pluto: The CRDT-Driven IMAP Server

Tim Jungnickel
TU Berlin, Germany
tim.jungnickel@tu-berlin.de

Lennart Oldenburg
TU Berlin, Germany
l.oldenburg@campus.tu-berlin.de

ABSTRACT

A typical cloud deployment of an IMAP service follows the service-statelessness principle, i.e. a load balancer distributes the requests to a mostly stateless backend that stores application state on a network file system like NFS. Within this work we suggest an alternative architecture allowing the backend to maintain its own replica of application state, leading to shorter response times and increased fault tolerance. To ensure convergence among replicas, we use CRDTs to model application state and evaluate the performance of our prototypical implementation against the de facto standard IMAP server Dovecot in multiple experimentation settings.

CCS CONCEPTS

•**Software and its engineering** →*3-tier architectures;* •**Computer systems organization** →*Cloud computing; Reliability;*

KEYWORDS

IMAP, CRDT, Consistency Control, Cloud Deployments

ACM Reference format:
Tim Jungnickel and Lennart Oldenburg. 2017. pluto: The CRDT-Driven IMAP Server. In *Proceedings of Principles and Practice of Consistency for Distributed Data, Belgrade, Serbia, April 23 2017 (PaPoC'17),* 5 pages.
DOI: 10.1145/3064889.3064891

1 INTRODUCTION

The Internet Message Access Protocol (IMAP) [8] is the standard protocol to retrieve email messages from an email server. Nowadays, we see that IMAP deployments serve millions of active users around the globe. Popular examples include Google's IMAP-based Gmail service, since it has recently been serving over a billion active users per month, and the Dovecot[1] installation of Deutsche Telekom, managing email for over 26 million customers.

To serve a growing number of users in large and unreliable networks like the Internet, in unpredictable and virtualized infrastructures like public clouds, *a certain degree of replication* is necessary, even though new challenges arise. As soon as application state is replicated over several peers, the application faces the CAP dilemma, as outlined by Eric Brewer [5]. Hence, trade-offs between consistency and availability have to be made. In more recent work [6], Brewer argues that an application needs to be aware of the existence of partitions and that it is the designer's challenge to mitigate partitions' effects on consistency and availability.

Brewer explicitly mentions Conflict-Free Replicated Data Types (CRDTs) [11] as "a class of data structures that provably converge after a partition" [6]. Commutative Replicated Data Types (CmRDTs), one part of the CRDT definition, utilize commutativity of update operations, making them prime candidates to accomplish the challenges outlined by Brewer. However, still a lot of work is necessary to close the gap between conceptual use of CRDTs to converge state after recovery from node or network failures and real-world deployments of fundamental IT services[2]. With this work, we contribute to closing this gap by reporting on our experience of using CRDTs to model application state of an IMAP service and delivering the following:

- We present a multitier application architecture for a partition-aware IMAP service deployed on public clouds. (Section 2)
- We model the application's state and an essential subset of IMAP commands with OR-Set CmRDTs. (Section 3)
- We present pluto: an open source prototype that implements the presented concepts. (Section 4)
- We evaluate our prototype against a conventional Dovecot deployment with a focus on response time performance, and discuss major drawbacks of the used consistency model. (Section 4)

2 SYSTEM ARCHITECTURE

In this section, we introduce an alternative system architecture for a deployment of an IMAP service in a public cloud and compare it to a conventional architecture that follows the commonly applied service-statelessness principle [7]. We begin by exemplifying the conventional architecture by an often used Dovecot setup.

A conventional system architecture for a deployment of Dovecot on public cloud infrastructure includes a multitier architecture with:

(1) a *stateful* **storage tier** where the mailboxes of the users are securely stored, nowadays mostly in the *Maildir* format. Stored data is accessed by network file systems, such as NFS or GlusterFS. The provided storage system should be replicated and robust to single disk failure.
(2) several *mostly stateless* **backends** running the IMAP service.
(3) a *mostly stateless* **proxy** redirecting IMAP requests from users to specific backend nodes.

The breakdown into multiple tiers with mostly stateless backends ensures the capability to serve growing amounts of users by scaling out horizontally. This is typically done by allocating additional worker nodes. Such statelessness architectures limit the storing and processing of state information in mostly stateless nodes to the

[1] `https://dovecot.org` — *Secure IMAP Server*

PaPoC'17, Belgrade, Serbia

978-1-4503-4933-8/17/04…$15.00
DOI: 10.1145/3064889.3064891

[2] In his keynote talk at PaPoC 2016 [9], Alan Fekete came to a similar conclusion, raising a *call to arms* in providing more knowledge about the proper use of most weak consistency definitions.

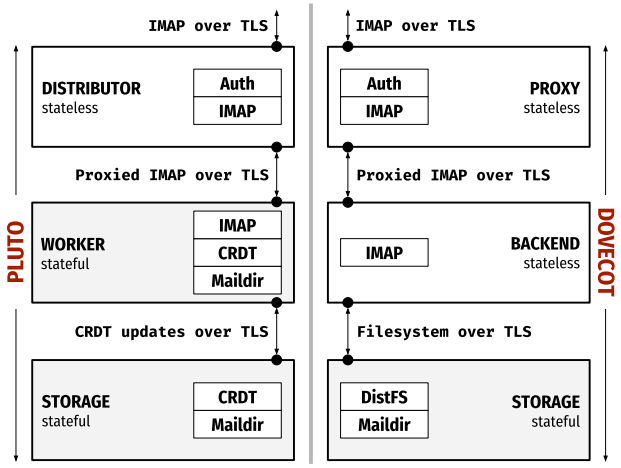

Figure 1: The proposed system architecture of our prototype (left) compared to a conventional system architecture based on Dovecot (right).

necessary minimum. To ensure the service's availability in case of failure or outage[3] of the proxy or backends, new virtual servers can easily be spawned. This *failover* is seamlessly possible since no critical state is held in early tiers. While outages in proxy and backend tier can easily be mitigated by this, a failure or partition affecting the storage tier amounts to stagnation and therefore unavailability.

We visualize the described conventional architecture in the right part of figure 1. Since the presented architecture is tailored for deployment in a public cloud, securing communication channels with TLS is highly recommended. Note that the presented conventional architecture only provides a rather simplified view of a Dovecot system optimized to perfection. We omit important improvements Dovecot has to offer, e.g. caching and index files on backend nodes. Nonetheless, we assume for all following considerations, as suggested by the Dovecot developers, that there is a static mapping between users and backend nodes in order to benefit from caches. Hence, in contrast to other statelessness architectures, one proxy in the Dovecot installation by default forwards requests of the same user always to the same backend node.

As a major contribution of this work, we propose an alternative architecture for an IMAP service that violates the service-statelessness principle. As visualized in the left part of figure 1, we aim to store a copy of the assigned user mailboxes on the disk of each respective worker node and include a CRDT representation of the application's state in worker and storage tier. Each worker thereby has a copy of the fraction of global state that represents its assigned users' data in the system. Incoming IMAP requests are processed by a worker node and authoritatively answered based on its state after a change has been written to the worker's disk. This results in shorter response times compared to conventional architectures since less tiers are required to be contacted in order to answer one request. Each update of the worker's disk triggers an asynchronous update of state in the storage tier by sending updates in the CRDT representation of the application's state *downstream*.

As an unfortunate consequence, state in the storage tier is not always *up-to-date*. In case of an outage of a *stateful* worker node, some state already persistent on its disk might not have yet been transmitted to storage tier, making it temporarily invisible. A *failover* instance continues to serve user requests based on the state in the storage tier, resulting in divergence with the state of the temporarily unavailable worker node. Since all updates between worker and storage are operations in the CRDT representation, we are able to converge application state if the unavailable worker node's state is ever recovered.

With this presented setup, we transfer Brewer's idea of a *Partition Mode* [6] to a multitier architecture of a cloud service, exemplified by an IMAP service. We expect promising implications for response time performance, service reliability, and resource efficiency, which we discuss in section 4. In the following, we introduce the translation of IMAP commands into updates in the CRDT representation.

3 MODELING IMAP WITH CRDTS

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 [8]. To reduce the complexity of this paper, we focus on major *consistency-critical* commands, i.e. commands that change the server's state. We start by giving an intuitive and simplified description of chosen commands and highlight input parameters:

`CREATE`: adds a folder **`$name`** to a mailbox.
`DELETE`: removes the folder **`$name`** from a mailbox.
`APPEND`: adds a message **`$msg`** to the folder **`$name`**.
`STORE`: manipulates meta data (flags) of messages. That means, flags such as `Seen`, `Answered`, `Flagged`, `Deleted`, and `Draft` of a message with identifier **`$mID`** can be changed to a supplied set of new flags **`$flags`**.

To model the state of the mailboxes and corresponding operations, we make use of the Observed-Remove Set (OR-Set) [10] in its operation-based (commutative) version. For our construction of the CRDT representation and IMAP command handlers OR-Set semantics are sufficient. The property that *add* has precedence in case of concurrent *add* and *remove* of the same element, is beneficial as well. We start by defining a message, a folder, and a mailbox structure in the CRDT representation of each user's mailbox.

A **message** is a pair of a file name and a unique tag.
A **folder** is a pair of an OR-Set with messages as payload, denoted as f_{OR}, and a folder name.
A **mailbox structure** is an OR-Set with pairs of folder names and unique tags as payload, denoted as s_{OR}.

Note that we do not use nested OR-Sets in this representation. The mailbox structure s_{OR} and any folder f_{OR} are connected by a common identifier, i.e. the folder name. When access to any folder is desired, s_{OR} is consulted whether a folder with such name exists. If that is the case, a file system access to an object by that name is made. By this, potentially multiple elements with the same folder name in s_{OR} point to the same underlying file system object. We can therefore view all involved OR-Sets as independent and have

[3] One prominent example is the major outage of Amazon Web Services (AWS) in 2011 [3] which led to multiple hours of downtime for sites like Reddit and Foursquare. Coincidentally, during review time of this paper, another major AWS outage took place. It caused multiple hours of disruption of service for a multitude of well-known Internet services [4].

to make sure that logical invariants between these sets are satisfied in the respective commands.

We now model IMAP's CREATE command by using the *lookup* and *add* operations of OR-Set s_{OR}. According to the IMAP RFC, the CREATE command is only successful if the parameter $name does not refer to an already existent folder. Therefore, we add the precondition that a *lookup* in s_{OR} must not be positive. If this precondition holds, we continue to *add* the new folder to the set. Since we follow the OR-Set's specification from [10], we omit a detailed description, assume a correct execution *at source*, and a successful *downstream* part of performed *add* and *remove* operations. Moreover, we assume that execution of the precondition is immediately followed by execution of the routine and that all concurrent accesses to shared data structures are secured by locking mechanisms. The implementation of the DELETE command is analogous.

IMAP Command: CREATE($name)

1: **Precondition**
2: $\neg s_{\text{OR}}.lookup(\$name)$ ▷ check if the folder does not exist
3: **Routine**
4: $s_{\text{OR}}.add(\$name)$ ▷ add the folder to the OR-Set

IMAP Command: DELETE($name)

1: **Precondition**
2: $s_{\text{OR}}.lookup(\$name)$ ▷ check if the folder exists
3: **Routine**
4: $s_{\text{OR}}.remove(\$name)$ ▷ removes the folder from the OR-Set

For the APPEND command, we need to define necessary file system operations to successfully use the OR-Set operations. We give an intuitive description of the defined operations below:

Definition ($\text{fs}_{\text{readOR}}$). The operation $\text{fs}_{\text{readOR}}$ reads the OR-Set for a given folder name from file system.

Definition ($\text{fs}_{\text{writeMsg}}$). The operation $\text{fs}_{\text{writeMsg}}$ writes the contents of a new message to file system. During this process, a file name for the message is generated in accordance to the Maildir specification and subsequently returned. Correctly configured, such file name is guaranteed to be unique in the system.

We model the precondition of APPEND by checking for existence of the folder into which the message has to be inserted. This step is similar to the previous commands. If the precondition holds, we use the $\text{fs}_{\text{readOR}}$ operation to read the corresponding OR-Set of the folder from disk and write the new message to file system using $\text{fs}_{\text{writeMsg}}$. With the obtained file name we use the *add* operation on the folder's OR-Set to insert the message.

IMAP Command: APPEND($msg, $name)

1: **Precondition**
2: $s_{\text{OR}}.lookup(\$name)$ ▷ check if the folder exists
3: **Routine**
4: $f_{\text{OR}} = \text{fs}_{\text{readOR}}(\$name)$ ▷ read OR-Set from file system
5: $\texttt{filename} = \text{fs}_{\text{writeMsg}}(\$msg)$ ▷ write $msg to file system
6: $f_{\text{OR}}.add(\texttt{filename})$ ▷ add filename to OR-Set

For the remaining STORE command, we define the following additional file system operations:

Definition ($\text{fs}_{\text{readMsg}}$). The operation $\text{fs}_{\text{readMsg}}$ returns the file name of the message with given identifier.

Definition ($\text{fs}_{\text{writeFlags}}$). The operation $\text{fs}_{\text{writeFlags}}$ writes new flags of the message with given identifier to file system and returns the new file name.

We model the precondition by using operations $\text{fs}_{\text{readMsg}}$ and *lookup* to check if the message exists. If the precondition holds, we use operation $\text{fs}_{\text{writeFlags}}$ to obtain a new file name for the message with updated flags. The *remove* and *add* operations are executed as a pair, i.e. with no intermediate functions allowed.

IMAP Command: STORE($mID, $flags)

1: **Precondition**
2: $\texttt{filename}_{\text{old}} = \text{fs}_{\text{readMsg}}(\$mID)$ ▷ look up mail identifier
3: $f_{\text{OR}}.lookup(\texttt{filename}_{\text{old}})$ ▷ check if the mail exists
4: **Routine**
5: $\texttt{filename}_{\text{new}} = \text{fs}_{\text{writeFlags}}(\$mID, \$flags)$
6: $(f_{\text{OR}}.remove(\texttt{filename}_{\text{old}}) ; f_{\text{OR}}.add(\texttt{filename}_{\text{new}}))$

Since we modeled the IMAP commands with only *add*, *remove*, and *lookup* operations, we expect worker and storage nodes to converge to same state due to the inherent property of how CRDTs are constructed. We note that the presented model is only a simplified view of a matured implementation. Nevertheless, the integrated operations require to be commutative. We checked this property by simulating all combinations of possibly concurrent IMAP commands and found no discrepancy. A formal and more detailed analysis is still to be made.

4 EVALUATION AND DISCUSSION

To evaluate the conceptual benefits of a CRDT-driven IMAP server, we implemented an open source prototype, called pluto[4], that features a subset of the IMAP commands defined by RFC 3501. The prototype follows the proposed system architecture from section 2 and is written in the Go language. We compare our prototype with the mentioned conventional architecture represented by a deployment of the de facto standard IMAP server, Dovecot. Our measurements focus on response time performance of multiple consecutive IMAP operations. We think it is an important criterion, to decrease idle times on server-side, improve efficiency of deployed software, and provide a comparative benchmark.

4.1 Experimental Setup

We deployed a Dovecot and pluto stack in suggested three-tier architecture on Amazon's EC2. The pluto installation includes three t2.micro (1 vCPU, 1GB RAM, Ubuntu 16.04) instances for a proxy, a worker, and a storage node. All connections between the instances are secured via TLS by default, allowing to deploy such setup over multiple clouds and off-site locations.

[4] https://github.com/numbleroot/pluto licensed under GPLv3 or later

Table 1: Response time per IMAP command in milliseconds.

IMAP		*pluto Ire*	*Dovecot Ire*	*pluto Ire/Lon*	*Dovecot Ire/Lon*	*Gmail*
APPEND	Average	25.11	34.66	24.02	445.96	553.99
	Std. Dev.	13.17	10.07	13.20	70.84	56.58
	Median	22.20	31.84	21.48	439.29	541.52
CREATE	Average	20.22	19.46	19.40	592.22	349.98
	Std. Dev.	13.01	2.37	12.67	30.01	72.72
	Median	16.34	18.83	15.67	601.62	337.15
DELETE	Average	17.99	106.49	18.07	1990.20	361.81
	Std. Dev.	12.16	18.86	12.17	93.65	56.34
	Median	14.60	107.06	14.78	1977.10	352.99
STORE	Average	80.52	19.43	80.26	267.63	126.62
	Std. Dev.	11.61	10.77	8.96	53.95	10.88
	Median	76.81	15.36	76.83	259.49	125.04

The Dovecot installation runs on three t2.micro instances with a Dovecot proxy, a Dovecot backend, and a GlusterFS that is mounted on the backend node. All connections are secured by TLS as well.

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 both deployments. With both installations we conducted five experimental setups:

- *pluto Ire* and *Dovecot Ire*: All instances are deployed on machines in the Ireland region of AWS.
- *pluto Ire/Lon* and *Dovecot Ire/Lon*: The storage and GlusterFS instance operate 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 pluto Ire and Dovecot Ire setups are what we would call the *comfort zone* of both products since there is almost no transmission delay between the instances. In the Ire/Lon setups we have a small but noticeable transmission delay of about 12 ms R/T to the storage tier. These setups are generally interesting since physically separating storage and workers makes a deployment less prone to outages of a single infrastructure provider. In these scenarios a complete outage of the worker nodes, like in mentioned major AWS outages in 2011 and 2017, 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.

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 1.

4.2 Results

We observe that Dovecot delivered a very solid performance in its *comfort zone* setup (Dovecot Ire) with respect to its low standard deviation shown in table 1. Our prototype (pluto Ire) delivered slightly better response times for command `APPEND`, comparable response times for `CREATE`, significantly better response times for `DELETE`, and significantly worse response times for the `STORE` command. The standard deviation of our pluto setup is generally higher than the one of the Dovecot setup. We explain these differences with the high quality of the Dovecot implementation. Dovecot is, after all, the de facto standard IMAP server worldwide. However, with respect to average and median response times, pluto was able to play off the conceptual advantage of storing state in early tiers.

For the Ire/Lon setups, we observe an expected increase in the response time of Dovecot. Since pluto replicates the majority of application state in the early tiers, the response time is almost identical compared to its Ire setup. This experiment demonstrates the major advantage of pluto's system design. 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 outperform pluto 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 run in production.

4.3 Discussion

The conducted experiments focus on the benefits for response time when application state is held in early tiers, in contrast to the widely applied service-statelessness principle. The fact that benefits exist is not very surprising; that our research prototype can play off the conceptual advantage compared to a highly sophisticated competitor used in industry, actually is. The presented results underline Brewer's CAP-latency connection [6] and Abadi's thoughts on the consistency-latency trade-off [1].

In contrast to an architecture pursuing the service-statelessness path, the storage layer in our architecture can become temporarily unavailable without reducing the service's functionality or risking its availability. Having a more complex evaluation of the system's reliability and failover mechanisms is part of future work. Furthermore, it might be interesting to look at the meta data cost incurred by adopting such CRDT-based replication system compared to a standard service-statelessness architecture.

The major drawback of pluto's architecture is that the effect of an IMAP command, which has been displayed to the user but not yet successfully transmitted to storage tier, might temporarily disappear in case of an outage of the worker node. Hence, if a user gets redirected to a failover machine that uses the state present in storage tier, a message that has successfully been appended seconds ago will not be visible. This inconsistent view that only exists if an outage occurs, is the price for lower response times and saved resources[5]. However, since pluto persists all unsynchronized operations to hard disk before sending a response, there is a good

[5]We note that pluto's architecture requires more storage capacity than the presented conventional architecture. Nevertheless, we expect this to be significantly more efficient compared to other approaches for fault-tolerance like active replication or checkpointing [2]. A comprehensive analysis of the resource efficiency is also part of future work.

chance that state can be recovered and integrated without the need for manual conflict resolution.

During the merge of two replicas' diverged states, the involved non-determinism in identifier generation deserves a second look. If, for example, an `APPEND` operation was executed twice in the system due to a partitioned worker and subsequent replacement by a failover node, two different file names for logically the same mail message will have been generated by the Maildir middleware. This may result in duplicate content from which users have to choose which copy to keep. Similar to this is the OR-Set's characteristic "add-wins" property when a logically equal object is concurrently inserted and removed. We generally prefer to have content present twice in contrast to potentially losing the most valuable elements in the service — the users' data.

5 CONCLUSION

Modeling an IMAP service with CRDTs is a manageable task. With the ideal system architecture, we are able to reduce response times and improve service reliability compared to conventional IMAP setups. Our evaluation demonstrates that service-statelessness setups are not taking full advantage of the availability potential. The price to achieve this full potential is to weaken consistency. In the end, it is the designer's choice which properties should be prioritized. We provided a promising alternative that is worth considering, especially for public cloud deployments.

The conducted experiments focus only on response time and ignore the major benefit of a CRDT-driven architecture: *reliability*. Future work includes simulations of failures and outages to gain knowledge of the application behavior in critical situations.

Ultimately, we would like to encourage more system designers to consider the use of optimistic concurrency control in other everyday IT services.

ACKNOWLEDGMENTS

We would like to thank Jana Saalfeld, Juan Cabello, Matthias Loibl, and the three anonymous reviewers for their valuable comments and feedback, which led to significant improvements of this work. Furthermore, we thank the German Research Foundation (DFG) and the graduate school SOAMED for making this work possible.

REFERENCES

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

[2] Navid Aghdaie and Yuval Tamir. 2003. Fast Transparent Failover for Reliable Web Services. In *International Conference on Parallel and Distributed Computing and Systems (PDCS'03)*. 757–762.

[3] Amazon AWS. 2011. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. (2011). https://aws.amazon.com/message/65648/.

[4] Amazon AWS. 2017. Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. (2017). https://aws.amazon.com/message/41926/.

[5] Eric Brewer. 2000. Towards robust Distributed Systems. In *Principles of Distributed Computing (PODC'00)*. (Invited Talk).

[6] Eric Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* 45, 2 (2012), 23–29.

[7] Wojciech Cellary and Sergiusz Strykowski. 2009. e-Government Based on Cloud Computing and Service-oriented Architecture. In *International Conference on Theory and Practice of Electronic Governance (ICEGOV'09)*. 5–10.

[8] Mark Crispin. 2003. *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. RFC 3501. https://tools.ietf.org/html/rfc3501

[9] Alan Fekete. 2016. Consumer-view of consistency properties: definition, measurement, and exploitation. In *Principles and Practice of Consistency for Distributed Data (PaPoC'16)*. (Keynote Talk).

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report. INRIA.

[11] 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'11)*. 386–400.