A Testing Framework for Cloud Storage Systems

Master Thesis
João Moreno

Supervised by:
Prof. Dr. Donald Kossmann
Dr. Tim Kraska
Simon Loesing

Systems Group
http://systems.ethz.ch/
Department of Computer Science
ETH Zürich

March 2010 - September 2010
Abstract

During the past decade, cloud systems have become a major competitor of relational databases as a data storage choice for web applications. A dominant design is yet to be seen, with several modern systems differing in many characteristics, such as data model, consistency promises or interfaces, for example. The choice of a better cloud storage system over another can only be performed under specific and fair benchmarking processes; the evaluation of a system can only be achieved through a complete and detailed testing infrastructure.

This thesis approaches the testing and benchmarking problem relative to cloud storage systems. It presents a distributed testing framework, capable of executing tests on a generic cloud storage system. Tests can cause variable amounts of load, perform simple put / get operations, trace events from the cloud storage system and interact with the cloud and its behaviour. Finally, statistics and graphs can be compiled, which give the user a clear picture of the system’s behaviour and important events throughout a test’s execution.
Acknowledgements

I would like to appreciate all the help, patience and guiding advice Simon Loesing has provided me throughout the whole duration of my thesis - every bit of his help was extremely valuable. I’d also like to thank Dr. Tim Kraska and Prof. Donald Kossmann for their advice and support; Tim’s assistance during our matinal meetings were crucial in guiding the thesis’ course and directions. Thank you!
# Contents

1 Introduction 1

1.1 Motivation and Problem Statement 1
1.2 Contributions 2
1.3 Structure of thesis 2

2 Cloud Storage Systems 3

2.1 Overview 3

2.1.1 The Need to Scale 3
2.1.2 The Cost of Scalability 4
2.1.3 Moving Away from SQL 5
2.1.4 The Pay-for-use Business Model 5

2.2 Existing Systems and Services 6

2.2.1 Amazon’s Dynamo 6
2.2.2 Google’s Bigtable 7
2.2.3 Cassandra 7
2.2.4 Other Systems 7
2.2.5 Cloudy2 8

2.3 Distributed Testing 8

2.3.1 Benchmarking Relational Data Systems 8
2.3.2 Benchmarking Cloud Storage Systems 9
2.3.3 Open Cirrus 10
2.3.4 Cloud Architectures 10

3 Implementation 11

3.1 Overview 11

3.1.1 Implementation Details 12
3.1.2 Definitions 12
3.1.3 Distribution and Synchronization 13
3.1.4 Interaction 13

3.2 Test Files 15

3.2.1 Test File Structure 15
3.2.2 Example Test 28

3.3 Interface 30

3.3.1 Command Line Interface 30
3.3.2 Web Interface 33

4 Experiments and Benchmarks 35

4.1 Experiments on Cloudy2 35
4.1.1 Throughput ........................................ 36
4.1.2 Load Balancing .................................... 38
4.1.3 Cloud Bursting .................................... 39
4.2 Unsuccessful Tests .................................... 40
   4.2.1 Fault Tolerance .................................. 40
   4.2.2 Correctness ...................................... 41
4.3 Problems and Limitations .............................. 41

5 Conclusion ............................................ 45
   5.1 Summary ........................................... 45
   5.2 Future Work ........................................ 45

A Code Listings .......................................... 47
Chapter 1

Introduction

This first chapter introduces the problem of testing cloud storage systems; it motivates the work behind this thesis, emphasizing the need for such a solution. Then, it describes the testing framework developed throughout the thesis’ duration, focusing on its contributions.

1.1 Motivation and Problem Statement

As cloud storage systems become the tool of choice for increasingly more web applications and services’ data storage solutions, the number of cloud solutions in the market also increases, pointing towards a dominant design yet to be seen. The relaxation of consistency guarantees opened the road to scalability and spawned several new ways of storing and handling data, which resulted in several existing cloud storage systems nowadays. The "X is better than Y" type comparisons can’t yet be made, due to the lack of proper testing and benchmarking tools. As in the cloud storage market, the cloud storage testing and benchmarking market is also a growth market.

The nature of a cloud storage system as a distributed system implies heavy parallelism and thus non-determinism in its behaviour, which renders testing and validation a very hard task to accomplish. It becomes difficult to consistently repeat the exact same actions in order to debug a problem that might emerge in the system. In this sense, testing such a system is a hard and repetitive task but nevertheless crucial for development. It is important that one manages to assure that some properties remain true throughout the lifetime of a cloud data storage system, such as

- scalability;
- cost optimization;
- security;
- fault tolerance;
- consistency promises;
- correctness.

Motivation for building a consistent testing framework arose out of the development of Cloudy2 [1, 2, 3]. Cloudy2 is a modular key-value based distributed database under development by the Systems Group at ETH Zürich, and has reached a fairly usable stage. However, it required an automated and consistent
method to evaluate each development stage, which this thesis aims to provide.

Although similar systems exist, they do not delve further into protocol details, specific to the system under test. It is desirable to have the ability to trace messages between nodes that make part of the cloud system, for instance. This becomes an issue, due to the ideally platform independent nature of the testing framework.

1.2 Contributions

This thesis’ contributions have brought us to a step closer into solving the problem mentioned above, through the development of a testing framework that is

• **generic** enough to be adapted to various cloud data storage systems,
• **complete** enough to comprise methods, techniques and algorithms that allow consistent and meaningful testing of such a system, and
• **scalable** enough to be able to stress a heavy load handling system like a cloud storage system.

The **framework** is autonomous and can be interacted with either through a command line or web interface – more details on the framework can be found in Section 3.1. It is ultimately a collection of software modules that are executed simultaneously and, through synchronization, manage to interpret and execute **tests**, which are described in Section 3.2.

1.3 Structure of thesis

The following chapters are organized in the following way:

• **Chapter 2** introduces cloud storage systems, existing solutions and distributed testing;
• **Chapter 3** describes the testing framework, its use, its characteristics and its interfaces;
• **Chapter 4** describes the experiments performed during the thesis, mentioning the problems and limitations faced;
• **Chapter 5** presents conclusions and possible future work on the problem of testing cloud storage systems;
• **Appendix A** lists source code used throughout the experiments detailed in Chapter 4.
Chapter 2

Cloud Storage Systems

This chapter focuses on cloud storage systems. It describes their origin, nature, goals, advantages and disadvantages. Then, it enumerates existing cloud storage systems implementations. It introduces and explains the problems related with testing these systems. Finally it presents some previous work related to the testing issue.

2.1 Overview

The cloud is nowadays an overly popularized term, often considered as a marketing strategy to rebrand old technologies into a new and flashy package. But in fact, even though a lot of the technologies and intellectual progress behind cloud systems were already around before, the actual motivations behind cloud storage systems weren’t.

It’s these motivations, or key points, that indeed allowed and pushed towards the establishment of the cloud term, despite some of its technologies have been around for a while.

Several modern solutions, such as Amazon’s Dynamo [4], Google’s Bigtable [5] and Yahoo’s PNUTS [6], are typical examples of cloud storage systems.

2.1.1 The Need to Scale

Once large internet applications started appearing in the 90’s, the need for faster, more scalable data storages appeared as an urgency, mainly in the field of web development. Not only the amount of data being stored kept increasing, but the quantity of users that accessed that data also grew increasingly faster in decreasingly smaller periods of time. It was a simple consequence of the web’s growing popularity.

The situation turned even more dramatic in the beginning of the century, when the world witnessed the incredible growths some websites faced and the problems that ensued. Such famous examples are websites like Twitter and Facebook; the
former being famous for recording a user growth of around 1.300%\textsuperscript{1} from February 2009 to February 2010.

Suddenly, scalability appears as an increasingly important characteristic of data storage systems, in web application environments. Websites not only needed to accommodate larger quantities of data, but also be prepared to support extremely rapid growths, if faced with success. But, like in so many other problems related to Computer Science, scalability has to be traded off for something else.

2.1.2 The Cost of Scalability

Going back to the 90’s, the *de facto* data storage solutions at the time were relational databases, implemented with RDBMS\textsuperscript{2} software packages. This solution offers a very powerful data model which, through the use of query languages, like SQL, the user could perform elaborate operations on the stored data, with low effort.

Relational databases are extremely powerful in the sense that they allow relating data across data sets and perform operations that understand these relationships. Take for example a *join* operation between two tables – it is an extremely rich operation, content-wise. But its strong point, on the other hand, limits the ability to partition data – relationships make partitioning data a very difficult task. Without partitioning, distribution is not feasible, which translates into a scalability limit.

Consider a distributed system, which faces very light partitioning restrictions. Then, two key points crucial to scaling data storage systems become easy to implement:

- partitioning data across the system’s nodes would reduce each node’s load, and
- replication would allow data to be accessed from more than one node, granting the system high availability.

The CAP theorem \cite{7} defines that, in shared data systems, out of *consistency*, *availability* and *partition tolerance*, one must be left out. Relational databases find their equilibrium by relaxing partition tolerance and thus are able to guarantee both high availability as well as strong consistency levels, as defined in the ACID\textsuperscript{3} properties.

If partition tolerance is the starting point towards scalability, either availability or consistency need to be considered less strict. Considering availability as of the essence, the solution appeared through both a data model paradigm change as well as a consistency relaxation.

\textsuperscript{1}Data from Nielsen Media Research – http://blog.nielsen.com/nielsenwire/online-mobile/twitters-tweet-smell-of-success/

\textsuperscript{2}Relational Database Management Systems.

\textsuperscript{3}ACID stands for atomicity, consistency, isolation and durability.
2.1 Overview

2.1.3 Moving Away from SQL

The shift in the data model originated the term NoSQL. By affording to lose its rich semantics, such as relations, data could be partitioned easier, thus creating a data store that is not accessed through SQL. Relaxing the data model originated two approaches:

- non-structured data, which completely disregards the data’s semantics and relies on the client to manage data syntax, semantics and relationships, of which Dynamo is a good example, and
- semi-structured data, which is a compromise between both extremes of the spectrum and understands some data semantics, allowing for more expressive query languages – Bigtable and PNUTS are both semi-structured data storage systems.

NoSQL is often associated with key / value storage. And indeed many implementations are distributed hash tables that, given a key or range of keys, return data as the value – take Dynamo, for example. But other systems take advantage of the semi-structured nature of data to be able to deliver higher expressiveness in handling data. For instance, Bigtable organizes data by indexing it both row and column-wise.

As mentioned before, granting partition tolerance and availability, the trade-off implies that consistency has to be less strict. Relational database systems, under the ACID properties, guarantee that in between transactions, the system is always in a consistent state. NoSQL systems need to work in a less constricted environment, given its distributed nature.

**Eventual consistency** [8] is such a compromise – it guarantees that, in the absence of new updates to a certain data, eventually accessing that data will always return the latest value updated, independently from where the data is accessed. Most cloud storage systems rely upon some variation of eventual consistency, although several compromises can also be achieved.

2.1.4 The Pay-for-use Business Model

The ability to distribute not only influenced the scalability of data storage systems but also the way they are employed in the practical world. Before cloud storage systems, the typical data center would be bought and maintained by the company that owned the data. These would typically be big mainframes of expensive hardware, which would incur not only purchase costs, but maintenance costs as well. Also, if data outgrew the system’s constraints, extra expenses would ensue. On the other hand, if the system’s usage didn’t match the demands originally planned, little could be done.

Distributed storage systems leveraged the idea that mid-range hardware would be enough to deliver the service, since load would be alleviated throughout the system’s nodes. And since each of the system’s components became cheaper, the potential to grow or shrink the system also became cheaper and thus feasible. This is called elasticity.
This is the most essential factor why the cloud has become a big phenomenon today: data storage companies began offering pay-per-use storage services, able to handle as much data and load as the client would need. Data owners needn’t care much about the infra-structure behind their data storage systems, and are able to be billed for exactly the use they demand from it. And data storage providers have their costs reduced by using less expensive hardware. It’s a win-win situation.

2.2 Existing Systems and Services

As more and more consumer oriented web applications spawned online, usually as products from potentially high growth internet startups, the need for elasticity in the services provided became evident.

Upon a new product launch, optimism foresaw big growths, which demanded for big expenditures in infrastructure; while pessimism warned about no popularity at all, advising to wait for the public’s reaction. Either the company would wait for increasing popularity and risk not being ready when it would reach the spotlight; or it would invest for the future by acquiring costly infrastructures and risk financial loss over time.

What follows is a collection of the most popular cloud systems and solutions that are used in practice today. Most of these systems originated from the need the companies behind them also had for scalable and elastic storage and computing space.

2.2.1 Amazon’s Dynamo

Realizing the demand for such services, back in 2006 Amazon started offering several cloud solutions, of which the Simple Storage System [9], or S3, and the Elastic Computing Cloud [10], or EC2, were the most popular ones. While the former offers data storage, the latter makes computing instances, or virtual machines, available for computation on demand. Combined together, clients have a platform that is as elastic as it is needed and as costly as it is used.

Dynamo [4] is Amazon’s ”highly available key-value store” and is used to power Amazon’s web services, such as S3 [11]. Inside the Simple Storage System, data is stored as a value, matching a respective key and bucket. While the bucket identifies a collection of key / value pairs, the key identifies the data itself. It can be accessed either using a RESTful API or SOAP. More recently, Amazon added the option to restrict data to geographical regions, either for latency optimization, cost reduction or legal restrictions; this feature comes easily, given the distributed nature of Dynamo, as a cloud storage system. Abstract layers can also be added on top of S3; the BitTorrent protocol interface is such an example.

Nowadays, Amazon.com is powered by S3’s data storage capabilities. Amazon.com is Amazon’s electronic commerce website, currently being the largest

---

4 Representational State Transfer.
5 Simple Object Access Protocol
2.2 Existing Systems and Services

online retailer in America.

2.2.2 Google’s Bigtable

Google, as a leader in Internet search facilities and later as a service provider with an extensive list of web applications, faced the scalability problem early in this decade. Bigtable [5] started its development in 2004, saw its publication in 2006, and is now being used to power many of Google’s services, such as GMail, Google Maps and Google Reader.

Bigtable is built on Google File System [12], Google’s own distributed file system, with all the characteristics of a cloud service. Unlike Dynamo, Bigtable can be seen as “a sparse, distributed multi-dimensional sorted map”. Data is stored in tables and accessed in a multidimensional way: both row and column-wise indexing as well as versioning provide the user with a planar space and a time dimension.

Bigtable is currently available for public use only through Google’s services. However, Google App Engine service provides a web development platform – which consists of both computing power and data storage – and is powered by Bigtable.

2.2.3 Cassandra

Another project that has been gathering up attention in the industry is Cassandra [13]. Created by Facebook in 2008, it has recently been adopted by Apache, which drives its main development nowadays.

The motivations behind Cassandra’s creation were to both possess Dynamo’s partitioning and replication features and provide a richer data model, such as Bigtable’s. A deployed system is organized in a DHT fashion, granting single-hop access to data from the client. Being an analogy to a multidimensional map, Cassandra organizes data in rows of columns and column families, providing a rich interface for tabled data.

Nowadays it powers several popular web applications such as Facebook, Digg and Reddit, and is gaining some popularity on the industry for being entirely open source, making it an excellent choice for deployment and use on own hardware.

2.2.4 Other Systems

As a direct competitor of Google, Yahoo also pursued distributed storage to accommodate its services needs, most notably in the Flickr [14] photo sharing web application. Yahoo published PNUTS [6] in 2008 as a hosted and centrally managed data storage system. Although less popular than Dynamo or Bigtable, PNUTS offers a data model that is closer to relational databases, taking advantage of developers’ familiarity. Data is stored in either hashed or ordered tables, in a simplified relational model. This simplification implies that, for example, the system does not manage relational integrity or does not allow join or group-by queries.

Other systems have also appeared, many being small open source projects, that
Cloud Storage Systems

have scalability as their priority and are able to handle considerably big loads, while serving several clients simultaneously. Some examples are:

- Voldemort
- CouchDB
- MongoDB
- HyperTable
- HBase
- Scalaris

2.2.5 Cloudy2

Cloudy2, developed at ETH Zürich, was built with the flexibility problem in mind. Its component based nature allows modules to be plugged in and out and create very different behaviors to reach the same end: host data in the cloud. Thus, it solves the problem by allowing the cloud storage system to adapt itself to miscellaneous application’s needs, via modularity and configurability.

This thesis originated from the development of Cloudy2. There was a need to incorporate a consistent, complete and autonomous testing system to Cloudy2’s development, in order to maintain stabler project builds, throughout Cloudy2’s development. As a consequence, this thesis’s contributions were built considering Cloudy2 as the default testee framework, while always keeping generality as a key point.

2.3 Distributed Testing

The very distributed nature of cloud storage systems, albeit opening the doors to scalability, availability and security, makes testing these systems, as well as benchmarking them, a hard task to accomplish. Factors like

- high concurrency,
- elasticity,
- many possible failure points,
- failure resilience, and
- large scale

are some of the reasons why testing these systems is hard.

Testing each separate component of a distributed system, in isolation, is of course an important step throughout the system’s development. Such testing can be done using common and well known tools like unit testing and debuggers, for example. But to make sure that the system behaves as expected, and that it delivers the promises needed to be kept to its users, requires testing it as a distributed system.

2.3.1 Benchmarking Relational Data Systems

Relational data based storage systems, having been around for quite a while [15], are nowadays quite satisfied as to testing and benchmarking methods, platforms
2.3 Distributed Testing

and frameworks. For instance, the TPC-benchmarks [16, 17, 18] are standard
database benchmarks that output specific and comparable results, when applied
to several different database implementations. They were created when relational
databases were the choice per default for any large web application’s data man-
agement; and they served their purpose well, for that matter.

Cloud storage systems, on the other hand, besides presenting characteristics that
are not part of the systems the TPC-benchmarks aim at, also lack features that
are in the very nature of those systems. First of all, cloud storage systems are
elastic, thus have the ability to improve their setup over time, matching demand.
The TPC-benchmarks require that the system under test stays in a fixed configu-
ratation, over the course of the benchmark. Also, the TPC-benchmarks rely on the
fact that the system under test is transaction-based and complies to the ACID
properties, which clearly are not present in cloud storage systems.

These crucial points lead to the conclusion that the current standard benchmarks
for data storage systems might need rethinking.

2.3.2 Benchmarking Cloud Storage Systems

The problem of standardizing the testing and benchmarking cloud storage systems
is still a very open one, mainly for three reasons:

• there is no common ground as to how should a cloud storage system should
  be built – nowadays implementations vary in terms of data models, con-
sistency promises, data access language expressiveness, etc – this not only
  reflects on different external interfaces but also on different inner workings,
  between different implementations;

• due to high concurrency, covering every possible use case of the system is
  very difficult and unfeasible; and

• the size these systems are able to achieve, network-wise, can reach hundreds
  or thousands of nodes – and problems might not occur until such size is
  reached.

The benchmarking problem was approached by [19], in which the authors argue
why existing data storage benchmarking approaches are not suited to fit cloud
storage systems’ behaviors and characteristics and discuss ideas towards building
a new benchmarking system. Here, the problem of creating a systematic way of
being able to compare cloud storage systems with one another is evident.

Yahoo Cloud Serving Benchmark

More specifically, Yahoo has taken the matter into practice and developed a bench-
mark system specifically towards cloud key/value data storage systems evaluation,
named Yahoo Cloud Serving Benchmark [20]. Its main goal is the development
of both a framework and a set of workloads that can easily be applied to sev-
eral cloud storage systems. Given the results, the storage systems can then be
compared against each other.
Cloudstone

A similar project to the previous one is Cloudstone [21], which is a toolkit developed by UC Berkeley and SUN Microsystems. It aims to provide benchmarking processes based on a web social application, Olio, automation tools that generate load on systems running Olio and a set of constraints for computing an ultimate comparison metric: dollars per user per month.

2.3.3 Open Cirrus

The need for such a standardization of comparison between large-scale distributed systems has also been very much in the interest of the business world. In 2008, HP, Intel and Yahoo announced Open Cirrus [22], a multi-data center, open-source test bed. Its main goals are to

- encourage further development and research on cloud services,
- discover new possible applications for cloud services,
- gather relevant data towards better and more consistent testing, and
- develop further software stacks and APIs for the cloud.

The project has already made available six sites online, each with a least 1000 computing cores, for the sole purpose of increasing cloud systems research both in quantity and in quality.

2.3.4 Cloud Architectures

Finally, the establishment of a benchmarking suited for cloud storage systems would also help answer the question of which cloud architecture is better suited for which applications. In [23] the same problem arises, albeit specific for transaction processing, in which the need for a comparison system between cloud systems is crucial. The authors explore several existing market cloud storage solutions, ranging different architectures, and compare them in benchmark tests.
Chapter 3

Implementation

This chapter describes the framework that was idealized and developed during the course of this Master thesis. An overview is given at first, which describes the main keypoints of the framework, and the following sections will delve further into its details, configuration and usage.

3.1 Overview

In order to understand the design choices for this framework, it is important to fully acknowledge the nature of the systems the framework aims to test. Each system should have its essential characteristics such as being

- distributed,
- large-scale and
- capable of handling considerably big loads.

It is thus essential to provide likewise high-load tests and make sure that these tests are able to cover whichever part of the system we would like to focus on.

The framework described here is itself another distributed system. It is centralized and provides synchronization among all of its parts in order to provide the consistent testing environment we would like to achieve. It features a well-defined structure for test description and multiple output formats for each test’s execution.

The framework is able to run autonomously, which is a desired feature while testing an ongoing development project. It currently supports a command line interface as well as a web interface, in which tests can be uploaded, edited, configured, run and deleted.
3.1.1 Implementation Details

The framework’s core was built using the Java programming language and is expected to run on any Java Virtual Machine version 6 or above.

All communication between the master and its slaves is handled by the remote procedure facilities offered by the cajo [24] project. This project is a lightweight framework designed for bidirectional inter-machine communication. It is very flexible in the sense that Java objects can easily be transferred through the network and communication can happen bi-directionally, via calling regular Java methods over proxy objects, thus allowing more complex patterns than simply the server/client one.

Both test and result files are represented using the YAML [25] format. This is a human-readable data serialization format which offers special features such as references and data merges, which other candidate formats lacked. It allows any user of this framework to easily write and extend test files, given the options available for the tests.

Note that result files only contain information relative to the outcome of a test execution. This outcome is generated from possibly large data sets; as an example, we might want to examine the population quartiles of all response times during one hour of high-load interaction with the cloud. And thus, every task’s results are stored in local databases using the SQLite [26] software library, for higher efficiency and speed. These databases are transferred to the master upon test completion and the master originates the result files from their contents.

3.1.2 Definitions

Clear definitions of several of this framework’s key points are presented and explained in this section.

- The master is the central coordinator of a running instance of the framework. It interprets tests, distributes tasks to its slaves, collects results and outputs them.

- Each slave is a task executioner. Multiple slaves connect to a single master, and are synchronized by it. The slaves are the framework’s components that communicate with the cloud service directly.

- A task is a clear instruction set for a slave to perform. Each slave stores each task’s results as it goes through them, executing them serially.

- Each step is a collection of tasks. A step contains exactly as many tasks as the number of slaves associated with the respective test. Thus, each step’s tasks are executed in parallel by each of the slaves.

- A test is a well-defined sequence of steps. It is always constrained to a specific number of slaves.
### 3.1 Overview

#### 3.1.3 Distribution and Synchronization

As hinted previously, this framework is organized in a **master/slave** architecture. The following is an usual sequence of steps while running a test:

1. The master is started and given a test to run. Since every test is bound to a specific number of slaves, it waits for enough slaves to connect to it.

2. Upon successful connection from the slaves, the master sends every slave the corresponding tasks they will need to execute.

3. As the test proceeds, step by step, the slaves are ordered to run their tasks; the master guarantees that each step’s tasks start at the same time. Tasks’ results are continuously stored locally on each slave.

4. At the end of the last step, the master recollects every slave’s results. It can then compile statistics, graphs, etc. about the test execution and output that information in several formats.

Figure 3.1 illustrates an example of a test which requires 4 slaves and consists of 3 steps. The test contains \(3 \times 4 = 12\) tasks in total and they are distributed accordingly to each slave.

#### 3.1.4 Interaction

The system under test can be defined as the **cloud**. To take full advantage of this framework’s features, the cloud should have installed and configured a piece

---

### Table 3.1: Example test distribution.

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Task 1</td>
<td>Task 1</td>
</tr>
<tr>
<td>Task 2</td>
<td>Task 2</td>
<td>Task 2</td>
</tr>
<tr>
<td>Task 3</td>
<td>Task 3</td>
<td>Task 3</td>
</tr>
<tr>
<td>Task 4</td>
<td>Task 4</td>
<td>Task 4</td>
</tr>
</tbody>
</table>

---

**Figure 3.1:** Example test distribution.
of software specially designed for tracing, interacting and reacting to miscellaneous events that occur inside itself.

In Cloudy2, this is called the **Testing Component**. It allows the testing framework to register for event tracing, get routing information about elements, order cloud nodes to sleep and wake up on demand, etc.

More information on interacting with the cloud will be given in the following section, under **Step** description.
3.2 Test Files

This section describes, in detail, how to create a test file that allows one to test a cloud storage system, using this framework. It specifies the syntax and semantics of test files, along with detailed explanations and examples.

3.2.1 Test File Structure

All test files are represented using the YAML format, which comes with a fairly readable syntax description. Basic data (or scalars) can either be numbers, strings or booleans. Complex data aggregates basic data through lists and dictionaries (or hashes).

An example document that defines metadata about this thesis might be written as such:

```yaml
  author: "Joao Moreno"
  months: 6
  institute: { name: "ETH Zurich", location: "Zurich" }
  supervisors: ["Simon Loesing", "Tim Kraska"]
```

The same document can be expressed using a more expanded syntax:

```yaml
  author: "Joao Moreno"
  months: 6
  institute: { name: "ETH Zurich", location: "Zurich" }
  supervisors: ["Simon Loesing", "Tim Kraska"]
```

The YAML format defines more elaborate features that will be used and explained throughout the thesis, as needed. It is, nevertheless, expected to be a format easily readable and rapidly understandable for the intended audience of this thesis.

Time Intervals

Whenever a value is a time interval, a special syntax can be used. This syntax eases the reading and writing of complex time intervals. The following examples are all valid time intervals, and encompass every syntax supported by this value type:

- 200h
- 1h30m
- 40s
- 3.333s
- 10ms
- 1h30m40.001s
Task Structure

A task, as mentioned previously, is a well-defined description of actions for a slave to perform on a single step. It contains information on what the slave should do, when to do it, how to do it, where to do it, for how long, and more.

Basics

The most basic details of a task are the action the slave will perform, the endpoints on which it is going to perform and a description that describes the task. An example task might begin as such:

```
1 description: "Put 1GB of data into the cloud"
2 action: put
3 endpoints:
4  - { address: cloud1.domain.com, port: 9090, interface: java }
5  - { address: cloud2.domain.com, port: 9090, interface: java }
6  - { address: cloud3.domain.com, port: 9090, interface: java }
7 ...
```

This task will put data into the cloud. The cloud can be reached through its endpoints. The endpoints is a list of network endpoints; each endpoint requires at least a specific network address, port and interface, and has an optional name field, for description purposes. An example endpoint might be:

```
1 address: cloud1.domain.com
2 port: 9090
3 interface: java
4 name: cloud1 # optional
```

Possible actions are: get, put, delete and verify. All actions should be self-explanatory except for the last one: the verify action performs a get operation and verifies if the result’s value matches the expected value, yielding a matches number in the results file, indicating exactly how many positive matches occurred throughout the task.

~

Elements

After describing what and where a slave should perform, a task should specify what it should perform with.

The element concept is a generic cloud storage system data unit. Each element consists of a key, a possible value, a possible type and a possible flag that indicates that the element is traceable (explained further).

Element populations can be described either explicitly or randomly, but not both simultaneously.

Explicit populations are described using a list of elements. Every element is a hash which describes the element itself. For example,
3.2 Test Files

```python
key: "eth"
value: "zurich"  # optional
type: "university"  # optional
trace: true  # optional, default = false
```

is an element that describes a University, namely ETH Zürich, and that is traceable. **Traceable** elements are useful for tracing the cloud’s behaviour and its use will be explained further in the **Traces** section, under the **Step** description.

Defining a task using an explicit population requires the elements to be contained in a list, associated with the `elements` key, as such:

```python
elements:
- { key: "eth", value: "zurich" }  
- { key: "epfl", value: "lausagne" }  
- { key: "mit", value: "massachusetts" }  
```

**Random** populations, on the other hand, can be used to specify large collections of elements. They are used when the population’s contents aren’t as important as its diversity and quantity.

Setting a random population to a task can be as simple as adding its details under the `random` key, like so:

```python
random:
name: "names"
seed: 1234
size: 10000
key_length: 5
value_length: 10000  # optional, default = 0
pre_generate: false  # optional, default = true
prefix: "mr_"  # optional, default = None
suffix: "_harris"  # optional, default = None
type: "person"  # optional, default = None
```

A random population should be given a `name`, for identification purposes; each distinct population should have distinct names. The other required fields are the random generator `seed`, the size of the population, defined with the `size` key, and the `key_length` key, which defines how many random characters the keys of the generated elements should have.

All the other fields are optional. The `value_length` field defines, in bytes, how big the generated value should be; the `pre_generate` flag indicates if the population should be generated before the task’s execution and be kept in memory, for faster access; the `prefix` and `suffix` fields apply to the generated elements’ keys; and the `type` is an optional element attribute.

An actual population originated from the example random configuration can be seen on Table 3.1.
### Limits

A task might either be limited by time, by number of steps or not at all. An **unlimited** – or **daemon** – task will run for as long as other limited tasks are running. Thus, in a step, not all tasks can be unlimited, because that would make the test run indefinitely.

A task can be **time bound** by defining a **duration**, which has a time interval value, like so:

```
1 duration: 30m
```

On the other hand, a task can be **bound to a number of steps**. This number defaults to the given population (either explicit or random) size. Nevertheless, this number can be overridden using the **times** and **loop** keys. The latter allows to specify the number of times the **action** should be performed, while the former is a flag that indicates that the population should be looped over, if the opportunity occurs.

For example, given a population size of 100, if a task contains the following keys:

```
1 times: 300
2 loop: true
```

each element of the population will be processed three times (300 steps in total).

But if the **loop** flag is not set (or if it is set to **false**) the task will finish whenever the first limit is hit. Thus, setting **times**: 300 without setting **loop**: true, has no effect on a population with a size lower than 300. On the other hand, if the population size is over 300, then it will be **cropped**: only part of the population will be processed.

A task can also be limited in both time and number of steps and will terminate whenever the first limit will be reached. Then, a task set with the following limits:

```
1 duration: 30m
2 times: 300
3 loop: true
```

will end either after 30 minutes of execution or after 300 steps.
Parallelism

Up until this point, a task was considered to run serially, step after step. Actually, parallelism can be achieved by simply replicating the task’s behaviour through several threads, like so:

```plaintext
threads: 6
```

Another form of parallelism results from ensuring that all steps run according to a specific frequency. This allows a task to cause specific load on the system under test. To enable load testing, the period between consecutive executions of the action on a population’s element must be specified. This value can either be a time interval:

```plaintext
period: 10ms
```
or a more complex structure:

```plaintext
duration: 30m
period: { from: 1s, to: 10ms }
```

In the first case, the task will execute each step at a fixed frequency of 100 requests per second (10 ms $\Rightarrow 100$ s$^{-1}$). In the second one, the frequency will increase linearly$^1$ from 1 to 100 requests per second, over 30 minutes. A task that varies its load over time needs to have an explicit duration value set, such that the linear growth can be configured unambiguously.

These two keys can also be combined:

```plaintext
threads: 4
duration: 30m
period: 100ms
```

In this case, there will be four threads running at 10 requests per second each; the task will then be run at 40 requests per second.

---

$^1$It is the frequency, instead of the period, that increases linearly. This is due to the frequency actually being the parameter one would ideally vary; but it has been proven to be troublesome to use as a configuration parameter. Thus, the period is used as the default configuration parameter, but when it increases or decreases over a task’s duration, it will do so sub-linearly to match a linear decrease or increase in the resulting frequency.
Statistics

To generate statistics out of a task’s execution, there is only one key to set:

```plaintext
statistics: true
```

By default, no statistics are generated from a task’s execution. If activated, statistics will output several results out of the response times, such as average, median, count, quartiles, etc. Activating statistics means adding more computing time to a task’s execution: recording and saving response times always require some extra resources, that would be free otherwise.

~

Graphs

Graphs can be generated from a task’s execution.

Up until this point, the timeline graph is the only type of graph developed for this framework. An example configuration for a timeline graph is:

```plaintext
type: timeline
ongoing: [mean]
overall: [mean, q1, q2, q3]
events: true
errors: true
```

The graph generated from this task would contain an ongoing line of the response times’ mean; overall horizontal marks with the whole set of response times’ mean as well as the first, second and third quartiles; and events and errors.
3.2 Test Files

markers. An example of such a graph can be found in Figure 3.2

All valid ongoing and overall series are:

- mean;
- min;
- max;
- q1 – 25th percentile;
- q2 – 50th percentile, or median;
- q3 – 75th percentile;
- p99 – 99th percentile;
- p999 – 99.9th percentile.

All combinations of these series can be used in both the ongoing and overall lists; their order in each list will determine the actual rendering order in the resulting graph.

Configuring graphs for a specific task requires only that a list of graphs is added under the graphs key, like so:

```
  - type: timeline
    ongoing: [mean]
    overall: [mean, q1, q2, q3]
    events: true
    errors: true
```

There is also a special syntax for the ongoing list of series, which allows smoothing ongoing curves. By using the smooth option in an ongoing series, the outcome line will be sampled less often, resulting in a flatter line. Here’s an example:

```
  ongoing: [{ type: mean, smooth: 0.3 }, q1]
```

In this example, the series mean will be plotted as an ongoing line and will be flattened out 30 percent. The smooth parameter should be in \([0.0, 1.0]\), in which 0.0 means no smoothing at all and 1.0 means maximum smoothing.

Warmup and Cooldown

A task can have warmup and cooldown periods, just by adding these keys to the task’s hash, with a time interval value:

```
  warmup: 1m
  cooldown: 2m
```

During these periods, the slave will perform no action.

---

2Adding a graphs key to a task automatically activates statistics generation as well.

3Note that in the current implementation, Google Charts API [27] is used to generate the chart images; and it currently lacks support for real curve smoothing.
Example Task

```plaintext
description: "Put several names, under increasing load"
action: put
endpoints:
- { address: cloud1.domain.com, port: 9090, interface: java }
- { address: cloud2.domain.com, port: 9090, interface: java }
- { address: cloud3.domain.com, port: 9090, interface: java }
random:
  name: "names"
  seed: 1234
  size: 10000
  key_length: 5
  value_length: 10000
  pre_generate: false
  prefix: "mr_"
  suffix: ".harris"
  type: "person"
  warmup: 30s
  period: { from: 100ms, to: 10ms }
  threads: 4
  loop: true
  duration: 1h30m
  statistics: true
  graphs:
    - type: timeline
      ongoing: [mean]
    overall: [mean, p99]
```

This task will put a random population in the storage system defined by its endpoints. After a warmup of thirty seconds, four threads will run simultaneously for one hour and thirty minutes, increasing the overall frequency of puts linearly from 40 requests per second to 400 requests per second.

The results from this task’s execution will be:

- **statistics**, generated out of the response times of all requests, and
- a **timeline graph**, plotted with an **ongoing mean** line as well as **overall mean and 99th percentile** markers.
Step Structure

The collection of tasks to be run simultaneously by all the slaves, at a given point in time, is defined as a test step.

Basics

A step always requires a description and a list of tasks, with as many elements as the number of slaves required for its test. Each element of the tasks list should either be a task hash, as defined in the previous section, or the string wait which literally means that a slave should do nothing at all and wait until all other slaves have finished their tasks.

An example step, from a test which requires two slaves, would be:

```plaintext
1  description: "Put universities"
2  tasks:
3    - description: "Put universities"
4      action: put
5        elements:
6        - { key: "eth", value: "zurich" }
7        - { key: "epfl", value: "lausagne" }
8        - { key: "mit", value: "massachusetts" }
9        - wait
```

Traces

In each step, the testing framework can be setup to listen to some cloud storage system’s events, as they happen. These events merely logged for later analysis and evaluation. This is called tracing and can be configured with the traces key in a step hash:

```plaintext
1  traces:
2    - endpoints: [{ address: "cloud1.domain.com", port: 14500 }]
3      types: [cloudburst, loadbalance]
4    - endpoints: [{ address: "cloud4.domain.com", port: 14500 }]
5      types: [will_put]
```

In this example, the testing framework will log every cloud bursting and load balancing events coming from cloud1.domain.com as well as every time a traceable element is about to be put in cloud4.domain.com.

Tracing elements becomes evident here: only traceable elements will cause the events to be sent to the testing framework. Regular elements will never fire an event such as will_put.

All possible trace types are in the following list:
The sleep and wakeup traces are fired if a cloud node is sent to sleep or is waken up, from the testing framework, respectively. All other trace types should be self-explanatory.

Events

In the test file context, events are how the testing framework directly interacts with the cloud under test. Events are fired at a specific time instance after the beginning of a step. Here's an example on configuring a step to fire events:

```bash
1 events:
2   - endpoints: [{ address: cloud1.domain.com, port: 14500 }]
3       event: sleep
4       start: 5m
5   - endpoints: [{ address: cloud1.domain.com, port: 14500 }]
6       event: wakeup
7       start: 15m
```

In this example, the node cloud1.domain.com will sleep after 5 minutes of the step's execution and will wake up after another 10 minutes.

As of writing this thesis, sleep and wakeup are the only events that the testing framework can cause on the system.

Triggers

Triggers are a mix between traces and events. They can be though as events that react to traces. Like traces, triggers only apply to traceable elements.

If, for example, one would like to drop a put request of a certain element\(^4\), on a specific node, here's how to do it:

\(^4\)Note that triggers always, and only, apply to traceable elements.
3.2 Test Files

traces:
- endpoints: [{ address: cloud1.domain.com, port: 14500 }]
types: [will_put]

triggers:
- endpoints: [{ address: cloud1.domain.com, port: 14500 }]
types: [will_put]
action: drop

Any traceable element that would be put in `cloud1.domain.com` will be dropped beforehand. This actually means that the cloud, once recognizing the message to put the element on its storage, will drop the handling of that message, as if the message itself was dropped on the network.

An important aspect about triggering is that to be able to trigger based on traces (`will_put`, in this case), those traces need to be configured to be sent to the testing framework, with the `traces` key.

Another possible action would be to modify the element itself, by changing its attributes, like so:

traces: # ...
triggers:
- endpoints: [{ address: cloud1.domain.com, port: 14500 }]
types: [will_delete]
action: modify
modification:
- key: foobar

This will simulate a corrupted message, initially intended to delete a traceable element, that will end up deleting the element with the `foobar` key, on node `cloud1.domain.com`.

There is also a special syntax for an endpoint, which is only used in triggers. A relative endpoint is a cloud node whose identity is given by asking the cloud about who are the nodes responsible for a certain element. Here's an example:

traces: # ...
triggers:
- endpoints:
  - relative_to: { address: cloud4.domain.com, port: 14500 }
    replication_factor: 3
    replica: 2
    types: [will_put]
    action: drop

Consider that the `will_put` trace is configured to be received from every cloud node. Then, this trigger will always drop the second replica of any traceable elements that are to be put in the cloud, given a replication factor of 3 and using `cloud4.domain.com` as a reference point. This is accomplished using the testing framework’s software module that resides within the cloud; it allows the framework to ask cloud nodes about routing information, relative to a given element.
Triggers only support the types

- `will_get`
- `will_put` and
- `will_delete`

because these traces are the only ones that offer some responsiveness: they occur right before the specific event actually happens. And when these traces are triggered, they offer the testing framework a chance to interfere with the cloud’s behaviour.

Possible trigger actions are `drop` and `modify`, up to the date when this thesis was written.

When using the `modify` trigger action, the `modification` key must also be in the trigger hash. Its contents are exactly the same as an `element`, detailed on the `Task` section.

Extra

Statistics and graphs may also be generated out of a step’s execution; its syntax is exactly the same as in a task:

```plaintext
1  statistics: true
2  graphs:
3    - type: timeline
4      ongoing: [mean]
5    overall: [mean, q1, q2, q3]
6    events: true
7    errors: true
```

If configured, step’s statistics and graphs will aggregate all tasks from that step and produce combined results out of them, which matches the total interaction the cloud was under, during that step’s execution.
Test Structure

The test is the encompassing unit that puts everything together. A test is a full blown YAML document. This document always requires a description, the number of slaves required for the test’s execution and its steps. All other keys in the document are ignored. A test normally starts as

```
1  description: Example test
2  slaves: 5
3
4  # steps and tasks
```

which indicates that this test requires five slaves to be executed.
3.2.2 Example Test

---
description: Example test
slaves: 5

endpoints: &cloud
- &n1 { address: node1.cloud.com, port: 9090, interface: java }
- &n2 { <<: *n1, address: node2.cloud.com }
- &n3 { <<: *n1, address: node3.cloud.com }

cajo: &cajo
- { <<: *n1, port: 14500 }
- { <<: *n2, port: 14500 }
- { <<: *n3, port: 14500 }

tasks:
- &put_1
description: Write a population
action: put
random: &r1
  name: r1
  seed: 1
  size: 1000000
  key_length: 10
  value_length: 20000
endpoints: *cloud
statistics: true
loop: true
period: { from: 80ms, to: 40ms }
threads: 4

- &put_2 { <<: *put_1, random: { <<: *r1, name: r2, seed: 2 } }
- &put_3 { <<: *put_1, random: { <<: *r1, name: r3, seed: 3 } }
- &put_4 { <<: *put_1, random: { <<: *r1, name: r4, seed: 4 } }
- &put_5 { <<: *put_1, random: { <<: *r1, name: r5, seed: 5 } }

steps:
- description: Write only
statistics: true
traces: &trace
  endpoints: *cajo
types: [cloudburst, cloudcollapse, loadbalance]

~
3.2 Test Files

This test requires five slaves and during its single step, all slaves will put data into three cloud endpoints:

- node1.cloud.com,
- node2.cloud.com and
- node3.cloud.com,

and will increase their load for 30 minutes. Since each slave has four threads, each slave will go from 50 \((4 \ast 1/0.08)\) to 100 \((4 \ast 1/0.04)\) requests per second, over the 30 minutes duration of the step.

Also, the testing framework will receive all cloud bursing and collapsing and load balancing events, to be logged in the results file. Notice that the endpoints used to configure the traces are the same, but with different ports, which connect to the Cajo interface in each node’s testing component.

Finally, the outcome of this test will the a results file with each task’s statistics and the step’s traces logged.

Note the extensive use of extra keys in the test document (endpoints, cajo and tasks) – since they are ignored, they can be helpful in writing a shorter test file, and keeping it both well organized and easily readable. Also, the use of YAML’s data merges and references helps considerably in keeping the file compact.
3.3 Interface

This section presents the interfaces through which the testing framework can currently be used.

3.3.1 Command Line Interface

The most flexible way to use the testing framework is through the use of a command line interface. GNU-like programs are exported from the Java platform to execute each of the framework’s components. Also, extra command line scripts for bash were tailored to fit most of the hypothetical user’s necessities and ease the framework’s usage. Thus, the command line scripts have become the framework’s default interface.

The following paragraphs will show how to use these scripts to run a test and output results. For a more detailed use of the command line interface, refer to the scripts’ documentation, which is packed along with the contributions of this thesis.

Setup

Along with the scripts comes an extra script called setup, which intends to set up an environment suited for the framework’s needs. This is needed because most of the scripts have dependencies on other scripts, and the exact directory in which these scripts are place has to be known. Then, to set up this environment, the user only needs to perform the following simple step:

1. $ cd path_to_testing_framework/
2. $ . setup

Once this is done, the framework can be used, from the current shell session, without restrictions or references problems.

The following examples assume that the current shell session is already set up in this fashion.

Master

The master script launches a master instance that parses a test file, sets up a results path and waits for slaves to connect. An example of its usage is:

1. $ master --testfile test.yaml \
2. --output-path test_results/
Simply put, this previous example executes a master instance, which parses the test file `test.yaml`, prepares the `test_results/` for results output and it immediately waits for the test’s required number of slaves to connect. Once enough slaves connect, it distributes the tasks to each of the slaves, synchronizes the whole test’s execution, outputs the results into `test_results/` and exits.

The master can also be launched remotely, given that the framework is also installed in a remote machine, in a known path, through `ssh`. The same command as the previous one, would be executed remotely like so:

```bash
$ remote_master master.domain.com \
   --testfile test.yaml \
   --output-path test_results/
```

**Slave**

Since most of the framework’s configurations come from the test file itself and the master’s instance, the script to execute a slave is pretty straightforward:

```bash
$ slave --master-hostname master.domain.com
```

This example will prepare a slave instance running on the local machine, which tries to connect to a listening master, at `master.domain.com`. All other test details will be sent by its master. Once it executes all the tasks given, the program will exit.

Analogously, a remote slave would be executed as such:

```bash
$ remote_slave slave1.domain.com \
   --master-hostname master.domain.com
```

**Single Script Run**

It is also possible to execute a test through the use of a single script, called `run_test`. It becomes a fairly easier way to use the framework, although it imposes certain limitations:

- the master needs to be executed locally, while the slaves need to be remote;
- no customization can be done to the master and slaves instances, such as the communication port or timeout periods.

Here’s a demonstration, that combines the previous examples into one single program call, but using two slaves, instead of one:

```bash
$ run_test test.yaml results_path/ master.domain.com \
   slave1.domain.com slave2.domain.com
```
With its strict syntax, the \texttt{run\_test} command doesn’t leave much choice for configuration. It is, nevertheless, the script used behind the scenes of the web interface, detailed in the following section.

\texttt{\~}

\textbf{Results}

Since a test’s execution is a lengthy process time-wise, re-running a test to obtain the same results but in a different output is indeed an excessive task. For that purpose, the \texttt{results} script was created. It picks up the data collected from the master and slaves during a past test’s execution, and recalculates statistics, recompiles graphs, etc. This script can be used as such:

```
1 $ cd output_path/
2 $ results --master-db master.db \ 
3    --slaves-db slave1.db,slave2.db \ 
4    --testfile test.yaml \ 
5    --output-path .
```

The example will recompile the test’s execution results, as specified in the \texttt{test.yaml} file, compiling the results and graph files all over again, from the data stored in the SQLite databases.
3.3 Interface

3.3.2 Web Interface

In contrast with the command line interface, the testing framework’s web interface is the easiest way to make, run and examine tests.

Implementation

The web interface is comprised of both a web server and an HTML client interface. The web server was built using a Python web micro-framework called Flask [28], which provides a basis for fast and simple development of web oriented applications. The client interface is build up on top of standard browser technologies: HTML, CSS and Javascript, including JQuery [29], a Javascript library for fast prototyping.

Web Server

The web server, as mentioned before, is a Python based application. Besides serving as an HTTP content provider, it is responsible for finding and parsing both test files as well as results folders. It is also responsible for executing tests and maintaining a queue of tests execution.

Like the several components of the framework, the web server also has a script which makes executing it an easy task:

```
1 $ cd output_path/
2 $ webservice --tests-path tests/ \
3   --results-path results/ \
```

The server will browse both the `tests/` and the `results/` folder and offer the user a much richer experience through the web interface.

Web Interface

The web client interface – as seen on Figures 3.3 and 3.4 – was designed to facilitate the use of the framework and streamline the process of executing tests. Its main features are:

- **tests management** — uploading, browsing, editing and deleting tests;
- **results management** — browsing and deleting test’s results;
- **tests execution**.
Figure 3.3: Web Interface: graphs visualization.

Figure 3.4: Web Interface: test file visualization.
Chapter 4

Experiments and Benchmarks

This chapter presents the experiments and benchmarks performed on Cloudy2 and executed using the testing framework. It first describes the conditions under which each test executed, then describes the test itself and finally presents its results and analysis. Finally it exposes the problems and difficulties that these experiments faced.

4.1 Experiments on Cloudy2

The following experiments were performed on a local ETH cluster. For every experiment, Cloudy2 was started from scratch, empty of data. A total of twelve machines were granted for this purpose, of which at most six were used as Cloud2 and the rest as the testing framework’s slaves. Table 4.1 contains the setup’s hardware details.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon L5520 2.26 Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>24 GiB</td>
</tr>
<tr>
<td>Hard disk</td>
<td>15k rpm, 73 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Debian Lenny Linux</td>
</tr>
</tbody>
</table>

Table 4.1: Experiments setup characteristics.

Concrete benchmarks and results on Cloudy2’s performance are presented in Flavio Pfaffhauser’s thesis, [2]. The following results demonstrate the possibilities this framework provides for testing cloud storage systems.
4.1.1 Throughput

The throughput experiment stresses the evaluated system under increasing amounts of load, over the course of an hour and a half. The tested system consisted of six cloud nodes, with no extra nodes for cloud bursting, load balancing turned on by default and a replication factor of 3. The framework’s setup constitutes five slaves contacting the cloud storage system at every instance. In total, there are three half-hour steps, each representing a different workload:

- write only,
- read only and
- read/write.

In every step, each slave maintains four threads running simultaneously; each thread contacts the cloud system in linearly increasing frequency. Overall, the load caused on the system grows from 250 to 500 requests per second. The limit of 500 requests per second was imposed to avoid exhausting the number of open files in each slave – this problem is mentioned in Section 4.3. The elements that are respectively put and get from Cloudy2 have uniformly random distributed keys and 100Kb values.

The first observation to take from this test’s execution is the average response times differences between all three steps, as detailed on Table 4.2. Higher response times occur during the write-only phase, due to the replication factor of 3: each write has to be forwarded twice; while the read-only phase presents lower response times.

<table>
<thead>
<tr>
<th>Action</th>
<th>Avg. RT</th>
<th>Action</th>
<th>Avg. RT</th>
<th>Action</th>
<th>Avg. RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave 1</td>
<td>put</td>
<td>61.185 ms</td>
<td>get</td>
<td>20.767 ms</td>
<td>put</td>
</tr>
<tr>
<td>Slave 2</td>
<td>put</td>
<td>60.738 ms</td>
<td>get</td>
<td>20.958 ms</td>
<td>get</td>
</tr>
<tr>
<td>Slave 3</td>
<td>put</td>
<td>62.015 ms</td>
<td>get</td>
<td>20.702 ms</td>
<td>put</td>
</tr>
<tr>
<td>Slave 4</td>
<td>put</td>
<td>61.406 ms</td>
<td>get</td>
<td>20.922 ms</td>
<td>get</td>
</tr>
<tr>
<td>Slave 5</td>
<td>put</td>
<td>62.146 ms</td>
<td>get</td>
<td>20.785 ms</td>
<td>put</td>
</tr>
<tr>
<td>Overall</td>
<td>put</td>
<td>61.498 ms</td>
<td>get</td>
<td>20.827 ms</td>
<td>put/get</td>
</tr>
</tbody>
</table>

Table 4.2: Throughput test.

The second observation can be done to the variation of the response time through the course of the several steps. Figures 4.1 and 4.2 illustrate both the load provoked by all slaves simultaneously as well as the system’s average response times, over time, for Steps 1 and 2 respectively. They also have horizontal marks for the first, second and third quartiles of all the response times observed. Overall, a sharper increase can be seen in the first step, as well as more variance in the response times, characterized by more oscillations in the curve.
4.1 Experiments on Cloudy2

Figure 4.1: Throughput Test, Step 1, Write only, System Throughput.

Figure 4.2: Throughput Test, Step 2, Read only, System Throughput.
Overall, this test demonstrates the framework’s ability to coordinate several slaves and perform a load test under very strict conditions – increasing the load linearly, for instance – as well as its plotting capabilities, being able to present relevant and easily recognizable data for fast performance evaluation.

This test’s YAML file can be found in Listing 1, page 48; its results are in Listing 2, page 49.

4.1.2 Load Balancing

This test had the goal of observing the system’s ability to react to load unbalances throughout the cloud’s nodes. Load balancing is a scheduled short-term mechanism that, if needed, allows the system to improve its performance by modifying each node’s responsibilities and transferring data between nodes.

To force load balancing events, a single slave puts 10Gb of biased information using elements of 10Kb in size. The bias is guaranteed by putting elements with keys that start with the sequence "aaa"; since distribution of data is done lexicographically in Cloudy2, in this way the cloud will have to adapt itself such that every node in the system will eventually have the same responsibility in the system, approximately. The slave has 100 threads putting data as fast as they can, with no restrictions to load.

Figure 4.3 details the system’s performance over time, measured by the average response times collected by all the 100 threads, during the first step. Overall, 1,000,000 elements of 10 Kb in size were put in the system, for roughly eight minutes and four seconds; the average load on the system was 2.066 requests per second. It’s observable that several load balancing events occurred around the beginning of the first minute, which allowed the cloud to adapt itself to the biased
4.1 Experiments on Cloudy2

<table>
<thead>
<tr>
<th>Instant</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0h00m</td>
<td>Test begins.</td>
</tr>
<tr>
<td>0h10m</td>
<td>First slave starts its task.</td>
</tr>
<tr>
<td>0h20m</td>
<td>Second slave starts.</td>
</tr>
<tr>
<td>0h30m</td>
<td>Third slave starts.</td>
</tr>
<tr>
<td>0h40m</td>
<td>Fourth slave starts.</td>
</tr>
<tr>
<td>0h50m</td>
<td>Fourth slave stops its task.</td>
</tr>
<tr>
<td>1h00m</td>
<td>Third slave stops.</td>
</tr>
<tr>
<td>1h10m</td>
<td>Second slave stops.</td>
</tr>
<tr>
<td>1h20m</td>
<td>First slave stops.</td>
</tr>
<tr>
<td>1h30m</td>
<td>End of test.</td>
</tr>
</tbody>
</table>

Table 4.3: Cloud Bursting test schedule.

population and thus spread the entire population’s responsibility throughout all the nodes. A drop in the response times confirms the improvements caused by the load balancing adjustments.

This test demonstrates the framework’s freedom in defining very specific random configurations for element populations generation, as well as fine observation of events that result in direct changes to system’s performance.

This test’s YAML file can be found in Listing 3, page 50; its results are in Listing 4, page 51.

4.1.3 Cloud Bursting

Cloud bursting is the cloud’s ability to expand in the face of high overall load – it’s a long term mechanism to achieve scalability. In order to test cloud bursting, the initial setup for this test consisted of three nodes running Cloudy2 initially, with the possibility of expanding to a total of six nodes, and four slaves creating load on the system. Each slave maintains four threads putting 10Kb random elements in the cloud at 50 requests per second, each. Thus, at the test’s peak, the cloud is under a load of

\[ 4 \text{ slaves} \times 4 \text{ threads} \times 50 \text{ req/s} = 800 \text{ req/s} \]

The whole test is scheduled in order to first increase the load caused in the system by starting successive slaves 10 minutes apart, and then decrease the load by stopping the slaves also 10 minutes apart. Table 4.3 is a better representation of this test’s schedule.

This test’s results can be better observed in Figure 4.4, where the overall system load is plotted against the average of all the response times measured over the course of the test, with the load balancing and cloud bursting events marked as well. Three cloud bursting events occurred, all during the 10 minute period when the system was under maximal load. Its observable a sharp increase in the re-
spose times average followed by a stabilization, right after each cloud bursting event; this is due to the transfer of data between existing cloud nodes and the newly arrived one to match the new distribution of data.

The framework proved successful in dramatically changing load levels over the course of a long duration test and demonstrating the immediate consequences of cloud bursting events on the system.

The Cloud Bursting test’s YAML file can be found in Listing 5, page 52; its results are in Listing 6, page 53.

### 4.2 Unsuccessful Tests

Several problems and setbacks made obtaining meaningful results for Cloudy2 a troublesome task, some related to the resources available, others to the still-in-development nature of the project. This section presents extra tests that were assembled to test other features of Cloudy2 but did not succeed as expected.

#### 4.2.1 Fault Tolerance

To test fault tolerance capabilities of Cloudy2, the testing framework caused heavy load on the system for a certain period of time; midway that task, one of the nodes would be shut down using a `sleep` event.

As for the expected observations, there are two distinct cases, concerning the size of the initial cloud:

- if the number of initial cloud nodes is less than the replication factor imposed by the cloud’s protocol, as soon as one of the nodes would go amiss, a new
node should be spawned and join the cloud;

- else, the cloud should make sure that, as soon as one of the nodes would disappear, data would be transferred around to guarantee the replication factor’s assertion and all the cloud’s properties should remain the same.

Upon this test’s execution, the first case resulted in several errors and exceptions, observable from the client side, for the system didn’t automatically increase its size, when the number of nodes in the cloud became less than the protocol’s replication factor.

The second case didn’t cause the cloud to yield the errors and exceptions, thus the failure remained transparent to the client. But, in the case that the shutdown node was the cloud bursting leader, the remaining nodes failed to elect a new leader, thus remained static in size indefinitely. The cloud bursting leader is the cloud node responsible for spawning new nodes, in situations of high load.

This test’s YAML file is in Listing 7, page 54.

4.2.2 Correctness

This simple test relied upon putting a small population in the cloud, corrupting a replica of a single element, and observe if, through time, the system would realize the discrepancy between all three replicas and correct the data accordingly.

In a first step, a population is inserted into the cloud; one of its elements is marked as traceable and through the use of triggers, its first replica’s value is modified. And thus, with a replication factor of 3, there are two identical values spread through the cloud, distinct from a third one. Periodically using the verify action, the cloud is polled about how many of the elements’ values, from the initial population, actually match the expected values.

Over the course of 30 minutes, no changes in the data storage occurred, observable through the match of 25 elements’ values against the expected values, over a total of 26 elements, every time the population was verified. The ideal observable results would be that 26 out of 26 element’s values matched the expected values.

This test’s YAML file can be found in Listing 8, page 55; its results file is in Listing 9, page 56.

4.3 Problems and Limitations

Several setbacks delayed and complicated the experiments performed throughout the course of this thesis. This section presents these problems and elaborates on them.

Computing Instances Limitations

In order to cause an amount of load considerably high to observe certain behaviors in the cloud system, the testing framework needs to have enough slaves contacting
the cloud. In a quantitatively restricted environment, it becomes difficult to properly allocate machines for both sides of the problem – the cloud and the testing framework – while executing meaningful tests that run on large scale systems. The Open Cirrus project, mentioned in Chapter 2, approaches this same problem, by creating an open, high scale platform for deploying and testing cloud systems.

OS Limitations

Many tests ran unsuccessfully due to the soft limit on the number of possible open files, per process, imposed by the operating system. Each of the testing framework’s slaves had a limit of 4096 open file descriptors at any instant. Since each connection to the cloud required 4 file descriptors, there was a limit of roughly 1024 simultaneous connections to the cloud. This limit proved to be both well under the hardware limitations the slaves were subjected to, as well as not enough to create a desired amount of load, in certain tests.

Additionally, once the number of file descriptors would reach its maximum, access to the slaves’ stores would be blocked, preventing the log of that same situation and corrupting the stores’ databases, in some occasions. In sporadic occasions, Java Virtual Machine crashes were also observed, which left no opportunity for error handling, from the testing framework’s side.

Cloudy2

Given Cloudy2’s still-in-development nature, some of its features were not fully developed, at the time of testing. Even though the use of the testing framework is exactly targeted to these situations, some of Cloudy2’s components setbacks made the use of the testing framework hard. Notably, Cloudy2’s Java and JavaRouting clients are still beyond completion and failed to expose anomalies to its users. For instance, if the number of open files was reached upon a new client instantiation, the respective exception would not be thrown to the testing framework, rendering the handling of such a situation impossible. Another example is the inability to disconnect clients, and thus close its assigned file descriptors, properly.

Since the client is a major part of testing, being the bridge that connects the system under test with the testing framework, a stable and complete client becomes extremely important.

Fine Tuning of Load Parameters

Forcing some events on Cloudy2, such as cloud bursting and collapsing, proved to be a hard and strenuous work. These events are triggered based on how much load each of the cloud’s nodes is under, at a given moment in time. The quantification of load is performed using a weighted average of several of each node’s components, such as CPU, memory and disk. It proved to be a difficult task to cause the exact amount of load that would force a cloud bursting, for example; and an even more difficult task to relax the amount of load caused, in order to provoke a cloud collapse event.
4.3 Problems and Limitations

Both the quantification of load performed in Cloudy2 and the description of the same problem are presented in [2]. The same frustration behind actually forcing cloud bursting and collapsing events specifically for demonstration purposes is described in an equivalent matter.
Chapter 5

Conclusion

This chapter delivers a summary of the work done throughout this thesis. It then provides further development highlights, which are improvements and features that were not implemented, but should be taken into consideration.

5.1 Summary

This thesis approached the problem of both testing and benchmarking cloud storage systems, while maintaining system independency and offering enough functionalities in order to perform meaningful and complete tests. As contributions, it delivered a testing framework capable of interpreting, distributing and synchronizing tests, as well as outputting execution’s results, such as statistics and graphs.

Executing tests is just a matter of writing test files and feeding them to the testing framework, which takes care of distribution, synchronization and cloud interaction. It’s possible to achieve interaction with the cloud through test file constructs, such as events and triggers. Many different results and outputs can automatically be generated from the framework directly after a test’s execution, which provides the user with fine grained observations over important events that occurred during the execution. Connecting the framework with another cloud storage system requires only the implementation of a small software module, inside the cloud storage system, which complies with the interface established.

The framework, although complete enough for testing some important characteristics of a cloud storage system, would require further development in order to accommodate the fully desired features that an ideal cloud testing framework would be comprised of.

5.2 Future Work

The testing framework developed throughout the course of this thesis is comprehensive, although there are some more features that would make it more complete and thus usable.
Continuous Integration

Considering the framework’s implementation, running a test still requires some human activity, whether it is to start the master and slaves through a command line or clicking a button on the web interface. Thus far, autonomy has been a result of setting up cron jobs\textsuperscript{1} to run the several programs that actually start tests.

The web interface simplifies the whole process by maintaining a job queue, executing all tests in order, one after the other. But it doesn’t have the same flexibility as a cron job, where the user can describe to run a set of tests every day at 4 AM, for example.

An ideal solution to this problem would be the integration of the testing framework with existing development tools, such as Hudson [30], which can provide both scheduling and user interface, as well as automatic access to the project’s source code repository.

Interactivity

The framework has set the path for capturing traces, causing events and running triggers easily, configurable with low effort through test files. It would be further work to develop more traces, events and triggers that would enable more diverse, interactive and useful tests. These new types of cloud interactivity would be developed always with system independence as a critical feature, taking as less assumptions about the cloud system under test as possible.

Results Diversity

Executing a test using the framework might result in a results file with statistics concerning the requests done to the cloud, logged events and errors and graphs. Future work could be focused on increasing the level of detail of the information that one can extract from a test’s execution. All sorts of statistical processes can be applied to a test’s execution data and be outcome in diverse visualization methods.

For example, developing extra graph types would be extremely useful for this framework’s users, as it would deliver more visualization to the potentially extremely rich data that comes out of a test execution.

\textsuperscript{1}Cron jobs are scheduled tasks in a Unix-like system.
Appendix A

Code Listings

This chapter lists source code relevant to the experiments done and results obtained for this framework’s demonstration purposes, as described in Chapter 4.
description: Throughput test
slaves: 5

endpoints: &cloud
- &bach15 { address: bach15.ethz.ch, port: 9090, interface: java }
- &bach16 { <<: *bach15, address: bach16.ethz.ch }
- &bach21 { <<: *bach15, address: bach21.ethz.ch }
- &bach22 { <<: *bach15, address: bach22.ethz.ch }
- &bach23 { <<: *bach15, address: bach23.ethz.ch }
- &bach24 { <<: *bach15, address: bach24.ethz.ch }

cajo: &cajo
- &cajo15 { <<: *bach15, port: 14500 }
- &cajo16 { <<: *bach16, port: 14500 }
- &cajo21 { <<: *bach21, port: 14500 }
- &cajo22 { <<: *bach22, port: 14500 }
- &cajo23 { <<: *bach23, port: 14500 }
- &cajo24 { <<: *bach24, port: 14500 }

tasks:
- &put_1
description: Write 1
action: put
random: &r1
name: r1
seed: 1
size: 10000
key_length: 10
value_length: 100000
pre_generate: false
endpoints: &cloud
statistics: true
graphs: &graphs
  - type: timeline
    ongoing: [mean]
    overall: [q1,q2,q3]
loop: true
duration: 30m
period: { from: 80ms, to: 40ms }
threads: 4

- &put_2 { <<: *put_1, description: Write 2, random: { <<: *r1, name: r2, seed: 2 } }
- &put_3 { <<: *put_1, description: Write 3, random: { <<: *r1, name: r3, seed: 3 } }
- &put_4 { <<: *put_1, description: Write 4, random: { <<: *r1, name: r4, seed: 4 } }
- &put_5 { <<: *put_1, description: Write 5, random: { <<: *r1, name: r5, seed: 5 } }
- &put_6 { <<: *put_1, description: Write 6, random: { <<: *r1, name: r6, seed: 6 } }
- &put_7 { <<: *put_1, description: Write 7, random: { <<: *r1, name: r7, seed: 7 } }

- &get_1 { <<: *put_1, description: Read 1, action: get }
- &get_2 { <<: *put_2, description: Read 2, action: get }
- &get_3 { <<: *put_3, description: Read 3, action: get }
- &get_4 { <<: *put_4, description: Read 4, action: get }
- &get_5 { <<: *put_5, description: Read 5, action: get }

steps:
- &step1
description: Write only
statistics: true
graphs: *graphs

- <<: *step1
description: Read only
tasks: [*get_1, *get_2, *get_3, *get_4, *get_5]

- <<: *step1
description: Read and Write
Listing 2: Throughput results file.
---
description: Load Balancing test
slaves: 1

endpoints: &cloud
- &bach15 { address: bach15.ethz.ch, port: 9090, interface: java }
- &bach16 { <<: *bach15, address: bach16.ethz.ch }
- &bach21 { <<: *bach15, address: bach21.ethz.ch }
- &bach22 { <<: *bach15, address: bach22.ethz.ch }
- &bach23 { <<: *bach15, address: bach23.ethz.ch }
- &bach24 { <<: *bach15, address: bach24.ethz.ch }

cajo: &cajo
- &cajo15 { <<: *bach15, port: 14500 }
- &cajo16 { <<: *bach16, port: 14500 }
- &cajo21 { <<: *bach21, port: 14500 }
- &cajo22 { <<: *bach22, port: 14500 }
- &cajo23 { <<: *bach23, port: 14500 }
- &cajo24 { <<: *bach24, port: 14500 }

tasks:
- &put
description: Write population
action: put
random: &r1
name: r1
seed: 1
size: 1000000
key_length: 10
value_length: 10000
pre_generate: false
prefix: aaa
endpoints: *cloud
statistics: true
graphs: *graphs
  - type: timeline
    ongoing: [mean]
    events: true
    errors: true
    threads: 100

steps:
- description: Put population
tasks: [*put]
endpoints: *cajo
types: [cloudburst, cloudcollapse, loadbalance]
statistics: true
graphs: *graphs

Listing 3: Load Balancing test file.
steps:
  - events:
    - {event: loadbalance, hostname: bach23.ethz.ch, timestamp: 1s06.762s}
    - {event: loadbalance, hostname: bach24.ethz.ch, timestamp: 1s06.958s}
    - {event: loadbalance, hostname: bach21.ethz.ch, timestamp: 1s09.614s}
    - {event: loadbalance, hostname: bach16.ethz.ch, timestamp: 1m23.669s}
    - {event: loadbalance, hostname: bach21.ethz.ch, timestamp: 3m20.826s}
    - {event: loadbalance, hostname: bach23.ethz.ch, timestamp: 5m05.829s}
    - {event: loadbalance, hostname: bach21.ethz.ch, timestamp: 5m39.470s}
  - slaves:
    - runtime: 8m04.452s
      statistics:
      99%: 287
      99.9%: 415
      average: 47.924
      count: 1000000
      max: 3236
      min: 1
      quartiles: [8, 37, 54]
    statistics:
    99%: 287
    99.9%: 415
    average: 47.924
    count: 1000000
    max: 3236
    min: 1
    quartiles: [8, 37, 54]

Listing 4: Load Balancing results file.
---
description: Cloud Bursting test
slaves: 4

epipoints: &cloud
  - &bach15 { address: bach15.ethz.ch, port: 9090, interface: java_routing }
  - &bach16 { <<: *bach15, address: bach16.ethz.ch }
  - &bach21 { <<: *bach15, address: bach21.ethz.ch }

cajo: &cajo
  - &cajo15 { <<: *bach15, port: 14500 }
  - &cajo16 { <<: *bach16, port: 14500 }
  - &cajo21 { <<: *bach21, port: 14500 }

tasks:
  - &task
description: Cause load
  action: put
  random:
    name: pop
    size: 1000000
    key_length: 10
    value_length: 10000
    pre_generate: false
  endpoints: &cloud
  statistics: true
  graphs: &graphs
    - type: timeline
      ongoing: [mean]
      events: true
      errors: true
      loop: true
      period: 20ms
      threads: 4

steps:
  - description: Cause load
    tasks:
      - <<: *task
        random: { <<: *random, seed: 1 }
        warmup: 10m
        duration: 70m
        cooldown: 10m
      - <<: *task
        random: { <<: *random, seed: 2 }
        duration: 50m
        warmup: 20m
        cooldown: 20m
      - <<: *task
        random: { <<: *random, seed: 3 }
        duration: 30m
        warmup: 30m
        cooldown: 30m
      - <<: *task
        random: { <<: *random, seed: 4 }
        duration: 10m
        warmup: 40m
        cooldown: 40m
        statistics: true
        graphs: *graphs
        traces:
          endpoints: *cajo
          types: [cloudburst, cloudcollapse, loadbalance]

Listing 5: Cloud Bursting test file.
steps:
- events:
  - {event: loadbalance, hostname: bach16.ethz.ch, timestamp: 11m45.522s}
  - {event: loadbalance, hostname: bach15.ethz.ch, timestamp: 13m29.795s}
  - {event: loadbalance, hostname: bach15.ethz.ch, timestamp: 15m29.968s}
  - {event: loadbalance, hostname: bach15.ethz.ch, timestamp: 17m30.058s}
  - {event: loadbalance, hostname: bach15.ethz.ch, timestamp: 19m30.175s}
  - {event: loadbalance, hostname: bach15.ethz.ch, timestamp: 21m30.260s}
  - {event: cloudburst, hostname: bach21.ethz.ch, timestamp: 37m10.262s}
  - {event: cloudburst, hostname: bach21.ethz.ch, timestamp: 42m10.791s}
  - {event: cloudburst, hostname: bach21.ethz.ch, timestamp: 47m11.337s}
slaves:
- runtime: 1h30m.038s
  statistics:
  99%: 23
  99.9%: 171
  average: 2.315
  count: 838626
  max: 412
  min: 1
  quartiles: [1, 1, 2]
- runtime: 1h30m.022s
  statistics:
  99%: 33
  99.9%: 171
  average: 2.465
  count: 599187
  max: 408
  min: 1
  quartiles: [1, 1, 2]
- runtime: 1h30m.023s
  statistics:
  99%: 62
  99.9%: 193
  average: 2.979
  count: 359307
  max: 399
  min: 1
  quartiles: [1, 1, 2]
- runtime: 1h30m.029s
  statistics:
  99%: 52
  99.9%: 214
  average: 3.42
  count: 119767
  max: 409
  min: 1
  quartiles: [1, 1, 2]
statistics:
  99%: 36
  99.9%: 179
  average: 2.655
  count: 1916887
  max: 412
  min: 1
  quartiles: [1, 1, 2]

Listing 6: Cloud Bursting results file.
### Listing 7: Fault Tolerance test file.

```yaml
---
description: Fault Tolerance test
slaves: 4
endpoints: &cloud
- &bach15 { address: bach15.ethz.ch, port: 9090, interface: java_routing }
cajo: &cajo
- &cajo15 { <<: *bach15, port: 14500 }
- { <<: *cajo15, address: bach16.ethz.ch }
- { <<: *cajo15, address: bach21.ethz.ch }
- &cajo22 { <<: *cajo15, address: bach22.ethz.ch }
tasks:
  - &task
description: Cause load
  action: put
  random: Random
  name: pop
  size: 1000000
  key_length: 10
  value_length: 10000
  pre_generate: false
  endpoints: &cloud
  statistics: true
  graphs: &graphs
    - type: timeline
      ongoing: [mean]
      overall: [q1, q2, q3]
      events: true
      errors: true
  loop: true
  period: 80ms
  threads: 4
  duration: 30m
steps:
  - &step2
description: Cause load
tasks:
  - { <<: *task, random: { <<: *random, seed: 1 } }
  - { <<: *task, random: { <<: *random, seed: 2 } }
  - { <<: *task, random: { <<: *random, seed: 3 } }
  - { <<: *task, random: { <<: *random, seed: 4 } }
  statistics: true
  graphs: &graphs
  traces:
    endpoints: &cajo
types: [cloudburst, cloudcollapse, loadbalance, sleep, wakeup]
  events:
    endpoints: &cajo22
    event: sleep
    start: 10m
```
---
description: Correctness test
slaves: 1
endpoints: &cloud
  - &bach15 { address: bach15.ethz.ch, port: 9090, interface: *java_routing }
cajo: &cajo
  - &cajo15 { <<: *bach15, port: 14500 }
  - { <<: *cajo15, address: bach16.ethz.ch }
  - { <<: *cajo15, address: bach21.ethz.ch }
  - { <<: *cajo15, address: bach22.ethz.ch }
  - { <<: *cajo15, address: bach23.ethz.ch }
  - { <<: *cajo15, address: bach24.ethz.ch }
tasks:
  - &put
description: Put population
action: put
elements:
  - { key: 'a', value: 'apple' }
  - { key: 'b', value: 'banana' }
  - { key: 'c', value: 'cherry' }
  - { key: 'd', value: 'doughnut' }
  - { key: 'e', value: 'elephant' }
  - { key: 'f', value: 'fight' }
  - { key: 'g', value: 'giraffe' }
  - { key: 'h', value: 'height' }
  - { key: 'i', value: 'ill' }
  - { key: 'j', value: 'joker' }
  - { key: 'k', value: 'kite' }
  - { key: 'l', value: 'lemon' }
  - { key: 'm', value: 'melon' }
  - { key: 'n', value: 'nylon' }
  - { key: 'o', trace: true, value: 'orion' }
  - { key: 'p', value: 'poseidon' }
  - { key: 'q', value: 'quark' }
  - { key: 'r', value: 'roark' }
  - { key: 's', value: 'shut' }
  - { key: 't', value: 'truck' }
  - { key: 'u', value: 'umbrella' }
  - { key: 'v', value: 'vacuum' }
  - { key: 'w', value: 'water' }
  - { key: 'x', value: 'xavier' }
  - { key: 'y', value: 'yes' }
  - { key: 'z', value: 'zebra' }
endpoints: *cloud
statistics: true
  - &verify { <<: *put, action: verify }
  - &wait_verify { <<: *verify, warmup: 5m }
steps:
  - description: Put population
    tasks: [*put]
    traces:
      endpoints: *cajo
      types: [will_put]
    triggers:
      endpoints:
        - relative_to: *cajo15
        - replication_factor: 3
        - replica: 1
      types: [will_put]
      action: modify
      modification:
        value: 'CORRUPTED'
  - { description: Verify 1, tasks: [*verify] }
  - { description: Verify 2, tasks: [*wait_verify] }
  - { description: Verify 3, tasks: [*wait_verify] }
  - { description: Verify 4, tasks: [*wait_verify] }
  - { description: Verify 5, tasks: [*wait_verify] }
  - { description: Verify 6, tasks: [*wait_verify] }
  - { description: Verify 7, tasks: [*wait_verify] }

Listing 8: Correctness test file.
Listing 9: Correctness results file.
Bibliography


