Designing a Planetary-Scale IMAP Service with Conflict-free Replicated Data Types

Tim Jungnickel[1], Lennart Oldenburg[2], and Matthias Loibl[3]

1 **Technische Universität Berlin, Germany**
`tim.jungnickel@tu-berlin.de`
2 **Technische Universität Berlin, Germany**
`l.oldenburg@mailbox.tu-berlin.de`
3 **Technische Universität Berlin, Germany**
`matthias.loibl@mailbox.tu-berlin.de`

Abstract

Modern geo-replicated software serving millions of users across the globe faces the consequences of the CAP dilemma, i.e., the inevitable conflicts that arise when multiple nodes accept writes on shared state. The underlying problem is commonly known as fault-tolerant multi-leader replication; actively researched in the distributed systems and database communities. As a more recent theoretical framework, Conflict-free Replicated Data Types (CRDTs) propose a solution to this problem by offering a set of always converging primitives. However, modeling non-trivial system state with CRDT primitives is a challenging and error-prone task. In this work, we propose a solution for a geo-replicated online service with fault-tolerant multi-leader replication based on CRDTs. We chose IMAP as use case due to its prevalence and simplicity. Therefore, we modeled an IMAP-CRDT and verified its correctness with the interactive theorem prover Isabelle/HOL. In order to bridge the gap between theory and practice, we implemented an open-source prototype *pluto* and an IMAP benchmark for write-intensive workloads. We evaluated our prototype against the standard IMAP server *Dovecot* on a multi-continent public cloud. The results expose the limitations of *Dovecot* with respect to response time performance and replication lag. Our prototype was able to leverage its conceptual advantages and outperformed *Dovecot*. We find that our approach is promising when facing the multitude of potential concurrency bugs in development of systems at planetary scale.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Geo-Replication, CRDT, Distributed Systems, IMAP, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.23

1 Introduction

When designing and developing a modern online service, researchers and developers are faced with a large and diverse spectrum of challenges. Due to its ever expanding reach, the Internet offers the unique while demanding prospect of connecting potentially billions of users scattered across the planet. The underlying distributed infrastructure requires enormous consideration on its own, but matters get even more involved when applications on top of it have state. While potentially possible, the penalties of declaring one cluster to be the single leader and routing all requests through it, render such a design unfeasible when reliability and responsiveness are of importance. A geographically-distributed system architecture, however, eliminates single points of failure and enables low response times on client requests.

Unfortunately, though, network and node failures are a given in large-scale infrastructures [2] and thus, our applications have to deal with partitions. To safeguard consistent state

during these, a trade-off between availability and consistency has to be made, commonly known as the CAP dilemma [11]. Choosing consistency requires minority partitions to reject client requests – becoming unavailable – while only a majority partition is allowed to make progress. This has been and still is a viable option, though with the rise of the NoSQL movement, available architectures have gained traction. They choose to always accept client requests even though all other nodes of the system might be temporarily unreachable and try to achieve consistency when connections are re-established. Naturally, in these *eventually consistent* [26] systems, conflicting requests arise. Much research has gone into efficient conflict resolution or even conflict avoidance.

More recently, *Conflict-free Replicated Data Types* (CRDTs) have been proposed as a method for avoiding conflicts [24]. If system state is built either on one of the state-based or operation-based data types, the inherent properties of these ensure that replicated state will always converge. Certain requirements have to be met in order to be able to use them, e.g., do we assume an underlying reliable causal-order network and commuting concurrent updates for the operation-based CRDTs. If we can provide these requirements, CRDTs are an elegant and efficient way to model available and partition-tolerant systems.

In this work, we set out to model, verify, implement, and evaluate a distributed application with non-trivial state based on CRDTs. As the provided service we chose to design an IMAP server. The *Internet Message Access Protocol* (IMAP) is the standard way to manage mailbox state and retrieve messages in an email service. 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. As an example, Gmail recently crossed the mark of one billion monthly active users [19].

Even though the provided CRDT primitives [23] are concise and simple, one can fail in numerous ways when constructing non-trivial system state based on these. We wanted to be sure of the correctness of our model and thus put effort into proving it correct. To this end, we extended the CRDT and network model framework by Gomes et. al. [13, 12] written in the Isabelle/HOL interactive theorem prover to include our IMAP-CRDT. After being assured that state will always be consistent in our model, we adapted our prototype to adhere to the theoretical proof. This way, we achieve provable consistency in practice.

In order to evaluate the benefits of our proposed IMAP-CRDT, we developed a prototype *pluto* and built a federated Kubernetes[1]-based test environment on Google's public container cloud, Google Container Engine[2]. We were primarily interested in two characteristics: *response time performance* and *replication lag* [17]. The first one captures the perceived responsiveness by users while the second describes how long it takes one system to synchronize and apply updates with the other nodes. With our self-developed IMAP benchmark for write-intensive workloads, we were able to gather both insights for *pluto* and *Dovecot*.

We find, that our approach of designing distributed applications at planetary scale yields a straightforward flow of modeling, verifying, implementing, and evaluating software where a proven system model consolidates experimental measurements. The results attest the advantages of our CRDT-based architecture and draw near reproducibility due to our automated and open-sourced deployment infrastructure.

[1] `https://kubernetes.io`
[2] `https://cloud.google.com/container-engine`

Related Work. Large-scale distributed systems replicating application state in an available and partition-tolerant way have received academic attention since the advent of the Internet. Bayou [25] was one of the first distributed storage systems that enabled users to always submit updates and ensured eventual consistency when network connection was available again. Inspired by the fundamental concepts captured in Amazon's Dynamo paper [8], a new class of distributed data stores was proposed and developed, such as Cassandra [18] and Riak[3]. Many of these new developments are also based in parts on the ideas of Google's Bigtable concept [5], which Google itself turned into Spanner [6], its planet-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 [4].

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*, where the update with the biggest timestamp is picked as winner and all others are lost. One well-known merge-based resolution strategy is *operational transformation* [9], though mostly used for collaborative text editing and of decreasing performance with increasing number of operations [1].

Conflict-free Replicated Data Types take a different approach as they avoid conflicts altogether due to their construction properties. Apart from the set of basic data types defined in the original report [23], constructing further CRDTs has been an active research interest, for example, sequence CRDTs such as Treedoc [20] and LSEQ [22], and a composable JSON CRDT [16]. However, integration into production software is only progressing slowly, e.g., as part of Riak or AntidoteDB[4], and we don't see many approaches for standard IT services such as our IMAP-CRDT.

As part of this work, we verify the data type we propose within Isabelle/HOL. Formal verification of CRDTs has been done before, e.g., by Zeller et al. for state-based CRDTs [27]. The framework underlying our formal verification efforts was proposed and implemented by Gomes et al. [12], including a realistic network model and verification of the *Observed-Removed Set* (OR-Set). We base our IMAP-CRDT on the OR-Set and make the extensions to the framework by Gomes et al. with our data type accessible.

Considering state replication for the most widely used IMAP server, *Dovecot*, the *dsync*[5] approach is currently the only application-level support for planet-scale deployments. Unfortunately, *dsync* is limited to pair-wise replication, forfeiting the advantages of a cloud deployment with nodes all over the world. To our knowledge, we propose the first approach to a truly planetary-scale IMAP service that at the same time achieves low response times.

Finally, this work is based on previous efforts by us into the direction of an IMAP service based on CRDTs, presented in an earlier workshop paper [14].

Contributions. Our contributions in this work are as follows:

- We propose an IMAP-CRDT by modeling IMAP commands as operations on a CRDT. (Section 4)
- We verify the convergence of the IMAP-CRDT with Isabelle/HOL. (Section 4.1)
- We propose an open-source prototype *pluto* that offers IMAP at planetary scale with multi-leader replication based on CRDTs. (Section 5.1)

[3] `http://basho.com/products/riak-kv`
[4] `https://syncfree.github.io/antidote/`
[5] `https://wiki.dovecot.org/Replication`

- We introduce a benchmark for IMAP services. (Section 5.2)
- We propose a Kubernetes-based deployment for planet-scale *Dovecot*. (Section 6)
- 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. (Section 7)

2 System Model

In a traditional 3-tier system architecture of an IMAP service, a *proxy* (sometimes called *director*) forwards a client's request to a responsible *backend* node, where the request is processed. The service state in form of all users' mailboxes is typically stored on one or more *storage* nodes, traditionally hosting a strongly-consistent shared file system like NFS or GlusterFS. However, this architecture is no longer feasible in a planetary-scale IMAP system. To illustrate the pitfalls of this approach in a geo-replicated setting, we installed a GlusterFS on two virtual machines in different regions (North America and Europe) of the Google Cloud Platform (GCP). Subsequently, a *Dovecot* in the recommended setting with one proxy and three backend nodes was deployed on our Kubernetes cluster in Europe. Our benchmark revealed, that the response times of write requests exceed the round-trip time between North America and Europe by almost two orders of magnitude. Hence, applying a naive replication over two continents can be much more costly than routing all request through a single location.

The only solution to efficiently apply geo-replication is to relax the consistency requirements and allow backends to progress state without prior synchronization with other regions. This approach introduces the need of conflict management, because concurrent writes from multiple regions may be conflicting. The architecture underlying our proposed system enables a more fault-tolerant and responsive geo-replicated online service. In it, a backend node is authoritatively responsible for a particular range of users and asynchronously connected to backends in all other regions that are responsible for the same range of users. Furthermore, a global storage node can be added to the service that might run on single-tenant, high quality hardware for long-term storage. It can further be used as a failover destination if any backend node fails. From this *distributed system* of backends, we derive our system model which we use in Section 4.1 to reason about the convergence of state.

We obtain an asynchronous network of backend nodes which can be seen as independent processes. The network can suffer from partitions and can recover after a certain time. The backends continue to operate, even if the backend is temporarily disconnected from parts of the network. If a backend crashes, the state of the backend can be recovered. We assume non-byzantine behavior.

3 Technical Preliminaries

In this section we provide a brief introduction to CRDTs and IMAP. Both will be combined in Section 4 when we introduce our IMAP-CRDT.

3.1 Conflict-free Replicated Data Types

The theoretical concept of a Conflict-free Replicated Data Type has been formalized by Shapiro et al. in [24]. 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 application state based on CRDTs are designed to be conflict-free in the first place.

CRDTs come in two variants: *Convergent Replicated Data Types* (CvRDT) and *Commutative Replicated Data Types* (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, the CmRDT. In contrast to state-based ones, replicas in this case exchange operations directly, with minimal state information. A reliable causal-order broadcast ensures that operations ordered by *happened-before* relation 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 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 [23].

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

With commutativity of concurrent updates and 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.

3.2 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 [7]. To reduce the complexity of this paper, we focus on the *consistency-critical* commands, i.e., commands that change the server's state. The commands *create* and *delete* respectively add or remove a mailbox (also called mailbox folder, or simply folder) to or from a user's account. An *append* command is used to add a message to a mailbox folder. With *store*, the message flags, e.g., `Seen`, `Answered`, or `Deleted`, can be altered. The *expunge* command is used to remove all messages with such a `Deleted` flag from a particular mailbox folder. Note, that the commands *store* and *expunge* are only allowed once a particular mailbox folder has been selected with the *select* command.

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.

4 IMAP-CRDT

In our scenario, the main challenge is to model the IMAP commands as operations on a CmRDT. We begin with the decision on the used payload, i.e., the underlying state representation. We identified a map that projects folder names 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 $u : \mathcal{N} \to \mathcal{P}(\texttt{ID}) \times \mathcal{P}(\mathcal{M})$ where $\mathcal{N}$ is the set of

Specification 1 IMAP-CRDT (payload, *create*, and *delete*).

```
1: payload map u : 𝒩 → 𝒫(ID) × 𝒫(ℳ)      ▷ {foldername f ↦ ({tag t}, {msg m}), ...}
2:     initial (λx.(∅, ∅))
3: update create (foldername f)
4:     atSource
5:         let α = unique()
6:     downstream (f, α)
7:         u(f) ↦ (u(f)_1 ∪ {α}, 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) ↦ (u(f)_1 \ R_1, u(f)_2 \ R_2)
```

foldernames, 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.

Because a folder f contains arbitrary items, the result of $u(f)$ is a tuple of two 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. Note, that 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 folder name in $\mathcal{N}$.

We present the complete IMAP-CRDT in Spec. 1 and Spec. 2. We adhere to the presentation style that has been introduced by Shapiro et al. in [24]. Next, we define the operations that represent the IMAP commands and begin with *create* and *delete* in Spec. 1.

The desired result of *create* is to create an empty folder f. Therefore, a fresh and unique tag t is generated on the replica that initiates the operation. This initiation phase of the replica is usually called **atSource**. Thereafter, the tag t is inserted into $u(f)_1$ and $u(f)_2$ remains untouched. This part of the operation is usually called **downstream** and 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. Note, that the map entries for the other folder names remain unchanged.

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 defined 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. Note, that the definitions of *create* and *delete* are very similar to the *add* and *remove* operations on the op-based Observed-Remove-Set (OR-set), which has been introduced in [24].

The remaining operations *append*, *expunge* and *store* are defined in Spec. 2. 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

Specification 2 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 $\alpha = \mathit{unique}()$
23: **downstream** (f, m, α)
24: $u(f) \mapsto (u(f)_1 \cup \{\alpha\}, u(f)_2 \setminus \{m\})$
25: **update** *store* (foldername f, message m_{old}, message m_{new})
26: **atSource** (f, m_{old}, m_{new})
27: **pre** $m_{\text{old}} \in u(f)_2$
28: **pre** m_{new} is globally unique
29: **downstream** $(f, m_{\text{old}}, m_{\text{new}})$
30: $u(f) \mapsto (u(f)_1, (u(f)_2 \setminus \{m_{\text{old}}\}) \cup \{m_{\text{new}}\})$

`UID`. We use this requirement to assure that no two *identical* messages are ever appended by different replicas, or even the same replica. Note, that *identical* is not referring to the message 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 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 a concurrent *delete* operation. 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 state visible at the initiation time of the *delete* operation is removed. Hence, we decided to pursue an *add-wins* semantic.

Design Decisions and Discussion. The proposed *add-wins* strategy comes at 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 currently 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 [3]. 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. Note, that our IMAP-CRDT requires causal-order delivery and we omit this precondition in every update operation for the sake of simplicity.

The commands left to achieve full compliance with RFC 3501 are mostly *read* commands like *search*, *fetch*, or *status*, and thus not in the scope of this work. Missing *write* commands like *copy* and *rename* bear many parts from the already implemented *write* commands but require further careful thought. However, modeling those commands as operations on the IMAP-CRDT is an effortful but realizable task.

4.1 Verification

To ensure that all required properties are satisfied and that convergence among replicas is achieved, we verified our IMAP-CRDT with the interactive theorem prover Isabelle/HOL. We base our Isabelle/HOL implementation on the recently published CRDT verification framework by Gomes et al. [12]. Our formalization follows the definitions we presented in Spec. 1 and Spec. 2 and is available in the *Archive of Formal Proofs* for Isabelle/HOL [15]. The only notable difference in the Isabelle/HOL formalization is, that we no longer distinguish between sets `ID` and $\mathcal{M}$ and that the generated tags of *create* and *expunge* are handled explicitly. This makes the formalization slightly easier, because less type variables are introduced. Ultimately, we show that our IMAP-CRDT achieves *strong eventual consistency* in our proposed system model, i.e., all replicas converge to an identical abstract state when they share the same history of operations.

5 Prototype

5.1 Pluto

Our prototypical implementation of the presented IMAP server system model, *pluto*, and all other software we wrote for this paper, is developed in the Go programming language and available[6] as open-source software under GPLv3 license. The prototype is designed as a distributed IMAP server, internally comprised of multiple *distributor* nodes, multiple *worker* (backend) nodes, and one *storage* node, as set out in Section 2. We put emphasis on application speed, security, reliability, and configurability.

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 partition of users the current user is part of, is determined, and all traffic is proxied to this

[6] `https://github.com/go-pluto/pluto`

node via a gRPC[7] connection. Should the determined worker node be unavailable due to any number of reasons, a *failover* to the global storage node is performed, which accepts the proxied IMAP traffic in place of the worker node.

As soon as requests of a regular IMAP session reach a worker or storage node, they potentially change the mailbox state of the respective user. To achieve availability even in case of failures, worker and storage nodes accept these state-changing requests and guarantee that eventual consistency with the other replicas is reached – an inherent feature of CRDTs. We say that worker and storage nodes are stateful because they first alter their local states and afterwards send messages downstream that apply the same operation on all remote states. This makes *pluto* a multi-leader replication system.

We verified the correctness of our state replication as part of Section 4 and implemented the two required components, the IMAP-CRDT and reliable causal-order broadcast of update messages, as parts of *pluto*. For 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 Spec. 1 is, that the map $u(f)$ for 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 Spec. 1 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. 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 user mailboxes in case they crash.

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, and 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 from 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 [21, 10] 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.

IMAP clients will not notice the replication efforts, as they happen asynchronously. This is one of the major advantages of a geographically distributed *pluto* deployment: requests can be responded to quickly due to authoritative local state on close-by worker and storage nodes while updates will get applied everywhere eventually due to CRDTs and reliable causal-order broadcast, thus achieving consistent state. We guide these claims with structured logging, metrics exposure to Prometheus, and tests for important packages. Further work might go into file checksum checking for ascertaining data integrity, deeper performance profiling, and increased RFC 3501 [7] compliance. We welcome contributions from the community.

[7] `https://grpc.io`

5.2 Benchmark

In order to evaluate *pluto* and *Dovecot* in Section 7, we needed 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 they are received on, without replica communication. Only state-changing commands potentially unearth consistency issues by generating edge cases. Thus, we required 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*.

We could not find such tool or data set available, and thus implemented an IMAP benchmark ourselves[8] that generates arbitrary amounts of random data, 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. Session generation 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 users file and logged in. Next, the session commands are applied one after another, a successive one as soon as the current one has finished and the time between sending it and receiving a complete answer – 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. The results are written to disk and optionally uploaded to a Google Cloud Storage (GCS) bucket.

5.3 Maildir Tools

With the benchmark ready we had almost everything in place required for putting our system model to a test. One more component was needed, though, for gaining insight into the replication performance. 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 visible the asynchronous replication part. Due to different replication mechanisms in *pluto* and *Dovecot*, though, we had to fall back to observing the Maildir file system in order to see when updates were applied.

We implemented a small utility[9] that periodically performs a disk usage calculation of a configured subset of the Maildirs present on a node (by running `'du -s'`). 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. The idea is to integrate one Maildir dumper into each stateful node deployed in a service to be evaluated. After having run an IMAP benchmark, timestamped disk usage reports can be collected and the time difference between the points in time when two observed Maildirs report the same size in bytes can be calculated. We consider this measure the replication lag. Please note, that the calculated time differences have to be taken as estimations rather than precise durations as we rely on synchronized clocks for timestamp elicitation. In Section 7, we will see that the clock synchronization in Google's data centers has negligible influence on our results.

[8] `https://github.com/go-pluto/benchmark`
[9] `https://github.com/go-pluto/maildir_tools`

6 Infrastructure

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

We decided to publish our configurations in our infrastructure repository[10], so that our setup can easily be re-created and re-used for further experiments. Hence, all resources, including the container images of all evaluated systems, are publicly available.

7 Evaluation

To evaluate our approach, we conducted a set of experiments. 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. The results of these experiments we will discuss in the end of this section.

7.1 Baseline Experiment

As introduced in Section 2, we used a *Dovecot* in a traditional 3-tier architecture as reference setup. For the storage layer, 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 (also known as *sharding*). 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 (see Section 5.2) 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. These 20 concurrent users were identified 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 were expected but no replication lag.

[10] `https://github.com/go-pluto/infrastructure`

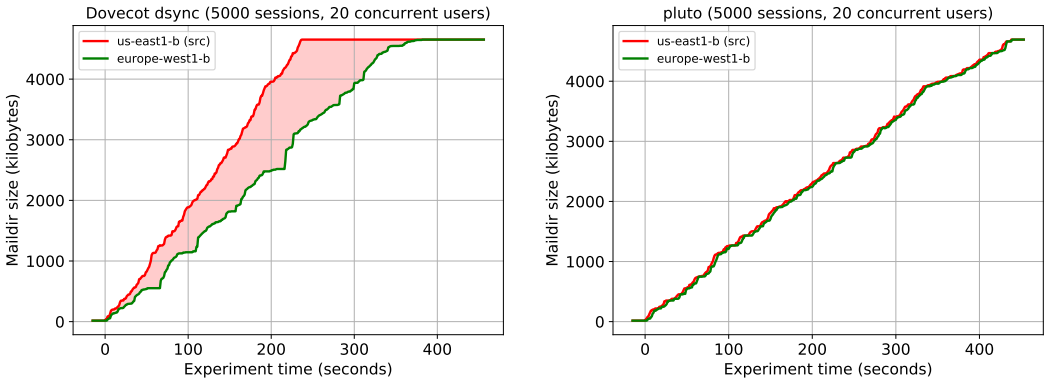

Figure 1 Replication lag diagram for *Dovecot dsync* (left) and our prototype *pluto* (right) for requests from `us` to `europe`.

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.

Results. We show the measured response times in the *baseline* column of Table 1. The average and median response times in milliseconds are grouped by IMAP command. We judge the measured values as realistic for this setup. In fact, our findings in [14] confirm the authenticity of the presented values.

7.2 Experiment 1: Single-Cluster Benchmark

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 experiments, we replayed the settings from our *baseline* experiment, except that 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 application 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 (see Section 5.3). 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.

Results: Dovecot dsync. The measured response times are given in the *dsync* column of Table 1. 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 1, we illustrate the average

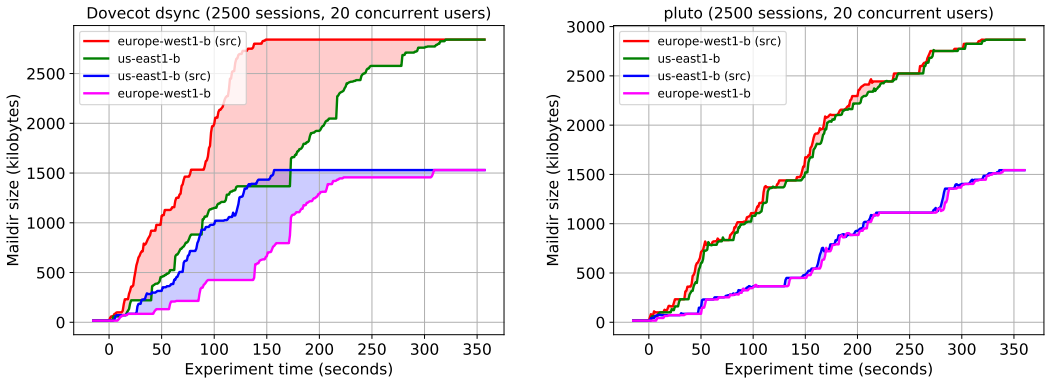

Figure 2 Replication lag for *dsync* (left) and *pluto* (right). The red areas represent the replication from `europe` to `us` while the blue areas represent the opposite direction.

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[11]. 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 1.

Results: pluto. For the *pluto* setup, we additionally deployed the *storage* node (see Section 5.1) 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 tool 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 1. We note that the response times are significantly higher than *Dovecot*'s, which we discuss in the end of this section.

The replication lag diagram is shown in the right part of Figure 1. 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 1.

7.3 Experiment 2: 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 1.

[11] 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.

Table 1 The combined results of Experiment 1 and 2, 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*	*dsync*^{2}		*pluto*^{2}	
						`us`	`eu`	`us`	`eu`
Response Time Performance	`CREATE`	Average	251.36	16.24	47.77	18.47	23.24	47.56	75.20
		Median	224.50	12.52	28.25	14.17	20.33	28.94	29.83
	`DELETE`	Average	602.05	30.81	48.85	32.46	37.03	47.12	74.61
		Median	539.38	27.84	28.30	29.46	34.31	29.16	29.89
	`APPEND`	Average	437.26	43.02	91.96	46.39	55.36	87.23	131.79
		Median	400.87	38.37	57.15	42.08	50.34	55.72	58.66
	`EXPUNGE`	Average	112.91	13.87	42.74	15.59	21.16	40.94	62.72
		Median	97.05	**6.18**	25.61	**9.44**	**18.91**	22.56	23.19
	`STORE`	Average	184.16	15.72	52.04	17.48	21.79	46.09	72.84
		Median	166.66	**11.93**	31.80	**13.83**	**19.64**	29.73	31.53
	Throughput		47.17	480.67	256.03	447.87	367.94	256.26	171.49
Replication Lag		Average		734.61	39.10	592.87	657.76	18.61	44.98
		Median		729.10	**34.44**	217.83	322.10	**6.1**	**34.33**
		Area		279.89	**17.13**	97.92	209.83	**5.83**	**14.32**

In order to measure the replication lag, we also split the sample users and configured our benchmark in a way that the mailboxes 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 receive commands from both regions. We present the replication lag diagram for both systems in Figure 2. 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.

7.4 Discussion

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 relaxation of consistency guarantees and presence of a replication lag.

By comparing the response times of both systems, and in extension to that, the achieved throughput, 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 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, our prototype clearly outperforms *dsync* and we judge our approach as successful. 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.

We would like to point out, that we chose IMAP as the protocol to model 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 allowed for completing work on time. We judge the fact that application 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 application state to model is manageable, our approach is promising. We expect, that with the recently introduced JSON CRDT [16], the modeling of more IT services with CRDTs will become even easier. However, a machine-checked verification of the JSON CRDT is still to be done.

8 Conclusion

The initial exploration of the feasibility of using CRDTs in the multi-leader replication of an IMAP service can be considered successful. We have 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*.

In our work, we consider 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 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 and consider CRDTs for modeling application state.

Future work includes the exploration of CRDTs for other everyday IT services and further improvement of our prototype.

Acknowledgements. We would like to thank the Software Technology Group at TU Kaiserslautern, Georges Younes, Vitor Enes, and the anonymous reviewers for their valuable comments and feedback. Furthermore, we thank the German Research Foundation (DFG) and the graduate school SOAMED for supporting this work.

References

1. Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In *ACM Symposium on Document Engineering*, DocEng'11, pages 103–112, 2011.
2. Peter Bailis and Kyle Kingsbury. The Network is Reliable. *Commun. ACM*, 57(9):48–55, 2014.
3. Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems*, DAIS'14, pages 126–140, 2014.
4. Eric Brewer. Spanner, TrueTime and the CAP Theorem. Technical report, Google, 2017.
5. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 15–15, 2006.
6. James C. Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Google's Globally-distributed Database. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.
7. Mark R. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501, University of Washington, 2003. URL: `https://www.rfc-editor.org/rfc/rfc3501.txt`.
8. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles*, SOSP'07, pages 205–220, 2007.
9. C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Rec.*, 18(2):399–407, 1989.
10. Colin J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*. Australian National University. Department of Computer Science, 1987.
11. Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.
12. Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes. *Archive of Formal Proofs*, 2017. `http://isa-afp.org/entries/CRDT.html`.
13. Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.*, 1(109):1–28, 2017.
14. Tim Jungnickel and Lennart Oldenburg. Pluto: The CRDT-Driven IMAP Server. In *International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC'17, pages 1–5, 2017.
15. Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. The IMAP CmRDT. *Archive of Formal Proofs*, 2017. `http://isa-afp.org/entries/IMAP-CRDT.html`.
16. M. Kleppmann and A. R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.
17. Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., 2017.
18. Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
19. Frederic Lardinois. Gmail Now Has More Than 1B Monthly Active Users. *techcrunch.com*, 2016. [Online; posted February 1, 2016]. URL: `https://tcrn.ch/1nJbAAe`.

20 Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. *Operating Systems Review*, 44(2):29–34, 2010. doi:10.1145/1773912.1773921.

21 Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

22 Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *ACM Symposium on Document Engineering*, DocEng'13, pages 37–46, 2013.

23 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical report, Inria, 2011. URL: https://hal.inria.fr/inria-00555588/file/techreport.pdf.

24 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, 2011.

25 D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *ACM Symposium on Operating Systems Principles*, SOSP'95, pages 172–182, 1995.

26 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.

27 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects*, FORTE'14, pages 33–48, 2014.