HAL open science

# CassMail: A Scalable, Highly-Available, and Rapidly-Prototyped E-Mail Service

Lazaros Koromilas, Kostas Magoutis

▶ To cite this version:

Lazaros Koromilas, Kostas Magoutis. CassMail: A Scalable, Highly-Available, and Rapidly-Prototyped E-Mail Service. 11th Distributed Applications and Interoperable Systems (DAIS), Jun 2011, Reykjavik, Iceland. pp.278-291, 10.1007/978-3-642-21387-8_23 . hal-01583577

HAL Id: hal-01583577

https://inria.hal.science/hal-01583577

Submitted on 7 Sep 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

# CassMail: A Scalable, Highly-Available, and Rapidly-Prototyped E-Mail Service

Lazaros Koromilas and Kostas Magoutis

Institute of Computer Science (ICS)
Foundation for Research and Technology Hellas (FORTH)
Heraklion, GR-70013, Greece
{koromil,magoutis}@ics.forth.gr

**Abstract.** In this paper we present the design and implementation of a scalable e-mail service over the Cassandra eventually-consistent storage system. Our system provides a working implementation of the SMTP and POP3 protocols and our evaluation shows that the system exhibits scalable performance, high availability, and is easily manageable under write-intensive e-mail workloads. The design and implementation of our system is centered around a synthesis of interoperable components for rapid prototyping and deployment. Besides offering a proof of concept of such an approach to prototyping distributed applications, we further make two key contributions in this paper: First, we provide a detailed evaluation of the configuration and tuning of the underlying storage engine necessary to achieve scalable application performance. Second, we show that the availability of scalable storage systems such as Cassandra simplifies the design and implementation of higher-level scalable services, especially when compared to the effort expended in projects with similar goals in the past (e.g., Porcupine). We believe that the existence of infrastructural services such as Cassandra brings us closer to the vision of a universal toolbox for rapidly prototyping arbitrary scalable services.

## 1 Introduction

The large-scale adoption of the Internet as a means of personal communication, societal interaction, and a successful venue for conducting business has raised the need for applications and services that can support Internet-scale communities of users. Cloud computing [1] has recently offered the infrastructural support necessary for small, medium and large enterprises alike to deploy such services. However both application developers and Cloud computing providers are in need of the infrastructural services and platforms that can support the scalability requirements of distributed applications. Compounded with the need for scalability is the need for rapid prototyping. Today's planet-scale social networks such as Facebook, and application marketplaces such as AppleStore have brought applications (and the "garage innovator" behind them [5]) closer to large communities of users. This trend has fueled a race among developers to bring new ideas to market as soon as possible without sacrificing scalability and availability along

the way. The combination of the above needs (scalability and rapid prototyping of application-specific designs) was certainly a challenging undertaking a decade ago. As the core theme of this paper indicates however, recent advances in scalable infrastructure services have improved on the state of the art.

In this paper we substantiate the above observations by focusing on a specific case study: scalable e-mail services. E-mail is an important application offered as an Internet-accessible service by companies such as Google (Gmail), Yahoo (Yahoo! Mail), and Microsoft (Hotmail) among others. Constructing a scalable, highly-available e-mail service has in the past been performed in a variety of ways, by either statically partitioning users and their data in specific machines [3] using a general-purpose distributed file system as an underlying scalable store [6,14] or by specifically designing and constructing an entire system from first principles for the targeted application [12]. Whereas the first approach results in simpler systems, experience shows that it suffers from scalability issues specifically in the areas of load balancing (due to static partitioning) and availability (due to strong consistency built into general purpose file systems). The latter approach has in the past been shown to address the above issues, however at the cost of significant system engineering to support (i) fine-grain, balanced data partitioning and (ii) a weaker consistency model that matches the semantics of e-mail protocols. The system presented in this paper combines for the first time the best of both approaches: A synthesis of interoperable components resulting in a simpler system, capitalizing on standardized support for (i) and (ii) above.

A key motivation for this paper is the observation that in recent years there has been significant interest in developing, and in many cases open-sourcing, scalable infrastructural services. This activity has culminated into systems that offer storage/file and data APIs with strong [2, 6, 13] as well as weak [8, 9, 15] consistency semantics. A prime example of the latter class of storage systems that is used in this paper is Apache Cassandra [9], a scalable, highly-available, eventually-consistent key-value store originally developed by Facebook [9]. The existence of Cassandra prompted us to revisit the question of how would one design and build a scalable e-mail service over an eventually-consistent replicated storage system today. More specifically, we considered the following questions:

1. Does Cassandra significantly reduce the development effort compared to the effort expended in a project with similar goals in the past [12]?

2. Does the resulting system exhibit the scalability and availability expected of a robust scalable e-mail service?

More generally, our work puts novel infrastructural services such as Cassandra into context with past efforts to explore the feasibility and utility of providing high-level abstractions or data structures as the fundamental storage infrastructure. The Boxwood project [10] and the scalable distributed data structure (SDDS) approach of Gribble et al. [7] are two examples of such efforts. We believe that a key missing piece in past proposals are primitives that explore

the space of data consistency semantics. This paper shows that Cassandra is such a missing piece that brings us closer to the vision of a toolbox of universal abstractions to support arbitrary scalable application development.

Our contributions in this paper are:

- The design and implementation of a fully functioning scalable, highly-available e-mail service based on synthesis of interoperable components (extensible high-level development interfaces, Cassandra storage system).

- Demonstration of the speed of prototyping that such a software engineering approach allows. Specifically, it took us a few tens of lines of Python code to implement a working prototype compared to the 41,000 lines of C++ code it took for a system with similar design principles a decade ago [12].

- Evaluation of the configuration and tuning of the underlying storage engine for the targeted application, exhibiting the scalabilibity, availability, and manageability properties of our rapidly-prototyped system.

The rest of the paper is organised as follows. We refer to related work in Section 2. In Section 3 we describe the system architecture and in Section 4 we provide the details of our implementation. In Section 5 we describe the evaluation of our system. In Section 6 we discuss possible optimizations and finally, in Section 7 we conclude.

## 2 Related work

The penetration of e-mail in our way of daily life is such that it is nowadays a mandatory offering by all Internet-scale service providers to their subscribers [4]. Large-scale e-mail services have in the past been implemented in a number of ways. Early distributed e-mail services partitioned e-mail state across a number of storage nodes but did so in a static partitioning scheme using distributed file systems [3]. Such a scheme is hard to manage (in particular, it is hard to rebalance e-mail to storage node in case of failure, or to correct an initially unbalanced partitioning). Cluster-based e-mail services [16] attempted to achieve scalability and availability via database failover schemes with limited success. Finally, application-specific designs [12] achieved better scalability via the use of hash-based partitioning of user e-mail to storage nodes, optimistic replication [15], and dynamic membership algorithms. Such approaches however were complicated and thus were met with little practical success in term of real-world deployment and use.

CassMail shares the basic premise behind systems such as Porcupine [12], namely that the semantics of e-mail protocols are naturally relaxed and users are used to e-mail being occasionally delayed, reordered, reappearing after being deleted, or even lost. Thus it is based on a storage system (Cassandra [9]) utilizing optimistic replication to achieve scalability and high availability. Cassandra is

an eventually-consistent storage system, meaning that replicas can temporarily diverge (and clients allowed to see inconsistent intermediate state) but guaranteed to eventually converge. Based on a general-purpose eventually-consistent key-value store rather than its own implementation, CassMail leverages a robust and tested scalable software component and at the same time radically simplifies the overall architecture focusing on the core logic of the application.

The idea of using foundational storage abstractions to support application-specific services is not new. Boxwood [10] explored the idea in which high-level abstractions can facilitate the creation of scalable and fault-tolerant applications. Boxwood proposed a set of components (replicated logical devices, a chunk data store, and a scalable B-tree, along with a set of infrastructure services such as global state management, lock service, and a transactional service) as a comprehensive toolbox for distributed system design. At a smaller scale, scalable distributed data structures [7] proposed a key-value hash table as another foundational abstraction for the support of scalable applications. A common theme in the above proposals was the assumption of strong consistency semantics (single-copy serializability [11]), which in some cases limit system availability and may be constraining applications, especially when their semantics do not strictly require it. Components such as Cassandra extend and enrich the above proposals.

## 3 Design

We will first give a brief overview of the Cassandra data model and the schema that we designed for CassMail. Cassandra's basic data unit is the *column*, or *block* as we will refer to it next. A block consists of a key and a value. Sequences of blocks (an arbitrary number) collectively form a *row*. Blocks in a row can be ordered in a user-specified manner depending on key type (for example in timestamp order). Each row is identified by a separate key. A row is individually placed on a Cassandra storage node based on a consistent hashing scheme [9] described later in this section. Rows are grouped into *column families*, which are entities akin to relational database tables. Column families are grouped into *keyspaces*.

Figure 1 displays the schema in which Cassandra stores user and mailbox information. There are two tables, Mailboxes and Users, within a keyspace called Mail. Users is used to validate a user (the origin or destination of an e-mail message) and to find the names of his mailboxes. Each row in Users is keyed by user name. The blocks stored in a row have as their keys the names of the user's mailboxes. Concatenating a username with a mailbox name forms a natural key to a row in the Mailboxes table. The row contains blocks that hold the actual e-mail messages in that user's mailbox. The key for each block is a time-based universally unique identifier (UUID) stamped by the SMTP daemon when the message arrives and is stored. The value of a block is the e-mail message itself. We chose not to fragment a user's mailbox across multiple rows (as for example Porcupine does [12]) for two reasons: First, we believe that spreading the load of retrieving a mailbox to several nodes can be achieved by reading different block

ranges from different replicas of the row rather than different fragments of the mailbox. Second, we want to avoid hard-to-adjust magic constants (such as the soft limit used by Porcupine [12]) to restrict the mailbox spreading too far across storage nodes.

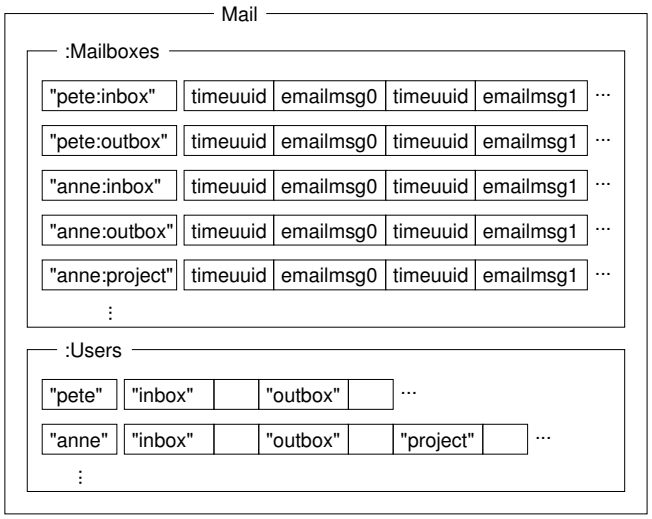


**Fig. 1.** The Cassandra schema designed for CassMail.

Cassandra runs on a cluster of $n$ storage nodes as shown in Figure 2. Each node maps to a specific position on the ring through a hash function. Similarly, each row maps to a position on the ring by hashing its key using the same hash function. Each node is in charge of storing all rows whose keys hash between this node's position and the position of the previous node on the ring. Cassandra members communicate with each other to exchange membership and liveness information through RPC calls. They also communicate when looking for the node in charge of the client's requested data. In our design each Cassandra node is also running an SMTP and a POP3 server. In this fashion, the cluster consists of functionally identical nodes that can perform any and all functions. This symmetrical configuration underlies the system's scalability and availability properties.

An example of e-mail delivery and retrieval is depicted in Figure 2. In this example anne connects[1] through her mail-submission agent (MSA) to the SMTP server on node 1 to send an e-mail message to pete. The SMTP server inspects the message and saves it to pete's inbox and anne's outbox on Cassandra. Now suppose that pete wants to check his messages by accessing node 2. He connects

[1] Normally an e-mail goes through the submission, relaying, and delivery steps. Without loss of generality in this discussion we omit relaying and think of the users connecting directly to a system node to submit and then deliver the message.

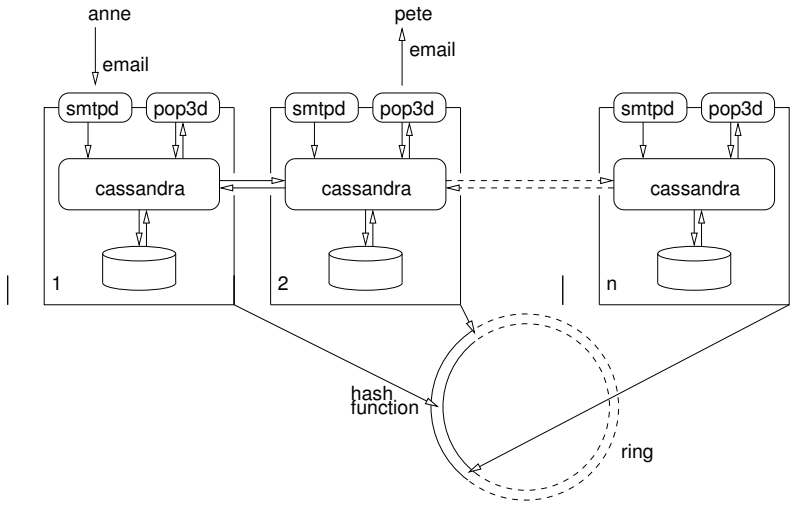


**Fig. 2.** System design

to the POP3 server on node 2 with his mail-retrieval agent (MRA) which first receives the number of e-mails (equal to the number of columns in the pete:*mailbox* row) then asks for message headers, and finally retrieves a number of e-mail messages. POP3 fetches pete's mailbox from Cassandra in fixed-size batches. Each batch would normally go to a different replica for the row to ensure that the read load is balanced across the Cassandra cluster. Eventually pete receives the new message from anne.

## 4 Implementation

In this section we describe our implementations of the SMTP and POP3 servers and the configuration and tuning of the Cassandra system to support our write-intensive e-mail workloads. The set of commands implemented by the SMTP and POP3 servers is shown in Table 1. This is the minimum set that enables Mail User Agents (MUAs) to properly receive and send e-mail. We initially implemented the SMTP and POP3 servers in Ruby using the generic `GServer` class[2]. We found that `Gserver` handles many low-level management tasks, allowing the developer to focus on the specifics of the SMTP and POP3 protocols. Additionally, the Ruby client for Cassandra[3] provides a clean, high-level interface that is easy to work with. However, our preliminary experiments showed that the underlying implementation of `Gserver` was not robust enough to successfully pass our stress tests. Given this early experience we decided to switch to using Python's `smtpd` module and the `pycassa`[4] client library for Cassandra which proved to be a

[2] `http://www.ruby-doc.org/stdlib/libdoc/gserver/rdoc/`
[3] `https://github.com/fauna/cassandra`
[4] `https://github.com/pycassa/pycassa`

| Server | Supported commands |
|---|---|
| SMTP | HELO MAIL RCPT DATA RSET NOOP QUIT |
| POP3 | USER PASS STAT LIST UIDL TOP RETR DELE RSET NOOP QUIT |

**Table 1.** Protocol commands supported by SMTP and POP3 servers.

more robust and performant implementation. Our working implementation of the SMTP and POP3 servers consists of a few tens of lines of code that is easy to reason about and extend.

In addition to implementing the POP3 protocol, we extended the implementation to support multiple mailboxes per user (a feature not directly supported by POP3). We achieved this by appending a delimiter to the username, followed by the specific mailbox name, as for example in `POPUSER="anne|outbox"`. Upon receiving such a name the POP3 server extracts the username and mailbox (using `inbox` as the default) and uses it to interact with Cassandra.

SMTP and POP3 servers access Cassandra through the Thrift[5] interface. Thrift transparently handles Cassandra node availability issues such as failing over to another Cassandra node when the current one appears to have failed. In our implementation we collocate SMTP, POP3, and Cassandra servers on each system node (thus exposing Thrift on the localhost interface). This was a design choice we took to arrive at a homogeneous system in which any node can perform any task. However it would be a trivial modification to enable SMTP/POP3 servers to access Cassandra over the network thus decoupling them into two separate tiers.

Properly configuring Cassandra is key for tuning the system towards specific workloads and environments. Our schema described in Section 3 can be embodied within a very concise description that comprises the name of the keyspace, the column families included, and the information of how to sort the blocks inside them, as shown in the following configuration excerpt:

```
keyspaces:
    - name: Mail
      replication_factor: 3
      column_families:
        - name: Mailboxes
          compare_with: TimeUUIDType
        - name: Users
          compare_with: BytesType
```

Another configuration decision is how to partition the logical ring between Cassandra nodes. Our experience with automatic/random node placement on the ring is that it can lead to hot-spots. We thus opted for precomputing initial tokens for our nodes ensuring that each node gets roughly equal key ranges. We

[5] `http://incubator.apache.org/thrift/`

used Cassandra's *RandomPartitioner*[6] to achieve balanced distibution of row keys on the ring by hashing them using the MD-5 hash function.

In case of a node failure, surviving nodes detect the failed node and exclude them from their membership list. Cassandra does not automatically repartition the ring to the surviving nodes. This is something that requires manual intervention with the execution of shell commands by an operator. However the system (depending on settings described below) can be available while operating under failure as our evaluation in Section 5 shows.

Two other key Cassandra parameters are the degree of replication (or *replication factor*) for row data and the level of consistency chosen for reads and writes. Describing the full set of options offered by Cassandra is outside the scope of this paper (see [8,9] for a complete discussion). We will however describe the options we exercised during our evaluation to highlight the key tradeoffs involved. In terms of consistency levels, we used the following conditions for acknowledging a write:

**ONE** Write must be stored at the memory table and commit log of at least one replica.

**ALL** Write must be stored at the memory tables and commit logs of all replicas.

Our implementation uses Cassandra[7] version 0.7.0 running on Java 1.6.0_22. We used Ruby version 1.9.2 and Python version 2.6.6.

## 5 Evaluation

In this section we report on our experimental results. Our experimental setup consists of a 10-node cluster of dual-CPU AMD 244 Opteron servers with 2GB DRAM running Linux 2.6.18 and connected through a 1Gbps Ethernet switch using Jumbo (9000-byte) frames. Each node was provisioned with a dedicated logical volume comprising four 80GB SATA disks in a RAID-0 configuration. We used the xfs filesystem on this volume on all nodes. The benchmark used in this study was Postal[8], a widely-used benchmark for SMTP servers. Postal operates by repeatedly and randomly selecting and e-mailing a user (`USER@example.com`) from a population of users. We created a realistic population of users by using the usernames from our departmental mail server (about 700 users). The Postal client is a multithreaded process that connects to a specific SMTP server. In our experiments we configured Cassandra with different replication factors (1, 2, 3) and consistency-levels (ONE and ALL described in Section 4). We used message sizes drawn uniformly at random from a range of sizes. We experimented with two ranges:

- 200KB–2MB (typical of large attachments)
- 50KB–500KB (typical of small attachments)

[6] `http://wiki.apache.org/cassandra/StorageConfiguration`
[7] `http://cassandra.apache.org/`
[8] `http://doc.coker.com.au/projects/postal/`

These ranges are chosen to reflect the increase in average e-mail size compared to a related study performed a decade ago [12] due to widespread use of attachments in everyday communications.

Each experiment consists of an e-mail-sending session blasting the CassMail cluster to saturation for about ten minutes. Our measurements are per-minute Postal reports of the sum of e-mail data sent during the previous minute (that is e-mail payload, excluding other control/header information). In order to avoid taking into account any bootstrapping overhead we only consider the last five minutes in our measurements. In all of our graphs we report aggregate average throughput and standard deviation (as error bars) of our measurements. Each node ran an instance of the SMTP server (Python code) and an instance of the Cassandra server (Java code), with each server consuming one of the two CPUs. In all cases performance is limited by the servers' CPUs. There was also swapping and garbage collection activity taking place during runs. We consider such activities unavoidable (especially when running software in high-level, scripted, and garbage collected languages such as Python and Java) and legitimate part of a node's load. We used two dedicated client machines with similar specifications to our servers to drive all experiments. The client machines hosted Postal processes in a setup that balanced load-generation work across the two machines.

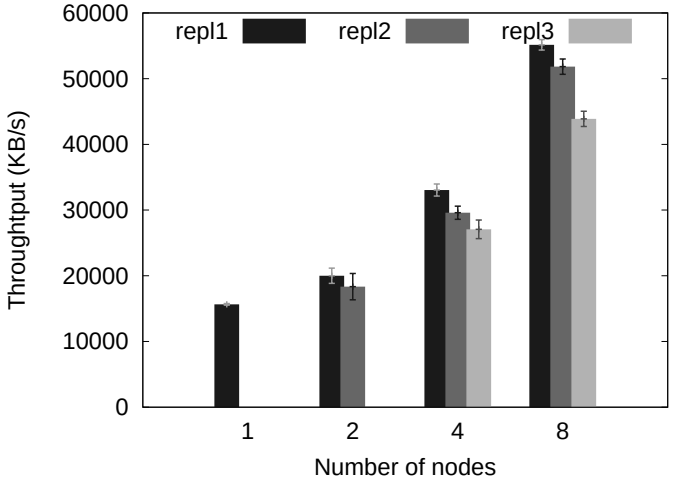


**Fig. 3.** Throughput for different replication factors.

Our first experiment measures the aggregate write throughput over increasing cluster sizes for messages in the range 200KB–2MB. Our results are depicted in Figure 3. Lighter bars correspond to higher replication factors (1–3). The consistency level is set to ONE in all cases. Increasing cluster size results into higher aggregate throughput across all replication factors. The performance increase is smaller going from 1 to 2 nodes due to the introduction of Cassandra server-to-server traffic to forward keys to the proper coordinator (since the e-mail clients

are unaware of the mapping between user mailboxes and Cassandra nodes). Increasing the replication factor results into decreased throughput by about 5–10% for each extra replica at all cluster sizes due to the additional traffic necessary to update replicas. We expect this drop to be steeper for stricter consistency levels such as ALL. Note that a replication factor of 2 and 3 does not make sense for cluster sizes of 1 and 2 respectively, explaining the missing bars in Figure 3.

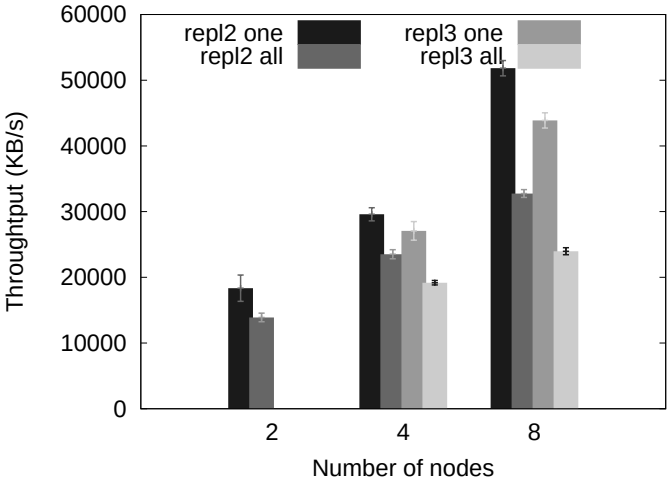


**Fig. 4.** Throughput for different consistency levels.

Figure 4 depicts the impact of the consistency level in aggregate write throughput with increasing cluster sizes (2–8) and replication factors (2–3). The key observation is that stronger consistency requirements (ALL instead of ONE) degrade performance in all cases. The degradation is more pronounced at larger cluster sizes and is about 35% in the case of 2 replicas (dropping from 50MB/s to 30MB/s) and about 45–50% in the case of 3 replicas. A key factor responsible for this degradation is the large imbalances in nodes' performance due to background tasks such as Java garbage collection or swapping activity. These imbalances are largely masked at consistency level ONE but exposed to the clients at consistency level ALL. This observation highlights a key advantage of eventually-consistent storage systems compared to strongly-consistent ones under write-intensive workloads. Previous work [7] has pointed out the adverse impact of garbage collection activity in strongly-consistent storage systems written in Java, namely stalling write operations when one out of a group of replicas freezes while undergoing some background activity. Eventually-consistent systems can hide that stall time by allowing operations to progress at the speed of the fastest replica.

We next focus on the impact of message size on aggregate throughput. Figure 5 depicts system performance with increasing cluster size at consistency level ONE and replication factor 1. We observe a performance drop when mov-

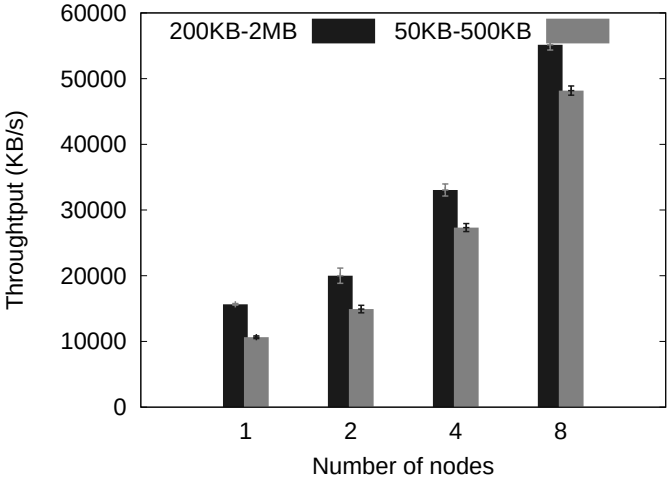


**Fig. 5.** Throughput for different message size ranges.

ing from larger (200KB–2MB) to smaller (50KB–500KB) e-mail messages, which is about constant in absolute value (≈5MB/s) but decreases in relative terms with increasing cluster size. This drop is caused by the higher impact of per-operation overheads (connection setup/teardown, header information generated and processed for each e-mail message, etc.).

We next explore the availability of CassMail service when experiencing a node failure under replication factor 2 at different consistency levels (ONE, ALL). Figure 6 depicts aggregate per-minute throughput before ($0'$–$3'$) and after ($4'$–$10'$) failure for consistency level ONE (squares) and ALL (dots). For this experiment we used an 8-node CassMail cluster in which all nodes run Cassandra servers and only four out of them also run SMTP servers. At minute 3 we inject a crash failure on one of the Cassandra-only nodes. In the case of relaxed consistency (ONE, squares) the node failure has no apparent effect on performance since the surviving replica takes the update (and thus all writes completing successfully without delay) while failover mechanisms (such as hinted handoff [8, 9]) are initiated in the background. In the case of consistency-level ALL (dots) we observe a measurable degradation of about 25% in the following minute and immediate recovery of service after that. This happens because some writes cannot get acknowledgments from the failed node and thus temporarily block until the failover mechanism has been activated. In all cases, CassMail can rely on Cassandra to gracefully handle the node failure with minimal or no availability loss, and without operator intervention.

Comparing CassMail experimentally to systems with equivalent functionality is hard since to the best of our knowledge no such systems are available in open source. The closest alternative —lacking several of CassMail's properties— would be a system relying on static partitioning of users over conventional SMTP servers. For the purpose of comparing to such a system we configured its build-

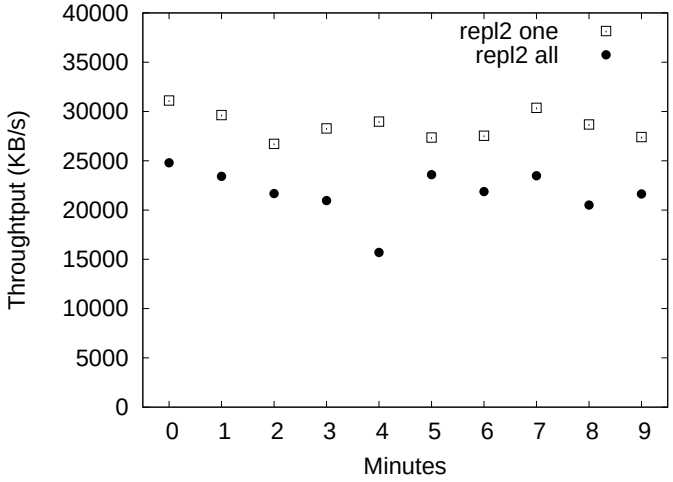


**Fig. 6.** Throughput over time before and after the occurrence of a failure event.

ing block (an SMTP server based on Postfix 2.7.1) and used it as a reference point for comparing CassMail's single-node performance with a mature tighly-configured software system. We did not focus on larger-scale experiments since with static partitioning, client awareness of data location, and no replication one can trivially achieve linear scalability up to the limits of the network. We used the Postal benchmark configured as described earlier and created a mailbox file for each user in the server's file system. Our results show that the Postfix-based server achieves average write throughput of 40MB/s and 25MB/s with large (200KB–2MB) and small (50KB–500KB) messages respectively, limited by CPU in both cases. This contrasts to CassMail's single-node write performance of 15MB/s and 10MB/s for large and small messages respectively. The performance difference can be attributed to implementation characteristics: CassMail is written in high-level programming languages and libraries (Python, Java) and combines e-mail protocol processing with significant storage system processing at each server node. Postfix on the other hand, is a mature performance-optimized software system written in C using a lightweight storage stack. We believe that CassMail's scalability properties can make up for the impact in single-node performance.

## 6 Discussion and future work

We are exploring deployment of CassMail in a Cloud infrastructure [1, 5] offering virtual machines (VMs) and local or remotely-mounted storage volumes. In a straightforward deployment scheme each Cassandra server maps to a VM and each disk to (possibly RAID setups of) local or remotely-mounted storage volumes. Assumptions about failure independence require VMs and storage volumes to not share any single point of failure (such as a physical server). Current

Cloud providers hide this level of information from the user raising a challenge to effective deployment. Our experimental results suggest the use of VMs with considerable CPU (number of cores) and physical memory allocations. In addition, the higher performance, reliability, and predictability of local storage makes it a better alternative to remotely-mounted storage for storing Cassandra data.

Our system has proven to be quite robust under intensive experiments but is currently lacking some features that are needed for real-world deployment. First, it does not deal with user authentication or data encryption of the messages being transferred. Also, the SMTP server currently receives e-mail but does not relay messages to other mailservers. We believe that these features are straightforward additions to our prototype and we plan to implement them in the near future.

## 7 Conclusions

Eventually-consistent storage systems have been shown to be key enablers of scalable, highly available, and manageable application services that do not require strict consistency semantics. In particular, Porcupine [12] demonstrated a scalable e-mail service that was based on an eventually-consistent storage system built from scratch. A drawback of such an approach is the complexity and long development effort it requires (Porcupine consists of 14 major components written in C++ with a total of about 41,000 lines of code [12]). The emergence of general-purpose scalable storage services that offer APIs with eventual-consistency semantics such as Cassandra raise the opportunity of realizing application services similar to Porcupine at a lower development cost.

In this paper we describe the results of a project to build an SMTP/POP service over Cassandra and show that such a service can be simple (consisting of a few tens of lines of Python code focused on the application logic) and thus rapidly implemented. We also show that the implementation exhibits good scalability properties: its throughput increases from 15MB/s to 55MB/s when the cluster size grows from 1 to 8 nodes and a crash failure of a single node results in minimal to no availability lapse, depending on the level of consistency. These properties also indicate an easy-to-manage system (no need for human intervention in mapping users to storage nodes or for restoring availability at the time of failure), which is a critical characteristic of a system meant to operate at a large scale.

A cost for the simplicity of our design is the additional overhead (evidenced by high CPU usage of Python SMTP/POP server and Java Cassandra client/servers) as well as the background activity inherent in scripted and garbage-collected programming environments. However, the combination of technology trends pointing to more cycles in future multi-core CPUs (which can to some extent absorb higher overheads of high-level language runtimes) and the strength of the eventually-consistent storage model in hiding the effect of slow replicas in write performance, paint a positive conclusion: We believe that the synthesis of interoperable (application and storage) components is a viable path to rapidly prototyping robust scalable systems. In this context, storage systems with general-

purpose APIs that explore alternative consistency semantics are important foundational abstractions for building scalable applications.

## 8 Acknowledgments

We thankfully acknowledge the support of the European ICT-FP7 program through the SCALEWORKS (MC IEF 237677) and CUMULONIMBO (STREP 257993) projects.

## References

1. M. Armbrust et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, UC, Berkeley, Feb 2009.
2. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
3. N. Christenson, T. Bosserman, and D. Beckemeyer. A Highly Scalable Electronic Mail Service using Open Systems. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997.
4. N. Ducheneaut and V. Bellotti. E-mail as habitat: an exploration of embedded personal information management. *ACM interactions*, 8(5):30–38, 2001.
5. J. Elson and J. Howell. Handling Flash Crowds from your Garage. In *USENIX 2008 Annual Technical Conference*, Boston, MA, 2008.
6. S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.
7. S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of 4th Conference on Operating System Design & Implementation*, San Diego, CA, 2000.
8. D. Hastorun et al. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of Symposium on Operating Systems Principles*, Stevenson, WA, 2007.
9. A. Lakshman and P. Malik. Cassandra: a Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
10. J. McCormick, N. Murphy, M. Najork, C.A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of Conference on Operating Systems Design & Implementation*, San Francisco, CA, 2004.
11. C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.
12. Y. Saito, B.N. Bershad, and H.M. Levy. Manageability, Availability, and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service. *ACM Transactions on Computer Systems (TOCS)*, 18(3):298, 2000.
13. K. Shvachko et al. The Hadoop Distributed File System. In *Proc. of IEEE Conf. on Mass Storage Systems and Technologies*, Lake Tahoe, NV, 2010.
14. C. Thekkath, T. Mann, and E. Lee. Frangipani: a Scalable Distributed File System. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, 1997.
15. W. Vogels. Eventually Consistent. *ACM Queue Magazine*, December 2008.
16. W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo. Scalability of the Microsoft Cluster Service. In *Proc. of 2nd USENIX Windows NT Symposium*, Seattle, WA, 1998.