *theory* and *practice* and are able to play off the conceptual benefits.

To conclude, we are convinced that the analyzed unconventional application designs are, in fact, considerable. We believe that the resulting opportunities and perspectives of such designs, which we briefly sketched in Chapter 5, are promising and worth exploring. Hence, we end this thesis by showing our gratitude that we were able to discuss our ideas and findings with the scientific community and are delighted to see that some of our contributions already made an impact on the work of other scientists and reputable colleagues.

BIBLIOGRAPHY

[Aba12] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE Computer* 45 (2), 37–42.

[AT03] Navid Aghdaie and Yuval Tamir. 2003. Fast Transparent Failover for Reliable Web Services. In: *Conference on Parallel and Distributed Computing and Systems* (PDCS), 757–762.

[AT09] Navid Aghdaie and Yuval Tamir. 2009. CoRAL: A Transparent Fault-tolerant Web Service. *Journal for System Software* 82 (1), 131–143.

[Aha+95] Mustaque Ahamad et al. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing* 9 (1), 37–49.

[AN+11] Mehdi Ahmed-Nacer et al. 2011. Evaluating CRDTs for Real-time Document Editing. In: *ACM Symposium on Document Engineering* (DocEng), 103–112.

[Akk+16] Deepthi Devaki Akkoorath et al. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In: *International Conference on Distributed Computing Systems* (ICDCS), 405–414.

[AB16] Deepthi Akkoorath and Annette Bieniusa. 2016. *Antidote: the Highly-available Geo-replicated Database with Strongest Guarantees.* *pages.lip6.fr*

[ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient State-Based CRDTs by Delta-Mutation. In: *Networked Systems* (NETYS), 62–76.

[ASB18] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta State Replicated Data Types. *Parallel and Distributed Computing* 111 (1), 162–173.

[ALaR13] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In: *European Conference on Computer Systems* (EuroSys), 85–98.

[Ama18] Amazon. 2018. *Elastic Block Store.* *aws.amazon.com*

[And+09] David Andersen et al. 2009. FAWN: A Fast Array of Wimpy Nodes. In: *Symposium on Operating Systems Principles* (SOSP), 1–14.

[App18] Apple. 2018. *WebKit: a fast, open source web browser engine.* *webkit.org*

[Att+16] Hagit Attiya et al. 2016. Specification and Complexity of Collaborative Text Editing. In: *Symposium on Principles of Distributed Computing* (PODC), 259–268.

[AB00] Luis Aversa and Azer Bestavros. 2000. Load Balancing a Cluster of Web Servers: Using Distributed Packet Rewriting. In: *International Performance, Computing, and Communications Conference* (IPCCC), 24–29.

[Bai+12] Peter Bailis et al. 2012. The Potential Dangers of Causal Consistency and an Explicit Solution. In: *ACM Symposium on Cloud Computing* (SOCC), 22:1–22:7.

[Bai+13] Peter Bailis, Ali Ghodsi, Joseph Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In: *International Conference on Management of Data* (SIGMOD), 761–772.

[BAL16] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 2016. The Problem with Embedded CRDT Counters and a Solution. In: *Principles and Practice of Consistency for Distributed Data* (PaPoC), 10:1–10:3.

[BAS14] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In: *Conference on Distributed Applications and Interoperable Systems* (DAIS), 126–140.

[Bas18] Basho. 2018. *Riak Key Value Database.* *basho.com*

[Bie+12] Annette Bieniusa et al. 2012. Brief Announcement: Semantics of Eventually Consistent Replicated Sets. In: *Distributed Computing* (DISC), 441–442.

[Bir94] Ken Birman. 1994. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *Operating Systems Review* 28 (1), 11–21.

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems* 9 (3), 272–314.

[Bis+16] Benjamin Bisping et al. 2016. Mechanical Verification of a Constructive Proof for FLP. In: *Interactive Theorem Proving* (ITP), 107–122.

[BEH14] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. 2014. Verifying Eventual Consistency of Optimistic Replication Systems. In: *Symposium on Principles of Programming Languages* (POPL), 285–296.

[BKR92] Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch. 1992. SPIKE, an Automatic Theorem Prover. In: *Conference on Logic Programming and Automated Reasoning* (LPAR), 460–462.

[Brä17] Ronny Bräunlich. "Collaborative Editing of JSON Objects based on Operational Transformation." Master's Thesis. Technische Universität Berlin, 2017.

[BRVR17] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2017. Saturn: A Distributed Metadata Service for Causal Consistency. In: *European Conference on Computer Systems* (EuroSys), 111–126.

[Bra14] Timothy Bray. 2014. *The JavaScript Object Notation (JSON) Data Interchange Format.* *RFC 7159*

[Bre00] Eric Brewer. 2000. *Keynote Presentation at PODC.* *podc.org*

[Bre12] Eric Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer* 45 (2), 23–29.

[Bre17] Eric Brewer. 2017. *Spanner, TrueTime and the CAP Theorem.* *research.google.com*

[BUS16] Loïck Briot, Pascal Urso, and Marc Shapiro. 2016. High Responsiveness for Group Editing CRDTs. In: *International Conference on Supporting Group Work* (GROUP), 51–60.

[Bry13] Paul Bryan. 2013. *JavaScript Object Notation (JSON) Pointer.* *RFC 6901*

[Bur06] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In: *Symposium on Operating Systems Design and Implementation* (OSDI), 335–350.

[Bus+96] Frank Buschmann et al. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing.

[Cha+08] Fay Chang et al. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems* 26 (2), 1:1–1:4.

[CBDM11] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. 2011. Formal Verification of Consensus Algorithms Tolerating Malicious Faults. In: *Stabilization, Safety, and Security of Distributed Systems* (SSS), 120–134.

[Che+15] Yunji Chen et al. 2015. Deterministic Replay: A Survey. *ACM Computing Surveys* 48 (2), 17:1–17:47.

[CS93] David Cheriton and Dale Skeen. 1993. Understanding the Limitations of Causally and Totally Ordered Communication. In: *Symposium on Operating Systems Principles* (SOSP), 44–57.

[Con14] Continuent. 2014. *Tungsten Replicator.* 🌍 *continuent.com*

[CDE+12] James Corbett, Jeffrey Dean, Michael Epstein, et al. 2012. Spanner: Google's Globally-distributed Database. In: *Conference on Operating Systems Design and Implementation* (OSDI), 251–264.

[Cri03] Marc Crispin. 2003. *Internet Message Access Protocol.* *RFC 3501*

[Cul+08] Brendan Cully et al. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In: *Symposium on Networked Systems Design and Implementation* (NSDI), 161–174.

[DI16] Quang-Vinh Dang and Claudia-Lavinia Ignat. 2016. Performance of real-time collaborative editors at large scale: User perspective. In: *IFIP Networking Conference and Workshops* (IFIP Networking), 548–553.

[DWL10] Alex Mah David Wang and Soren Lassen. 2010. *Google Wave Operational Transformation.* 🌍 *svn.apache.org*

[DSL02] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. 2002. Generalizing Operational Transformation to the Standard General Markup Language. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 58–67.

[DR18] John Day-Richter. 2018. *What's different about the new Google Docs: Making collaboration fast.* 🌍 *drive.googleblog.com*

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. 2007. Dynamo: Amazon's Highly Available Key-value Store. In: *ACM Symposium on Operating Systems Principles* (SOSP), 205–220.

[Dig18] Digiconomist. 2018. *The Bitcoin Energy Consumption Index.* 🌍 *digiconomist.net*

[Du+13] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In: *ACM Symposium on Cloud Computing* (SOCC), 11:1–11:14.

[Du+14] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In: *ACM Symposium on Cloud Computing* (SOCC), 4:1–4:13.

[EG89] Clarence Ellis and Simon Gibbs. 1989. Concurrency Control in Groupware Systems. *SIGMOD Record* 18 (2), 399–407.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR.

[Fet11] Ian Fette. 2011. *The WebSocket Protocol.* *RFC 6455*

[Fid88] Colin Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Australian Computer Science Communications* 10 56–66.

[FLP85] Michael Fischer, Nancy Lynch, and Michael Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM* 32 (2).

[Fou18a] Cloud Native Computing Foundation. 2018. *Sustaining and Integrating Open Source Technologies.* *cncf.io*

[Fou18b] Etherpad Foundation. 2018. *Etherpad.* *etherpad.org*

[Fou18c] Web3 Foundation. 2018. *Web3 is the vision of the serverless internet, the decentralised web.* *web3.foundation*

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc.

[Fow14] Martin Fowler. 2014. *Microservices.* *martinfowler.com*

[FB99] Armando Fox and Eric Brewer. 1999. Harvest, Yield, and Scalable Tolerant Systems. In: *Workshop on Hot Topics in Operating Systems* (HOTOS).

[Fra09] Neil Fraser. 2009. Differential Synchronization. In: *ACM Symposium on Document Engineering* (DocEng), 13–20.

[Gar18] Garner. 2018. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016.* *gartner.com*

[Gen11] Joseph Gentle. 2011. *JSON0 OT Type.* *ottypes/json0*

[Gen12] Joseph Gentle. 2012. *JSON2.* *josephg/ShareJS*

[GL02] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33 (2), 51–59.

[Gom+17a] Victor Gomes, Martin Kleppmann, Dominic Mulligan, and Alastair Beresford. 2017. *A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes.* Isabelle Archive of Formal Proofs. 🌍 *isa-afp.org*

[Gom+17b] Victor Gomes, Martin Kleppmann, Dominic Mulligan, and Alastair Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. In: *Proceedings of the ACM on Programming Languages* (OOPSLA), 109:1–109:28.

[GBR17] Chathuri Gunawardhana, Manuel Bravo, and Luis Rodrigues. 2017. Unobtrusive Deferred Update Stabilization for Efficient Geo-replication. In: *Usenix Annual Technical Conference* (ATC), 83–95.

[HN10] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In: *Symposium on Functional and Logic Programming* (FLOPS), 103–117.

[Has18] Hashnode. 2018. *MERN: Build production ready universal apps easily.* 🌍 *mern.io*

[Her13] Tobias Herb. "Collaboration via Convergent Replicated Data Structures." Master's Thesis. Technische Universität Berlin, 2013.

[Her91] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems* 13 (1), 124–149.

[HW90] Maurice Herlihy and Jeannette Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12 (3), 463–492.

[HK06] Antawan Holmes and Marc Kellogg. 2006. Automating functional tests using selenium. In: *IEEE Agile Conference*.

[IN03] Claudia-Lavinia Ignat and Moira Norrie. 2003. Customizable Collaborative Editor Relying on treeOPT Algorithm. In: *European Conference on Computer Supported Cooperative Work* (ECSCW), 315–334.

[Imi+03] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. 2003. Proving Correctness of Transformation Functions in Real-Time Groupware. In: *European Conference on Computer Supported Cooperative Work* (ECSCW), 277–293.

[Imi+06] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. 2006. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theoretical Computer Science* 351 (2), 167–183.

[Inc18] MongoDB Inc. 2018. *MongoDB Replication.* *docs.mongodb.com*

[Jas13] Scott Jaschik. 2013. *MOOC Mess.* *insidehighered.com*

[Jun14] Tim Jungnickel. “Formal Analysis of Collaboration via Convergent Replicated Data Structures.” Master’s Thesis. Technische Universität Berlin, 2014.

[JB17] Tim Jungnickel and Ronny Bräunlich. 2017. formic: Building Collaborative Applications with Operational Transformation. In: *Conference on Distributed Applications and Interoperable Systems* (DAIS), 138–145.

[JCR17] Tim Jungnickel, Juan Cabello, and Klemens Raile. 2017. HotPi: Open-Source Collaborative Patient Documentation. In: *ACM Conference on Computer-Supported Cooperative Work and Social Computing Companion* (CSCW), 219–222.

[JH15] Tim Jungnickel and Tobias Herb. 2015. *TP1-valid Transformation Functions for Operations on ordered n-ary Trees.* *arxiv.org*

[JH16] Tim Jungnickel and Tobias Herb. 2016. Simultaneous Editing of JSON Objects via Operational Transformation. In: *ACM Symposium on Applied Computing* (SAC), 812–815.

[JO17] Tim Jungnickel and Lennart Oldenburg. 2017. pluto: The CRDT-Driven IMAP Server. In: *Workshop on Principles and Practice of Consistency for Distributed Data* (PaPoC), 1:1–1:5.

[JOL17a] Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. 2017. Designing a Planetary-Scale IMAP Service with Conflict-free Replicated Data Types. In: *Conference on Principles of Distributed Systems* (OPODIS), 23:1–23:17.

[JOL17b] Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. 2017. *The IMAP CmRDT.* Isabelle Archive of Formal Proofs. *isa-afp.org*

[Kle15] Martin Kleppmann. 2015. *A Critique of the CAP Theorem.* *arxiv.org*

[Kle16] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O’Reilly.

[KB17] Martin Kleppmann and Alastair Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28 (10), 2733–2746.

[Kle+18] Martin Kleppmann et al. 2018. *automerge.* *automerge*

[Kub18] Kubernetes. 2018. *Production-Grade Container Orchestration.* *kubernetes.io*

[Kul+14] Sandeep S. Kulkarni et al. 2014. Logical Physical Clocks. In: *Conference on Principles of Distributed Systems* (OPODIS), 17–32.

[KK10] Santosh Kumawat and Ajay Khunteta. 2010. A Survey on Operational Transformation Algorithms: Challenges, Issues and Achievements. *International Journal of Computer Applications* 3 (12), 30–38.

[Lad+92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10 (4), 360–391.

[LM10] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *Operating Systems Review* 44 (2), 35–40.

[Lam78] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21 (7), 558–565.

[Lam79] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* 28 (9), 690–691.

[LPS09] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. 2009. *CRDTs: Consistency without Concurrency Control.* *arxiv.org*

[LL04] Du Li and Rui Li. 2004. Preserving Operation Effects Relation in Group Editors. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 457–466.

[LL08] Du Li and Rui Li. 2008. An Approach to Ensuring Consistency in Peer-to-Peer Real-Time Group Editors. *Computer Supported Cooperative Work* 17 (5-6), 553–611.

[Liu+14] Yang Liu, Yi Xu, ShaoJie Zhang, and Chengzheng Sun. 2014. Formal Verification of Operational Transformation. In: *International Symposium on Formal Methods* (FM), 432–448.

[Llo+11] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In: *Symposium on Operating Systems Principles* (SOSP), 401–416.

[Llo+13] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In: *Conference on Networked Systems Design and Implementation* (NSDI), 313–328.

[Lux18] Blockchain Luxembourg. 2018. *Confirmed Transactions Per Day.* *blockchain.info*

[MUW10] Stéphane Martin, Pascal Urso, and Stéphane Weiss. 2010. Scalable XML Collaborative Editing with Undo. In: *On the Move to Meaningful Internet Systems* (OTM), 507–514.

[Mat88] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In: *International Workshop on Parallel and Distributed Algorithms*, 120–131.

[MIM15] Frank McSherry, Michael Isard, and Derek Murray. 2015. Scalability! But at What Cost? In: *Workshop on Hot Topics in Operating Systems* (HOTOS).

[MVR15] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A Language for Distributed, Coordination-free Programming. In: *International Symposium on Principles and Practice of Declarative Programming* (PPDP), 184–195.

[Mei+17] Christopher Meiklejohn et al. 2017. Practical Evaluation of the Lasp Programming Model at Large Scale: An Experience Report. In: *International Symposium on Principles and Practice of Declarative Programming* (PPDP), 109–114.

[Mil16] Ross Miller. 2016. *Gmail now has 1 billion monthly active users.* *theverge.com*

[Mon15] Jan Monschke. 2015. *DiffSync.* *janmonschke/diffsync*

[MCT08] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Ef?Ciently. In: *International Symposium on Computer Architecture* (ISCA), 289–300.

[Nak08] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system.* *bitcoin.org*

[NMM16] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 2016. CRATE: Writing Stories Together with Our Browsers. In: *International Conference Companion on World Wide Web* (WWW), 231–234.

[Néd+13] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing. In: *ACM Symposium on Document Engineering* (DocEng), 37–46.

[Nic+95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In: *Symposium on User Interface and Software Technology* (UIST), 111–120.

[Nic17] Shaun Nichols. 2017. *AWS's S3 outage was so bad Amazon couldn't get into its own dashboard to warn the world - Websites, apps, security cams, IoT gear knackered.* theregister.co.uk

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag.

[Ora18] Oracle. 2018. *MySQL Replication.* dev.mysql.com

[Ost+05] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2005. *Proving Correctness of Transformation Functions in Collaborative Editing Systems.* hal.inria.fr

[Ost+06a] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data Consistency for P2P Collaborative Editing. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 259–268.

[Ost+06b] Gérald Oster, Hala Skaf-Molli, Pascal Molli, and Hala Naja-Jazzar. 2006. *Supporting Collaborative Writing of XML Documents.* hal.inria.fr

[Ost+06c] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In: *Conference on Collaborative Computing: Networking, Applications and Worksharing* (CollaborateCom), 1–10.

[Par08] Craig Partridge. 2008. The Technical Development of Internet Email. *IEEE Annals of the History of Computing* 30 (2), 3–29.

[Pos82] Jonathan Postel. 1982. *Simple Mail Transfer Protocol.* RFC 821

[Pos18] PostgreSQL. 2018. *High Availability, Load Balancing, and Replication.* postgresql.org

[PK94] Atul Prakash and Michael Knister. 1994. A Framework for Undoing Actions in Collaborative Systems. *ACM Transactions on Computer-Human Interactions* 1 (4), 295–330.

[Pro18] Prometheus. 2018. *Monitoring system & time series database.* *prometheus.io*

[Ran+15] Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. 2015. On Synthesizing a Consistent Operational Transformation Approach. *IEEE Transactions on Computers* 64 1074–1089.

[Res08] Peter Resnick. 2008. *Internet Message Format.* *RFC 5322*

[RNRG96] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. 1996. An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 288–297.

[Roh+11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *Parallel and Distributed Computing* 71 (3), 354–368.

[RGO12] Arnon Rotem-Gal-Oz. *SOA Patterns*. Manning Pubications Co.

[SS05] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *ACM Computing Surveys* 37 (1), 42–81.

[Ser17] Amazon Web Services. 2017. *Summary of the Amazon S3 Service Disruption in the Northern Virginia Region.* *aws.amazon.com*

[Sha+11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* *hal.inria.fr*

[Sha+11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems* (SSS), 386–400.

[Sha+18] Marc Shapiro et al. 2018. *Just-Right Consistency: reconciling availability and safety.* *arxiv.org*

[Shi17] Laura Shin. 2017. *Why Bitcoin's Greatest Asset Could Also Spell Its Doom.* *forbes.com*

[SYB17] Ali Shoker, Houssam Yactine, and Carlos Baquero. 2017. As Secure As Possible Eventual Consistency: Work in Progress. In: *Workshop on Principles and Practice of Consistency for Distributed Data* (PaPoC), 5:1–5:5.

[Sie07] Rob Siemborski. 2007. *The Post Office Protocol (POP3).* *RFC 5034*

[SCS16] Sergey Sinchuk, Pavel Chuprikov, and Konstantin Solomatov. 2016. Verified Operational Transformation for Trees. In: *International Conference on Interactive Theorem Proving* (ITP), 358–373.

[SG15] Nate Smith and Joseph Gentle. 2015. *ShareDB.* *sharedb*

[Spi10] Daniel Spiewak. 2010. *Understanding and Applying Operational Transformation.* *codecommit.com*

[SCF97] Maher Suleiman, Michèle Cart, and Jean Ferrié. 1997. Serialization of Concurrent Operations in a Distributed Collaborative Environment. In: *Conference on Supporting Group Work: The Integration Challenge* (GROUP), 435–445.

[SCF98] Maher Suleiman, Michèle Cart, and Jean Ferrie. 1998. Concurrent Operations in a Distributed and Mobile Collaborative Environment. In: *International Conference on Data Engineering* (ICDE), 36–45.

[Sun02] Chengzheng Sun. 2002. Undo As Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interactions* 9 (4), 309–361.

[SE98] Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 59–68.

[SXA14] Chengzheng Sun, Yi Xu, and Agustina Agustina. 2014. Exhaustive Search of Puzzles in Operational Transformation. In: *ACM Conference on Computer Supported Cooperative Work & Social Computing* (CSCW), 519–529.

[Sun+98] Chengzheng Sun et al. 1998. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interactions* 5 (1), 63–108.

[Sun+06] Chengzheng Sun et al. 2006. Transparent Adaptation of Single-user Applications for Multi-user Real-time Collaboration. *ACM Transactions on Computer-Human Interaction* 13 (4), 531–582.

[SS06] David Sun and Chengzheng Sun. 2006. Operation Context and Context-based Operational Transformation. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 279–288.

[SS09] David Sun and Chengzheng Sun. 2009. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems* 20 (10), 1454–1470.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc.

[Tau13] Joshua Tauberer. 2013. *JSON Operational Transformation (JOT)*. *JoshData/jot*

[Ter+94] Douglas Terry et al. 1994. Session Guarantees for Weakly Consistent Replicated Data. In: *International Conference on Parallel and Distributed Information Systems* (PDIS), 140–149.

[Ter+95] Douglas Terry et al. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In: *ACM Symposium on Operating Systems Principles* (SOSP), 172–182.

[Tod18] Altcoin Today. 2018. *Bitcoin and Ethereum vs Visa and PayPal – Transactions per second*. *altcointoday.com*

[Und16] Sarah Underwood. 2016. Blockchain Beyond Bitcoin. *Communications of the ACM* 59 (11), 15–17.

[Vid+00] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. 2000. Copies Convergence in a Distributed Realtime Collaborative Environment. In: *ACM Conference on Computer Supported Cooperative Work* (CSCW), 171–180.

[VV16] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Computing Surveys* 49 (1), 19:1–19:34.

[WUM10] Stephane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems* 21 (8), 1162–1174.

[Woo14] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*.

[ZBPH14] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In: *Formal Techniques for Distributed Objects, Components, and Systems* (FORTE), 33–48.

[gRP18] gRPC. 2018. *A high performance, open-source universal RPC framework*. *grpc.io*

A APPENDIX: OPERATIONAL TRANSFORMATION

A.1 THE WAVE CONTROL ALGORITHM

In both prototypes *HotPi* and *formic*, we used the Wave [DWL10] control algorithm in conjunction with the transformation functions we introduced in Section 4.3. Alongside with the sketched mechanics of the Wave algorithm in Section 4.2, we provide a more detailed description of the algorithm in pseudocode in Listing A.1 and A.2 based on the descriptions in [Her13; Spi10].

We start with the description of the Wave server part in Listing A.1. In the pseudocode of the server we see, that the server maintains a history of operations, which can be seen as sequenced version of all (possibly concurrent) operations that happened. We note that the server's history might differ from the order of operations executed at a client, because the operations that were initiated at a client will always be executed before the operations that happened concurrently. However, the concurrent operations from other clients are always in the same order as on the server. This way, causal communication is achieved.

We note that the server's broadcast function can be implemented in different ways, for example by queuing the operations until a client polls the next time, or by sending the operations over a WebSocket connection.

We present the client part of the Wave algorithm in Listing A.2. We note that the pseudocode is divided into two parts: one function to receive an operation from the server, and one function to initiate an operation. Here, the global variables are slightly more complex, because the list of operations that are not transmitted to the server yet, i.e. the *bridge*, is further divided into the *in flight* operation and the buffer.

We note that the function to send the operations to the server can be executed asynchronously, i.e. the client does not need to wait for the server's approval before applying an operation locally. This is particularly important, because otherwise the client would be

Listing A.1: Server part of Wave (adapted from [Her13]).

```

# history of all applied operations
history = []

def wave_server_receive_operations(remote_op, revision_number):

  # compute all concurrent operations from the history
  concurrent_ops = sublist(history, revision_number)

  for op in concurrent_ops:
    # transformation of the remote operation against the
    # concurrent operations
    remote_op = fst(XFORM(remote_op, op))

  # add the transformed version of the remote operation
  # to the history
  add(history, remote_op, revision_number)

  # sends the transformed version asynchronously to the clients,
  # the client that initiated the operation receives
  # an acknowledgment
  broadcast(remote_op)
```

blocked until the server has confirmed an operation. If the client receives a new operation from the server, the server's operation is transformed against the bridge, i.e. the concurrent operations that happened on the client, before it is applied locally.

While we find the presented pseudocode of the Wave algorithm particularly useful to understand the mechanics, for a more detailed description, however, we refer to the Wave documentation [DWL10] and further literature [Spi10].

A.2 THE REMAINING TREE TRANSFORMATION FUNCTIONS

In Section 4.3 we introduced a transformation function for replace$_T$ operations on n-ary trees and omitted a details of the transformation for combinations of insert$_T$ and delete$_T$, mainly because those transformations have already been introduced in a previous thesis [Jun14]. For the sake of completeness, however, we state the remaining transformation functions in Listing A.3, A.4, and A.5. We note, that the correctness with respect to TP1 has been proven in detail in our corresponding technical report [JH15].

Listing A.2: Client part of Wave (adapted from [Her13]).

```

# the operation that has been sent to server
# but not yet acknowledged
inflight_operation = None

# the local operations that have been applied but
# not yet sent to the server
buffer = []

# the number of operations received from the server
revision_number = 0

def wave_client_initiate_operations(operation):

  if inflight_operation == None:
    inflight_operation = operation
    send_to_server(operation, revision_number) # asynchronous
  else:
    buffer.append(operation)

  applyLocal(operation)

def wave_client_receive_operation(operation):

  revision_number = revision_number + 1

  if is_acknowledge(operation):
    # send the next operation if the server has acknowledged the
    # previous operation
    if buffer != []:
      inflight_operation = buffer.pop()
      send_to_server(inflight_operation, revision_number)

    else:
      inflight_operation = None

  else:
    # the server's operation happened concurrently to the local
    # operations in the bridge

    if(inflight_operation != None):
      # transform the server's operation against the bridge
      operation = fst(XFORM(operation, inflight_operation))

      for op in buffers:
        operation = fst(XFORM(operation, op))

    applyLocal(operation) # apply the transformed version
```

Listing A.3: Pseudocode of the transformation of insert$_T$ against insert$_T$.

```
function XFORMT(insertT(t1, pos1), insertT(t2, pos2)):
  TP = TPt(pos1, pos2)

  if effectIndependent(pos1, pos2):
    return(insertT(t1, pos1), insertT(t2, pos2))

  if pos1[TP] > pos2[TP]:
    return(insertT(t1, update+(pos1, TP)), insertT(t2, pos2))

  if pos1[TP] < pos2[TP]:
    return(insertT(t1, pos1), insertT(t2, update+(pos2,TP)))

  if pos1[TP] == pos2[TP]:
    if len(pos1) > len(pos2):
      return(insertT(t1, update+(pos1, TP)), insertT(t2, pos2))

    if len(pos1) < len(pos2):
      return(insertT(t1, pos1), insertT(t2, update+(pos2, TP)))

    if pos1 == pos2:
      # use application dependent priorities
```

We present the pseudocode for the transformation of an insert$_T$ operation against one insert$_T$ operation as XFORM$_T$ in Listing A.3. Transforming one insert$_T$ operation against another one is similar to the transformation of two insert$_L$ list operations in XFORM$_L$. Particularly, in XFORM$_T$ we perform exact the same transformation at the transformation point as in XFORM$_L$, only the items of the lists are now subtrees. First, we check whether we need a transformation, i.e. if both operations are effect independent as defined in Definition 16. Thereafter, we check, similar to Listing 4.1, how the position parameters at the transformation point are related and transform the position parameter at the transformation point as in XFORM$_L$. If we need to transform one insert$_T$ operation against another one with an identical access path, application dependent priorities are used to prioritize one operation.

We introduce a transformation function for two delete$_T$ operations in Listing A.4. The main difference to the previous transformation of insert$_T$ operations is, that we decrement the position parameters at the transformation point as in XFORM$_L$. Therefore, we use the function update$^-$. If the transformation point of both

Listing A.4: Pseudocode of the transformation of delete_T against delete_T.

```
1  function XFORMT(deleteT(pos1), deleteT(pos2)):
2    TP = TPt(pos1, pos2)
3
4    if effectIndependent(pos1, pos2):
5      return(deleteT(pos1), deleteT(pos2))
6
7    if pos1[TP] > pos2[TP]:
8      return(deleteT(update-(pos1, TP)), deleteT(pos2))
9
10   if pos1[TP] < pos2[TP]:
11     return(deleteT(pos1), deleteT(update-(pos2, TP)))
12
13   if pos1[TP] == pos2[TP]:
14     if len(pos1) > len(pos2): # delete from a deleted tree
15       return(no-op, deleteT(pos2))
16     if len(pos1) < len(pos2): # delete from a deleted tree
17       return(deleteT(pos1), no-op)
18     if pos1 == pos2:
19       return(no-op, no-op)
```

delete_T operations is equal, we either delete a subtree from an already deleted subtree or we have two identical position parameters. Both variants are handled with no-op operations.

After introducing transformation functions for two insert_T or two delete_T operations, we combine both functions to achieve a transformation function for a transformation of insert_T against delete_T. We state the last transformation function in Listing A.5. In the transformation function we modify the access paths exactly as shown in the previous XFORM_L functions. We observe one special case if the access paths of both operations at the transformation point are identical. In this case, the access path of the insert_T operation contains more items than the access path of the delete_T operation. For example, one tree is inserted into a deleted tree (corresponding lines 14-15). To solve this conflict, we use a no-op operation as shown in Listing A.4.

A.3 EXTENDED EVALUATION RESULTS

In contrast to the experiment of Dang and Ignat [DI16], the presented comparison between Google Docs and *formic* in Section 4.6

Listing A.5: Pseudocode of the transformation of insert$_T$ against delete$_T$.

```
function XFORMT(insertT(t, pos1), deleteT(pos2)):
  TP = TPt(pos1, pos2)

  if effectIndependent(pos1, pos2):
    return(insertT(t, pos1), deleteT(pos2))

  if pos1[TP] > pos2[TP]:
    return(insertT(t, update-(pos1, TP)), deleteT(pos2))

  if pos1[TP] < pos2[TP]:
    return(insertT(t, pos1), deleteT(update+(pos2, TP)))

  if pos1[TP] == pos2[TP]:
    if len(pos1) > len(pos2): # insert into deleted tree
      return(no-op, deleteT(pos2))
    else:
      return(insertT(t, pos1), deleteT(update+(pos2, TP)))
```

only shows the result for a collaborative editing session with one keystroke per second. In Figure A.1 to A.4, we visualize the remaining runs with 2-10 keystrokes per second. Furthermore, we reduced the number of series per run to save computing time. Hence, we only simulated the collaborative editing sessions for 1, 2, 5,10,20, and 30 DummyWriters.

Ultimately, we would like to thank Quang-Vinh Dang and Claudia-Lavinia Ignat for publishing their evaluation tool under the terms of a free software license that allows other researchers to use, study, improve, and share the source code. We believe, that this is a crucial step to ensure the reproducibility of scientific results. We are especially thankful for giving us the opportunity to contribute to their research by accepting our pull request.

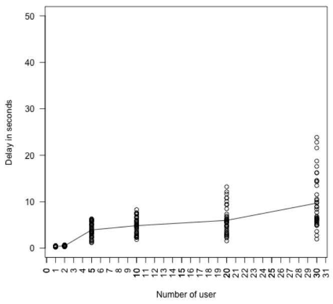


Figure A.1: Performance of *formic* with a typing speed of two keystroke/second.

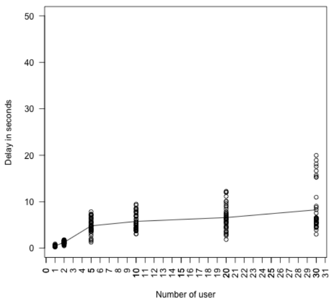


Figure A.2: Performance of *formic* with a typing speed of five keystroke/second.

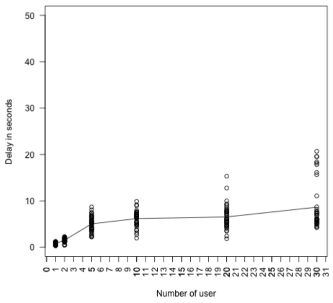


Figure A.3: Performance of *formic* with a typing speed of eight keystroke/second.

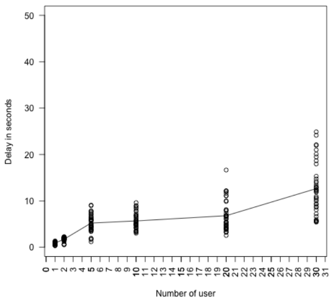


Figure A.4: Performance of *formic* with a typing speed of ten keystroke/second.

Universitätsverlag der TU Berlin

On the Feasibility of Multi-Leader Replication in the Early Tiers

In traditional service architectures that follow the service statelessness principle, the state is primarily held in the data tier. Here, service operators utilize tailored storage solutions to guarantee the required availability; even though failures can occur at any time. This centralized approach to store and process an application's state in the data tier implies that outages of the entire tier cannot be tolerated. An alternative approach, which is in focus of this thesis, is to decentralize the processing of state information and to use more stateful components in the early tiers.
The possibility to tolerate a temporary outage of an entire tier implies that the application's state can be manipulated by the remaining tiers without waiting for approval from the unavailable tier. This setup requires multi-leader replication, where every replica can accept writes and forwards the resulting changes to the other replicas.
This thesis explores the feasibility of using multi-leader replication to store and process state in a decentralized manner across multiple tiers. To this end, two replication mechanisms, namely Conflict-free Replicated Data Types and Operational Transformation, are under particular investigation. We use and extend both mechanisms to demonstrate that the aforementioned decentralization is worth considering when designing a service architecture.

ISBN 978-3-7983-3001-6 (print)
ISBN 978-3-7983-3002-3 (online)

ISBN 978-3-7983-3001-6

http://verlag.tu-berlin.de