UNIVERSITY OF TRENTO
DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

Distributed algorithms course

Final report

TESTING CONSENSUS ALGORITHMS: A BLACK-BOX APPROACH

Professor: Alberto Montresor
Students: Bortoli Gianluca, Federici Marco, Taneburgo Gianvito

ACADEMIC YEAR 2015-2016

Contents

1 Introduction **1**

2 Platform architecture **3**

2.1 The provisioner 4

2.2 The network manager 6

2.3 The test daemon 6

2.4 The test executor 6

3 Testing language **7**

3.1 Network modification commands 7

3.1.1 Connection specifications 7

3.1.2 Examples 8

3.2 Run command 8

3.2.1 Example 9

3.3 Macros 9

3.3.1 Example 9

4 Test suite **11**

4.1 Tuning the number of operations 12

4.2 Choosing the cluster size 16

4.3 Test suite description 16

4.3.1 Gold standard 16

4.3.2 Symmetric scenarios 20

4.3.3 Asymmetric scenarios 25

4.3.4 Raft vs Paxos 27

4.3.5 Raft vs Raft 27

5 Conclusion **29**

Chapter 1

Introduction

One of the fundamental problem in distributed computing is the so called consensus problem, that arises every time a number of processes (or agents) have to agree on some data value. In the last thirty-five years the consensus problem has been formally defined in terms of properties that must be satisfied by the algorithms trying to solve it. Since then, some results on the possibility or impossibility of consensus have also been proved and some distributed algorithms have been designed to address it with some environment constraints. Nowadays, many commercial products implements these algorithms in their core engines to ensure data is reliably stored on clusters of nodes despite unexpected crashes of the machines.

The goal of this work is to test some of these distributed algorithms in order to gather some statistics about their performances in different conditions, resembling environments with different degrees of criticality. For this reason, almost no details about the consensus problem itself is going to be provided in this document and the general approaches to solve such problem are assumed to be known to the reader.

All the tests performed on the algorithms that will be presented are black-box. No assumption has been done on the implementation language, the underlying technologies used by nodes to communicate over the network or the hardware configuration of the nodes in the cluster. For such reason we expect the designed tests to be meaningful also on every other consensus algorithm. As a consequence, the testing platform can be easily extended to support other implementations that have not been considered in this work. It is right and proper to underline that the tests performed probably do not cover the worst-case scenarios of every algorithm. In order to exploit the intrinsic

criticalities of each algorithm design, a white-box test will be necessary instead.
The consensus algorithms chosen are Paxos and Raft. Two implementations have been considered for each of them, one developed by a big team and used in many commercial systems, while the other developed by a single person. The list of the implementations is the following:

- Datastore: a NoSQL document database developed by Google, built for automatic scaling, high performance and ease of application development. It uses an optimized version of Paxos to reach consensus on objects versions. It is close-source and it is only available as a web-service provided by Google itself.
- Multi-paxos: an academic open-source implementation of Paxos designed to maintain a single replicated value across a cluster. The code is written in Python using the asynchronous programming model implemented by Twisted, a framework providing networking primitives, the asynchronous callback mechanism, event scheduling and the overall reactor loop.
- RethinkDB: an open-source scalable JSON database designed for realtime applications, built over five years by a team of database experts with the help of hundreds of contributors all around the world. The logic underlying the system relies on a C++ implementation of Raft, tightly integrated with lower-level RethinkDB subsystems, to allow replicas to elect an acting primary.
- PySyncObj: a Python library providing data synchronization capability between multiple instances. It uses Raft for leader election and log replication and it offers a convenient interface to transform an arbitrary class into a replicated one.

Chapter 2

Platform architecture

A testing platform has been developed in order to easily evaluate the chosen implementations of the algorithms in different conditions. The architecture of the platform has been designed with two main goals: extendability and test efficiency. In fact, the platform can be easily enriched to support new implementations with little effort, by adding small pieces of code only in some specific components. The other key target is to minimize the overhead introduced by the architecture itself during the testing phase. The platform is composed by five main components, encoded in different Python programs: a provisioner, a test daemon, a network manager and a test executor. A diagram of these components is represented in figure 2.1.

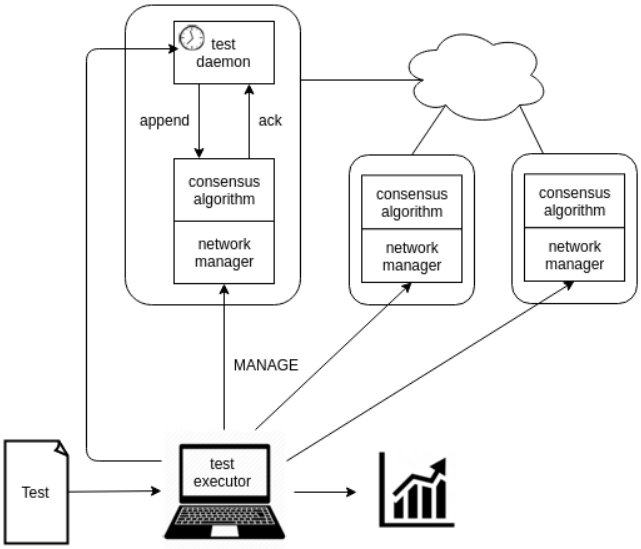

Figure 2.1: Testing platform architecture overview.

2.1 The provisioner

The provisioner is the tool executed before any kind of test is performed. It is is responsible for initializing a cluster of an arbitrary number of nodes and running an algorithm at choice. The cluster can be either launched in a pseudo-distributed mode on the the machine running the script or in a fully-distributed mode on rented virtual machines in the cloud. In the first case, each node will consist in a set of processes running on the local machine; in the latter, nodes will be independent instances of Compute Engine, Google's Infrastructure-as-a-service that is able to provide virtual machines across the world, linked by fiber network. The provisioner is thus capable of setting up two kind of environments where tests can be executed, one local and one in the cloud. This flexibility enables the platform to execute each test in the most suitable environment for the kind of operations planned. For instance, it might not be meaningful to test network communication delays locally or to measure the influence of some artificially-introduced noise in a cloud environment subject to other interfering factors, like network latency.

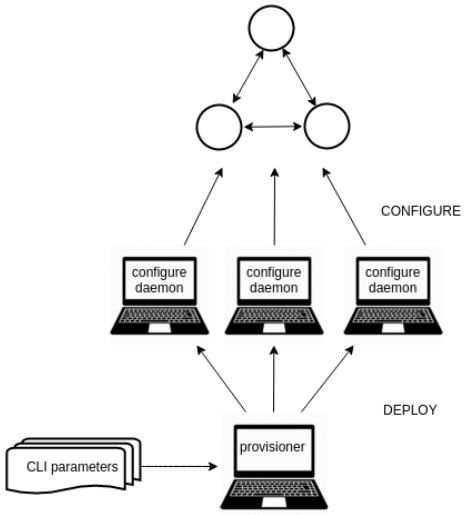

Figure 2.2: Provisioner architecture diagram.

When configuring a pseudo-distributed cluster, the provisioner starts one process for every node and runs the consensus algorithm on each of them as in figure 2.2. If required by the implementation, additional operations are locally executed to properly configure the cluster. A network managers is then launched and binded to every node. Finally, a test daemon is started and binded to an arbitrary node.
On the contrary, the deployment in the cloud involves more steps and requires an additional component to be completed, the configure daemon. To set up a fully-distributed cluster, first, the provisioner uses the Compute Engine API to spin up in a data center as many virtual machines as the number of nodes, attaching a startup script on each of them that will be automatically executed on boot.
Every node will run Ubuntu 15.10 on one virtual CPU and 3.75 GB of memory. The startup script downloads on the virtual machine some platform components and the dependencies required by the algorithm. Finally, it runs a configure daemon. This daemon allows the provisioner to remotely configure the cluster. In this way a test daemon is also started on an arbitrary node.
In both cases the final situation after the deployment resembles the one depicted in the picture 2.1, with multiple nodes connected together, each of them running a configure daemon and a network manager. The test daemon is executed on one of them.

2.2 The network manager

The network manager is the component responsible for controlling the network underlying every node. The script uses netem [1], a low-level kernel tool interacting directly with the network card in order to minimize the interference of this component on the tests. Netem is capable of adding or removing filter rules on the incoming and outgoing packets that are applied before they are dispatched to upper layers of the ISO-OSI stack. The network manager is thus able to corrupt, reorder, delay or lose IP packets in many different ways with minimal overhead on the system. At the same time, the great flexibility of the filters syntax makes modelling some real-life scenarios (e.g. network congestion, partitioned nodes, etc.) very easy.

2.3 The test daemon

The test daemon is a component running only on one node of the cluster. It measures the time required by the consensus algorithm to complete a number of operations specified in input when invoked by the test executor. The unitary operation involves sending a new value to the nodes and wait until they reach consensus. The way this operation is completed differs from one algorithm to another and may involve synchronous API calls, system calls or other kind of interactions. All the operations are executed sequentially and measured individually in a way such that only the effective consensus time is taken into account (external communication overhead is excluded with implementation-specific techniques). The daemon responds to the test executor with the collected times only when the requested number of operations has been successfully completed.

2.4 The test executor

The test executor provides the user a way to interact with the cluster. It can be used to perform tests encoded in a particular language (more details are provided in the next chapter) and outputs a csv file with the results.

[1] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`

Chapter 3

Testing language

A specific language has been designed in order to provide a flexible tool for testing. It is composed by three different kinds of instructions: network modification commands, the run command and macros designed to provide handful shortcuts to simplify the writing and the understanding of the test files. Each command consists in a single line in the test file and an id is assigned to each node to identify it.

3.1 Network modification commands

This set of commands can be used to modify the network status affecting the nodes in the cluster. They are composed of two parts: the first one specifies the connection to address, while the second represents the netem rule to apply.

3.1.1 Connection specifications

Each connection can be either selected at random or be referred by a couple (source, destination). While picking random connection does not grant a lot of flexibility, source and target nodes definition allows users to create much more complex network simulations.
A specific node id can be referred as well as a random one in the cluster by means of the "rand" keyword. Each time it is used, a node id is chosen at random from the ones that are not referred in the context of the same command. This restriction prevents from applying network modifications that would affect the link from a node to itself (loopback).

Moreover, set of nodes can be used both as source and destination in order to allow users to enumerate multiple connection at once. In this case every couple (source i, destination j) belonging to the cartesian product between source set i and destination set j will be affected. Therefore, the keyword "all" has been added to identify the set of all nodes in the network.
Instead of addressing a single direction of a connection, rules can be easily applied to both using the handy "bidirectional" keyword. Finally, a specific syntax (i.e. "on n connections") has been introduced to affect n random connections without the need to specify any source or destination.

3.1.2 Examples

- *from 2 to 3 set delay 100ms*
  IP packets from node with id 2 to node with id 3 are delayed by 100 milliseconds.

- *from 1 to rand bidirectional set loss 10%*
  every packet from node 1 to another random node (apart from node 1) has a 0.1 probability of being dropped. The same rule affects the channel also from the random node to node 1 because of the "bidirectional" keyword.

- *from rand 2 to all set corrupt 20%*
  20% of packets sent by node 2 and another random node (anyone but node 2, because has already been referred) to any other node will be corrupted.

- *on 5 connections set delay 1s*
  apply a 1s delay on 5 connection chosen at random in the network.

3.2 Run command

The run command is responsible for running the consensus protocol on the cluster, collecting the resulting times from the test daemon and saving them into a csv file. The user can specify how many operations he wants to run and a label to be applied to the data results. Consensus time measurement can vary according to the algorithm specific implementation in the test daemon 2.3.

3.2.1 Example

- *run 1000 delay_100ms*
  ask the test daemon to perform 1000 operations, collect times and save a row labelled *delay_100ms* in the csv file.

3.3 Macros

Loops and network reset commands have been also added to the testing language. They do not directly add expressive power to the language, but they provide a handy way to write very common and standard rules.
The first one consists in two instructions, "do" and "n times". The command "do" specifies the beginning of the sequence of instructions that have to be repeated, while "n times" specifies both the end of the loop and the number of repetitions.
A "reset" command has been introduced to reset the network status, removing every modification that has been applied so far by the program.

3.3.1 Example

- *do*
  from all to rand set delay 100ms
  run 1000
  reset
  5 times
  apply the *delay 100ms* netem command on the incoming all the incoming connection to a random node, then runs 1000 consensus and, then, reset the network status. These three operation are repeated 5 times. The "rand" keyword can be resolved with a different node id every time is repeated.

Chapter 4

Test suite

The experiments that are going to be proposed aim at reproducing network configurations that are plausible on real distributed systems. Some of them are specifically crafted for causing troubles to the consensus algorithms taken into consideration and may seem far from typical real-life scenarios. Extreme network problems, such as multiple link failures, node crashes and network partitioning, indeed frequently happen.
The consensus algorithms have been analysed from different points of view. They have been firstly investigated to understand which of them overall performs best in a noise-free scenario. This test also provides the gold standard measure for each implementation to be taken into consideration for every successive comparison. Another key aspect has been the robustness of the consensus algorithms when the underlying network was suffering from problems. Extensive tests have been executed to understand how the system behaves when the network performances degrade gradually.
Before describing every test, it is worth underlying some peculiarities of the considered implementations. Multi-paxos suffers from unexpected crashes (with a probability of $\sim 0.005\%$), especially when filters are applied by netem on the underlying network layer. The test daemon has retried failed operations until no exception was thrown. Particular attention has been paid to RethinkDB in order to obtain results consistent with respect to the other implementations. More specifically, caching mechanisms have been disabled on every node and operations have been configured to use the strongest available consistency level by using appropriate parameters during the API calls. In this way RethinkDB has been forced to reach consensus for every operation and to store the results on persistent storage, as Raft requires. A read operation has been

executed after every write to ensure data was really replicated across the cluster. With these constraints test results were comparable with those gathered on other implementations. Google Datastore has been used in an improper way to some extent. The web-service is specifically designed to work best when batching lot of operations into a single request rather than sequentially uploading small objects, as it was done during the tests. Finally, some problems were encountered running PySyncObj on physically different machines. Due to these limits, tests of this implementation have not been performed on Compute Engine clusters.

4.1 Tuning the number of operations

Before running the test suite, the number of operations to perform had to be defined. Such value had to be chosen carefully in order to obtain statistically significant results. To tune it properly, the followed approach consisted of:

- running twice one algorithm in a normal network environment for a determined number of operations;
- computing the cumulative distribution function (CDF) of the times collected in the previous step;
- comparing the CDFs;
- increasing the number of operations if CDFs were too different or validating the same number on another implementations otherwise.

The correctness of this approach can be justified by the frequentist definition of probability. In the "long run", as the number of trials approaches infinity, the relative frequency will converge exactly to the true probability, as stated by the following formula:

$$P(x < t) = \lim_{nTot \to \infty} \frac{Nx}{nTot}$$

where Nx represents the number of operations where the completion time is lower than t.
Consequently, assuming that the underlying distribution remains the same, the two

curve will both converge to the same originating cumulative distribution function. If two CDFs are similar enough no further investigation is needed, since more operations will only produce more similar distributions to those already computed, adding no significance to the tests. As it is possible to see from the experiments depicted in 4.3 run on RethinkDB, 5000 has been proved to be a reasonably large value.

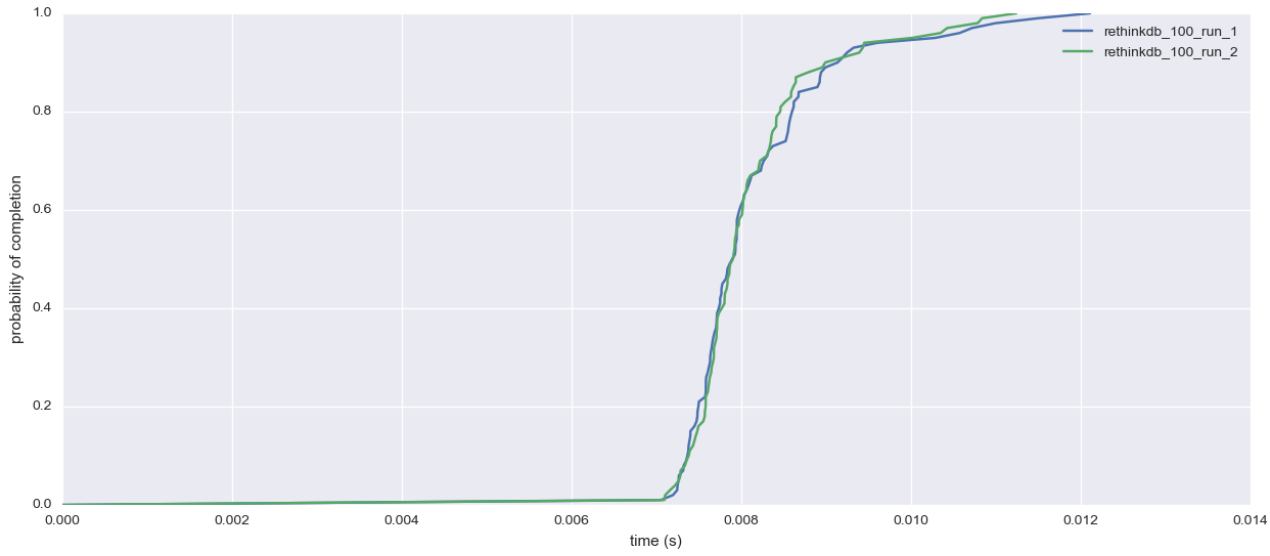

Figure 4.1: RethinkDB: 100 run.

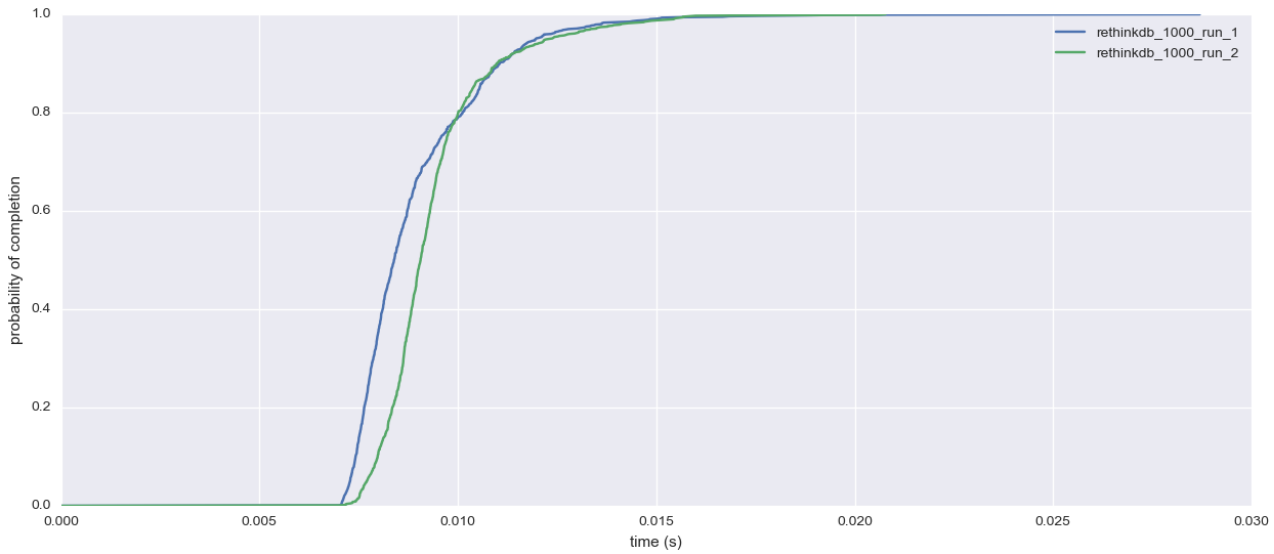

Figure 4.2: RethinkDB: 1000 run.

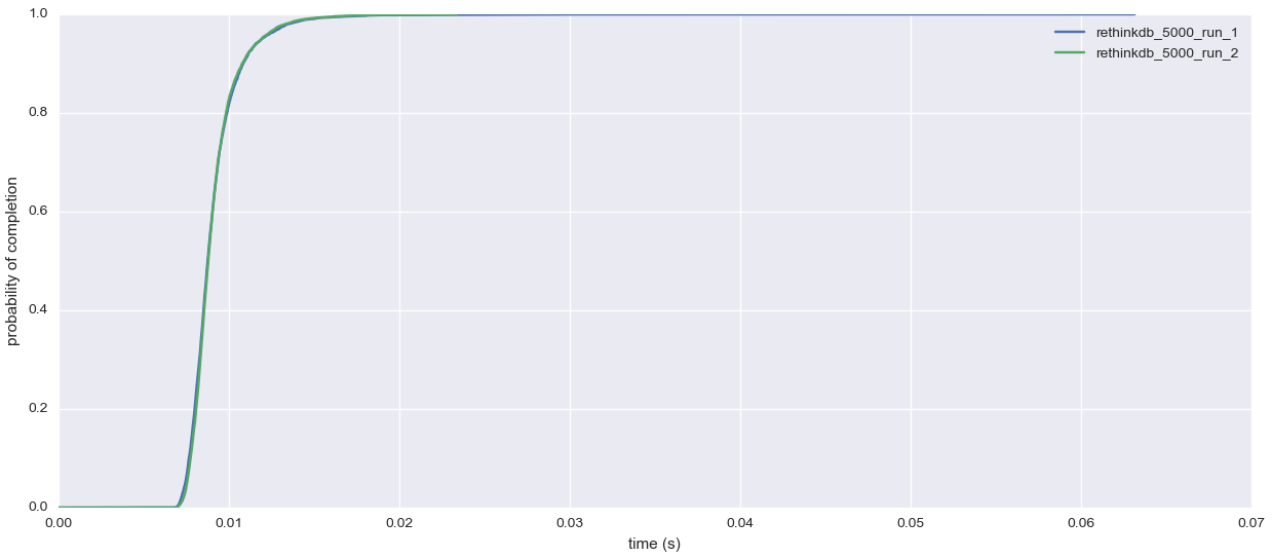

Figure 4.3: RethinkDB: 5000 run.

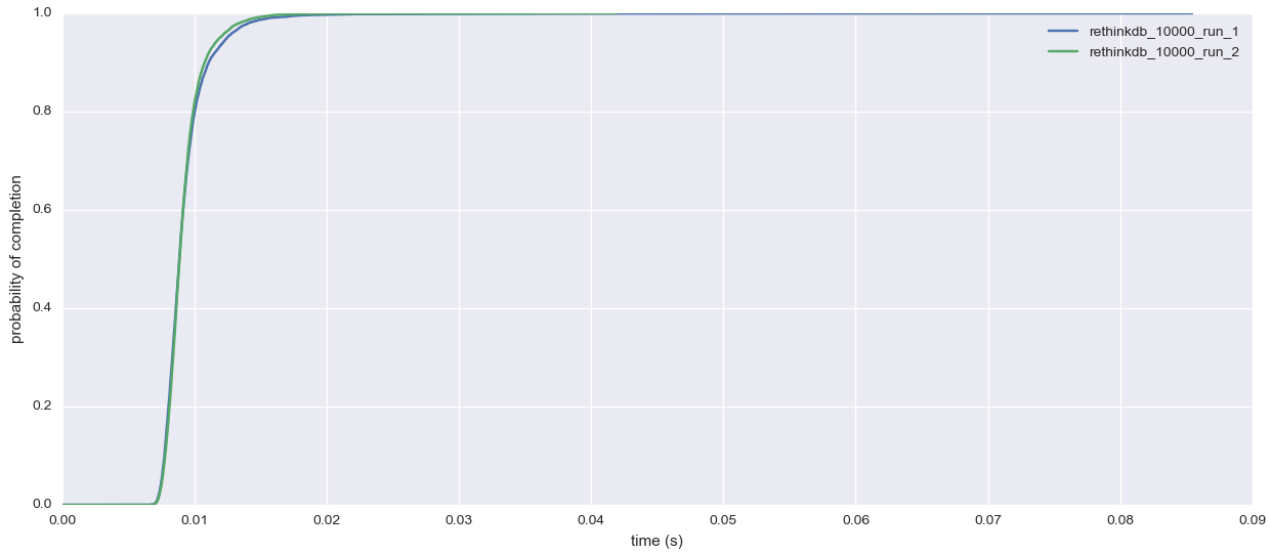

Figure 4.4: RethinkDB: 10000 run.

The parameter has been validated also on every other consensus algorithm. Figure 4.5 shows the test on Multi-paxos.

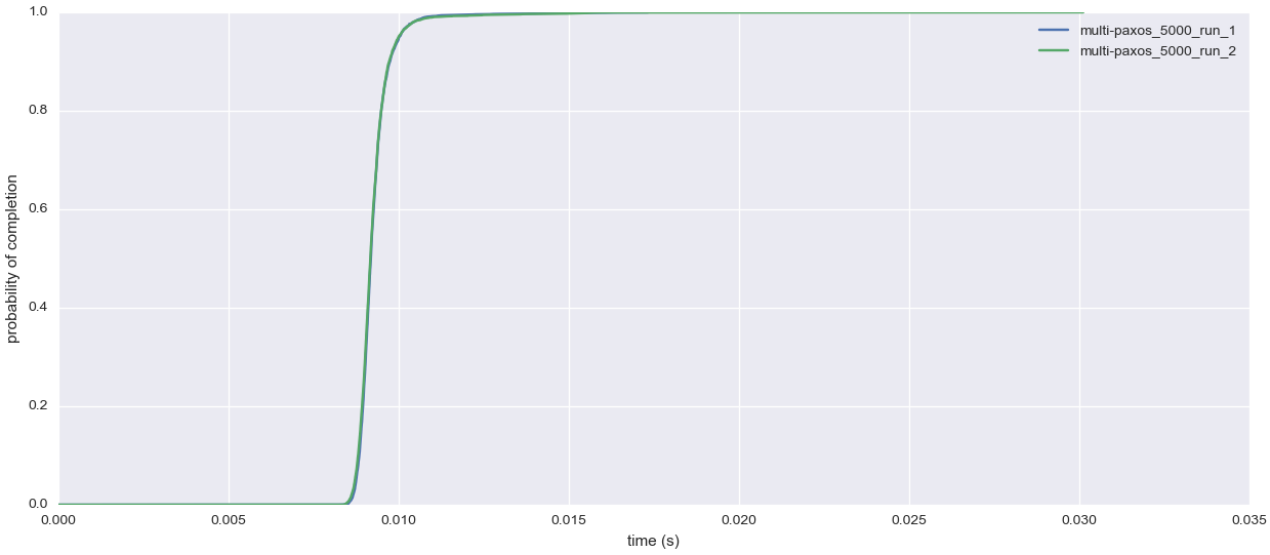

Figure 4.5: Multi-Paxos: 5000 run.

4.2 Choosing the cluster size

Another parameter to be fixed in advance has been the number of nodes composing the cluster. An upper bound of 8 has been set due to Compute Engine that doesn't allow free-trial users to create more instances at the same time in the same region. Therefore, the value 5 has been chosen because it is reasonable enough for the majority of real distributed systems. Moreover, tests have been made to ensure that the performances on 5 nodes are not that far from the ones obtained on different size of the clusters. This has been investigated also for Multi-paxos, leading to very similar results.

4.3 Test suite description

This section describes in depth each test in the suite. They can be divided into the following different main areas: gold standard, symmetric scenarios, asymmetric scenarios, Raft vs Paxos and Raft vs Raft.

4.3.1 Gold standard

This represents the baseline performance metric used as a guide to understand all the other results, providing insights on what to expect from every implementation. As explained before, these tests do not involve any change in the network layer.

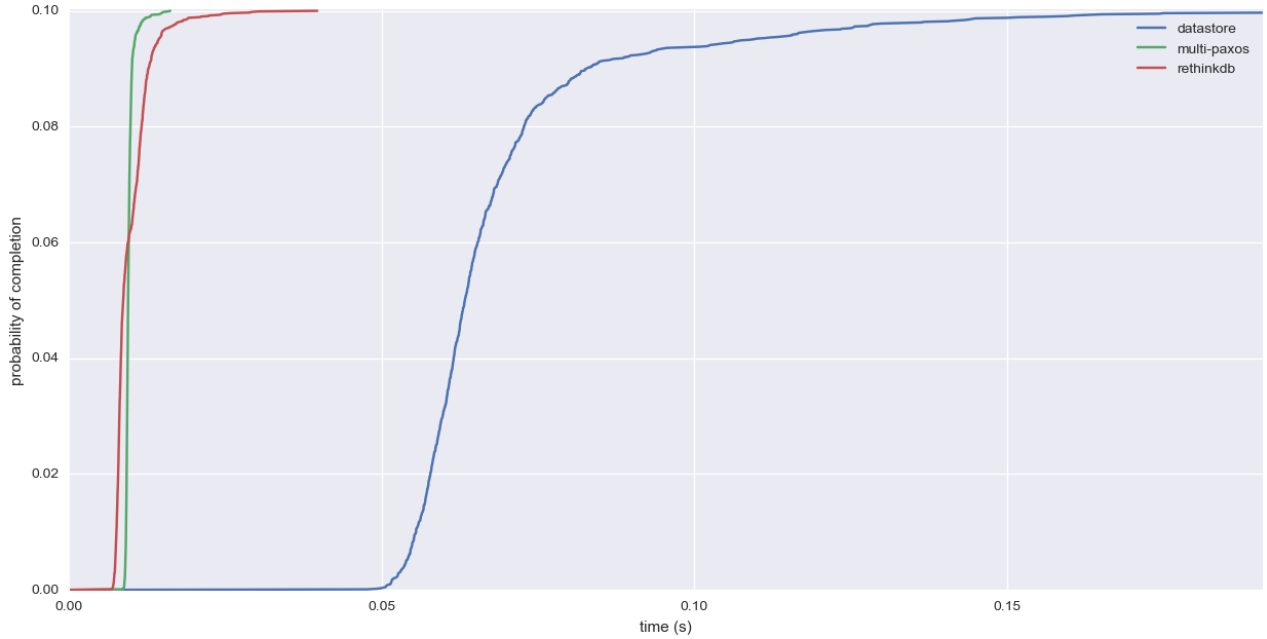

Figure 4.6: Datastore vs Multi-Paxos vs RethinkDB: cumulative density function comparison.

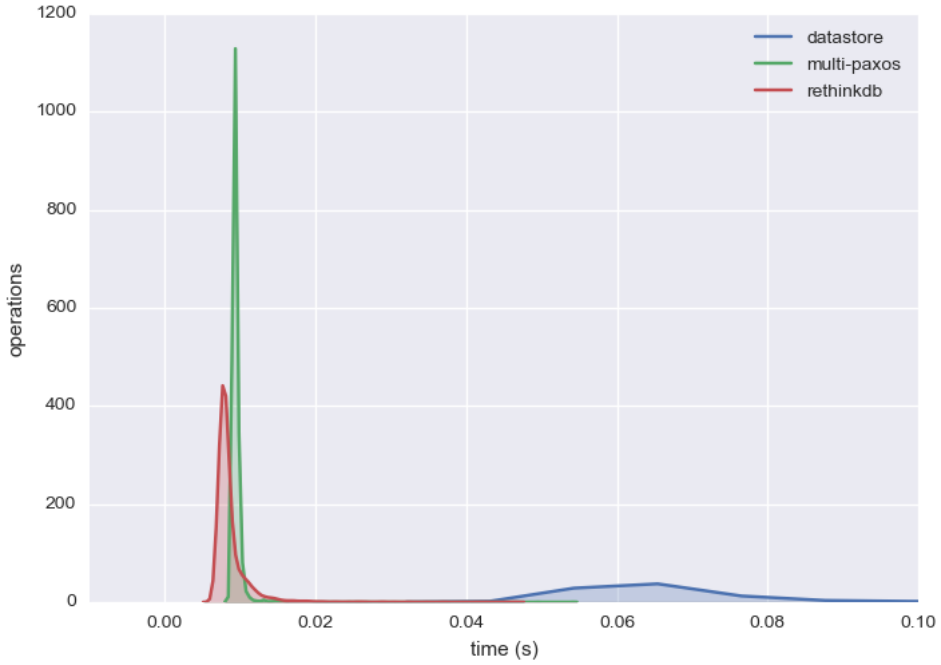

Figure 4.7: Datastore vs Multi-Paxos vs RethinkDB: probability density function comparison.

The figures above show both the CDFs and the distributions functions of the chosen implementations. It is possible to notice how Datastore performs significantly worse than the other two algorithms and has the largest variance in the execution time. These results may be due to the fact that Datastore is not customizable by the user at will, implying that it is not possible neither to know the number of nodes the physical cluster is composed of nor to control the network traffic between them. Its execution time is approximately four time slower than the others, but it's likely that the values stored on Datastore are replicated across Google data centers around the world on a huge amount of nodes.

A similar experiment has been executed in the pseudo-distributed mode. The results shown in figure 4.8 demonstrates how PySyncObj is significantly slower than every other implementations. Moreover, Multi-paxos seems to be slower than RethinkDB in this test, conversely to what shown in the previous experiment in the cloud. This is probably caused by the CPU-intensive nature of the implementation, whose effect is mitigated when nodes resides on physically different machines. On the contrary,

on the local cluster all the nodes compete for the same CPU, thus mutually slowing themselves.

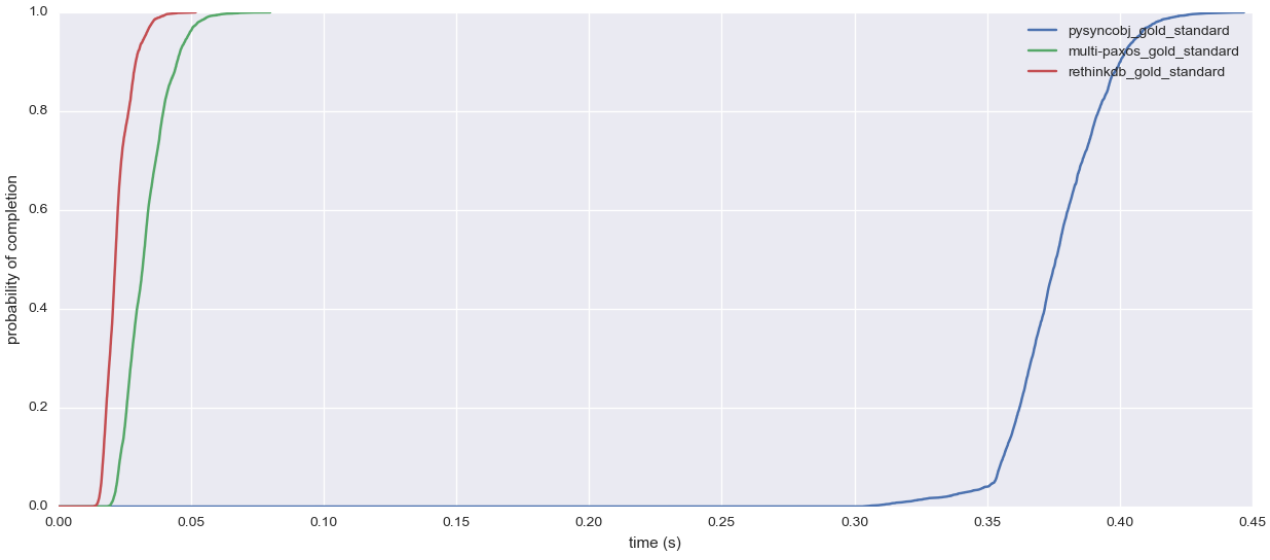

Figure 4.8: PySyncObj vs Multi-Paxos vs RethinkDB: cumulative density function comparison.

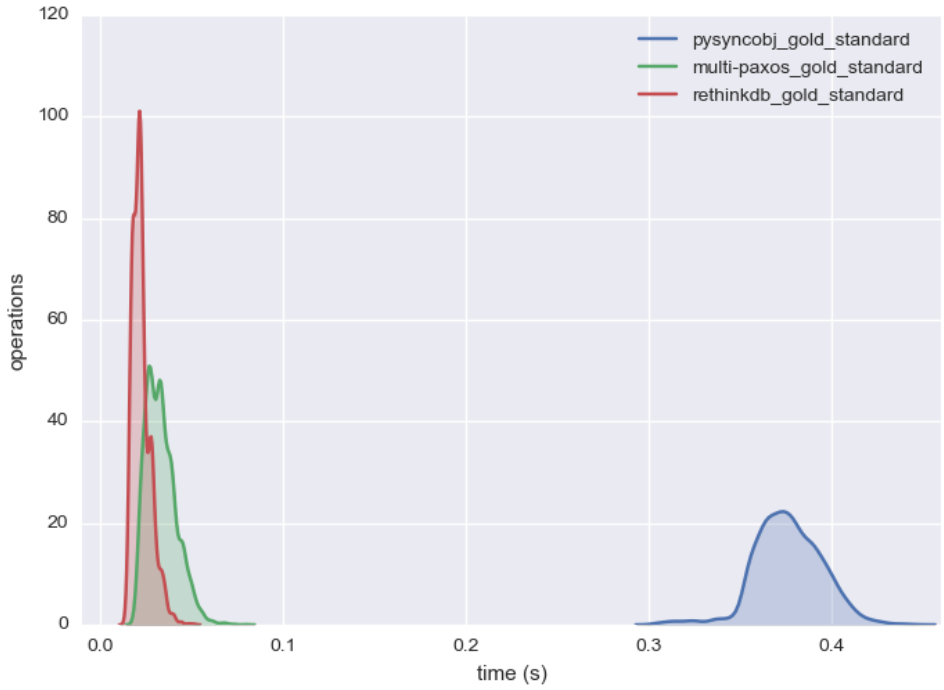

Figure 4.9: PySyncObj vs Multi-Paxos vs RethinkDB: probability density function comparison.

4.3.2 Symmetric scenarios

These tests shows how the system performance degrades while increasing problems on the network in a homogeneous manner. They have been specifically crafted to give the consensus algorithm hard times in replicating data between the nodes. Every distributed algorithm relying upon message passing for communication should be affected by these changes. The completed tests are the following:

- packet delay: every node in the cluster transmits with a certain delay its packets, both ingoing and outgoing. The tested delay are 10 ms, 15 ms and 20 ms.

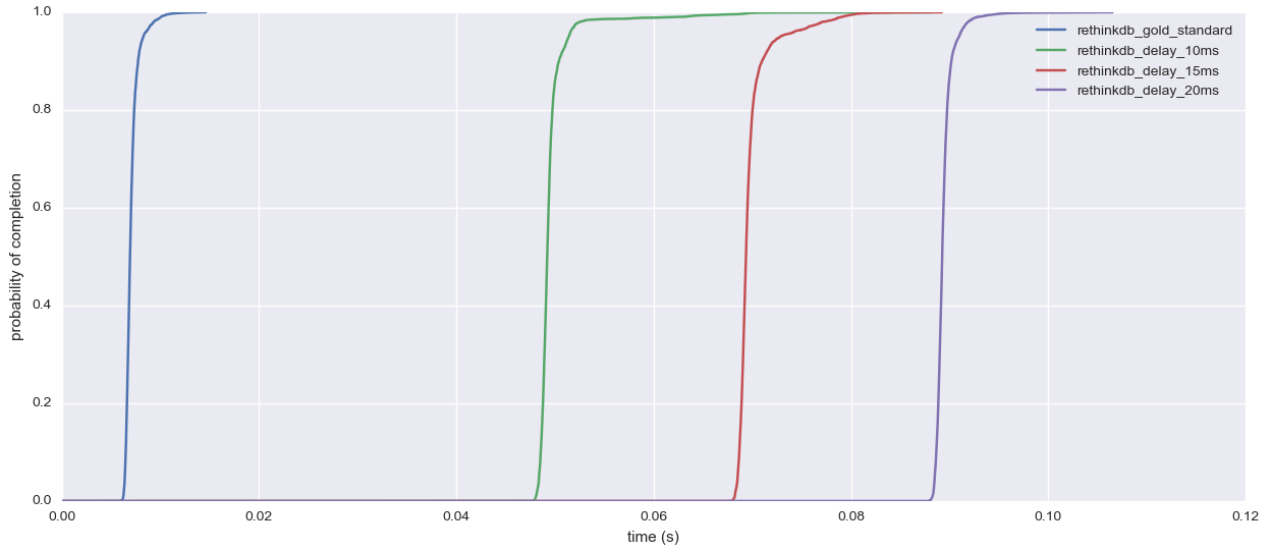

Figure 4.10: RethinkDB: cumulative density function with delay.

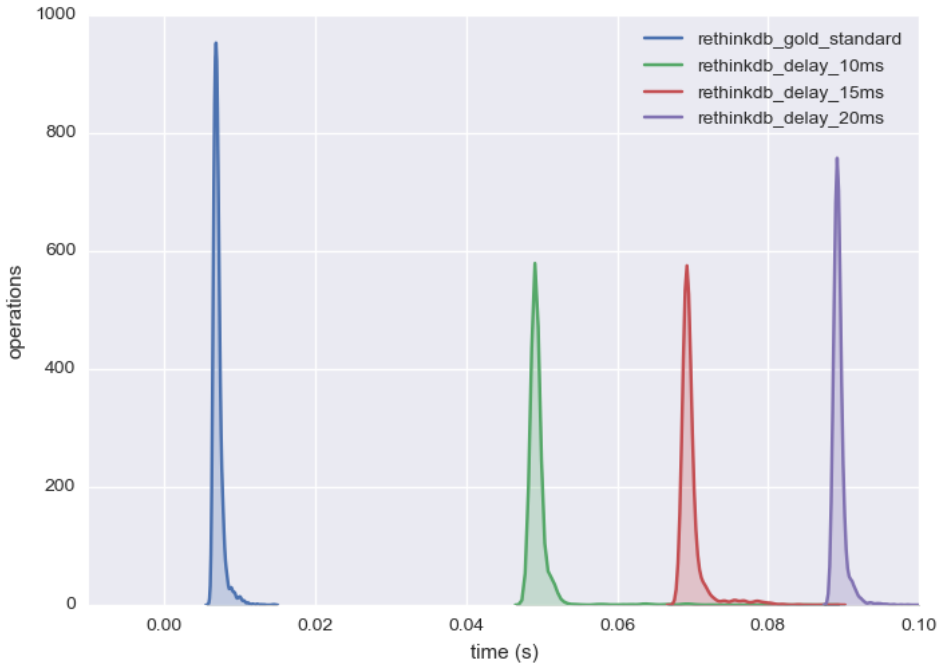

Figure 4.11: RethinkDB: probability density function with delay.

- packet loss: every node in the cluster loses a certain percentage of packets, both ingoing and outgoing. The tested percentages are 2%, 4% and 6% with a uniform distribution.

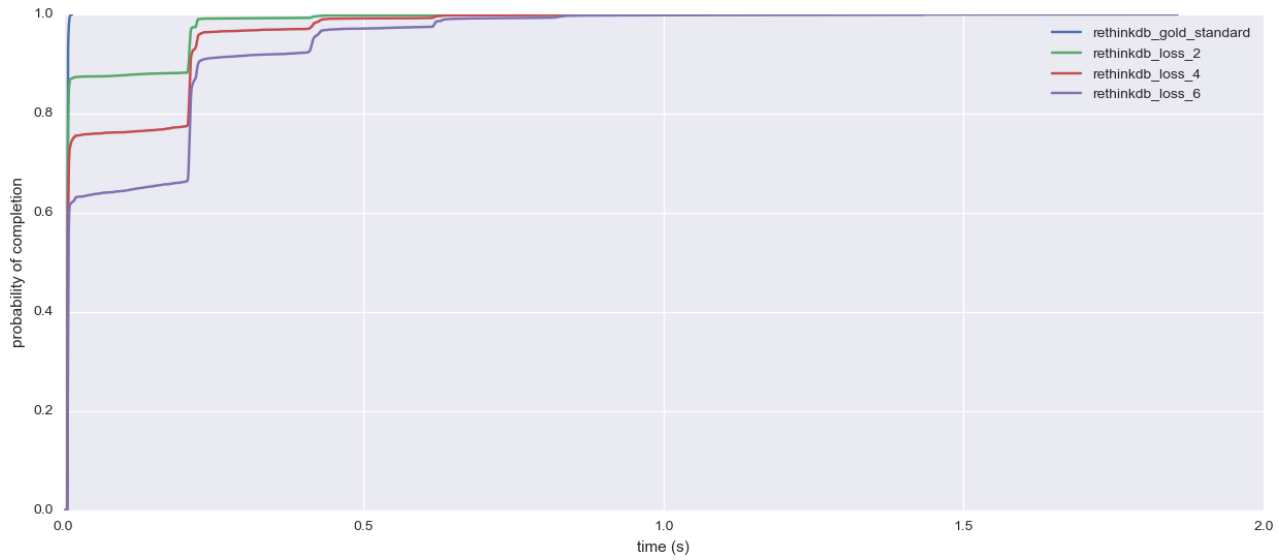

Figure 4.12: RethinkDB: cumulative density function with loss.

- packet corruption: every node corrupts a certain amount of packets, both ingoing and outgoing. The tested percentages are 2%, 4% and 6% with a uniform distribution.

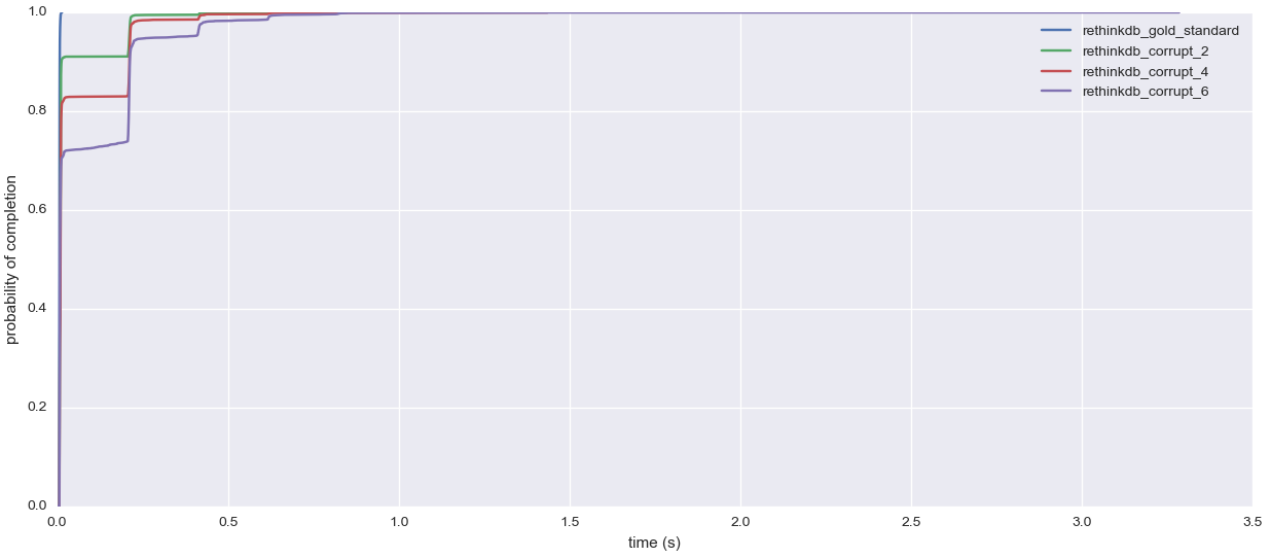

Figure 4.13: RethinkDB: cumulative density function with packet corruption.

- packet inversion: every node randomly reorders 50% of the packets, both ingoing and outgoing and sends the other half with a certain delay. The chosen delays are 10 ms and 20 ms.

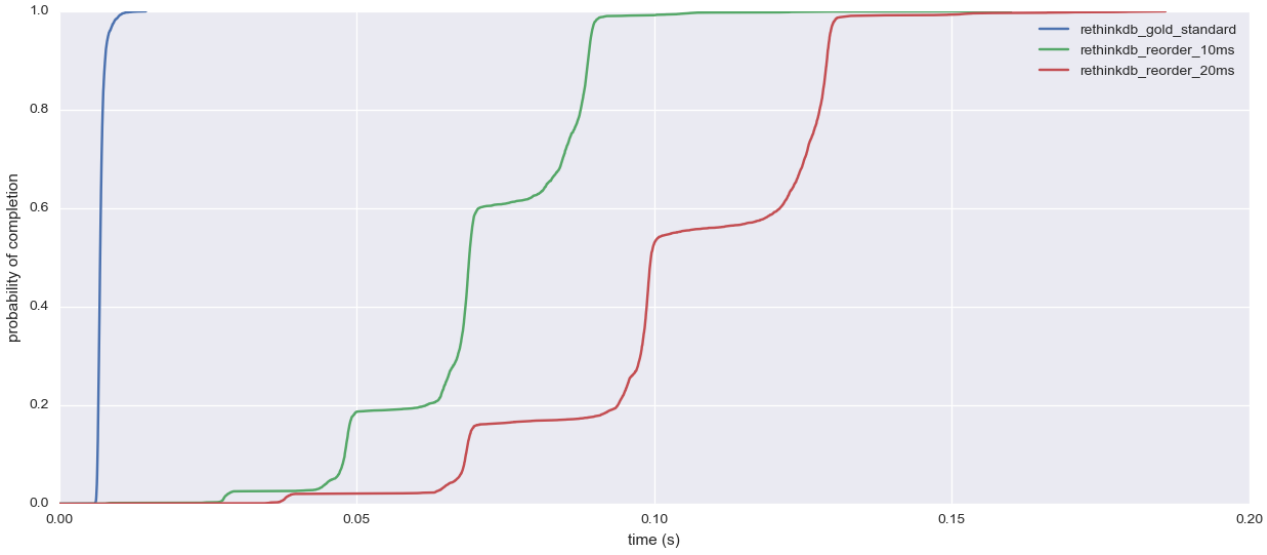

Figure 4.14: RethinkDB: cumulative density function with packet reordering.

In the figures 4.10, 4.12 and 4.13 the trend of the CDFs clearly shows how introducing these network issues influences the time the system needs to reach consensus in a proportional manner. This effect can be seen by looking at the shift of the lines along the x axis towards higher values of elapsed time. The proportionality factor can provide some insights on the number of messages that are exchanged by RethinkDB to reach consensus. As it can be seen in the figure 4.11, with a 10 ms delay the consensus is reached on average in 50 ms. By increasing the delay of only 5 ms, the consensus time increases to 70 ms. The average number of sequential messages exchanged by RethinkDB can be thus computed with the following formula:

$$0.50 - 0.10 * n = 0.70 - 0.15 * n$$
$$n = 4$$

The packets loss graph shows different information. The first gap underlines the effect of one packet loss on the consensus algorithm. The height of each gap represents the probability of not losing more than one packet and it represents an index of the minimum number of messages (not necessarily in sequence) that the underlying algorithm needs to exchange to reach consensus. In fact, increasing loss percentages decreases the probability of reaching consensus (red and purple lines). The second gap, similarly, represents the loss of two packets and so do the successive ones. The distance on the x axis between two gaps represents the time that RethinkDB takes to recover from a packet loss and it is constant on each gap ($\sim 0.2s$).
A similar behaviour can be seen on the packet corruption graph. The main difference from the packet loss is that corruption seems to affect less the protocol, since consensus probabilities are slightly higher. This trend can be due to the fact that one bit corruption has a non-zero probability of leaving the TCP packet intact from an application level point of view.
The reordering graph, on the other hand, shows a different behaviour which seems to be the combination of both packets loss and delay.

4.3.3 Asymmetric scenarios

These tests show how the system performance degrades when the network faces problems on a specific subset of nodes. Adversarial network conditions have been applied only to some nodes to model new network situations, common in distributed systems. The tests that have been completed are the following:

- faulty network cards: 1 random bit of ingoing and outgoing packets gets corrupted with a probability of 0.1. One at a time, every node of the cluster suffers this condition.

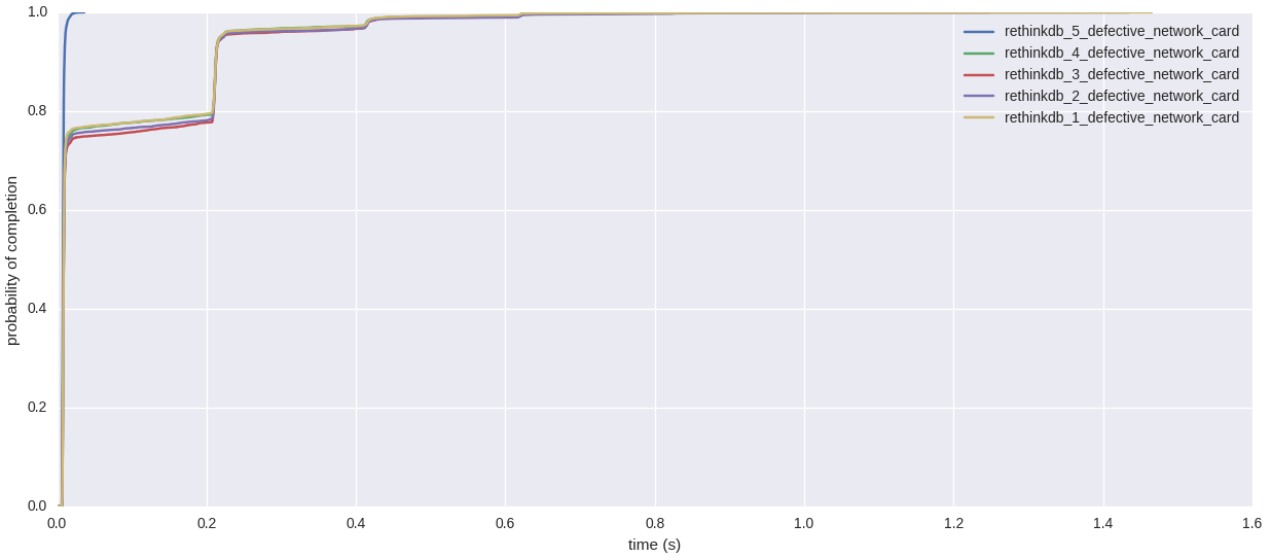

Figure 4.15: RethinkDB: cumulative density function with broken network cards.

- partitions: some links that connect a group of the nodes in the cluster to the others are slowed down. This situation simulate a distributed system whose nodes are logically partitioned into two different regions. In this way the intra-partition communication is slow, but, at the same time, the inter-partition communication is left unchanged.

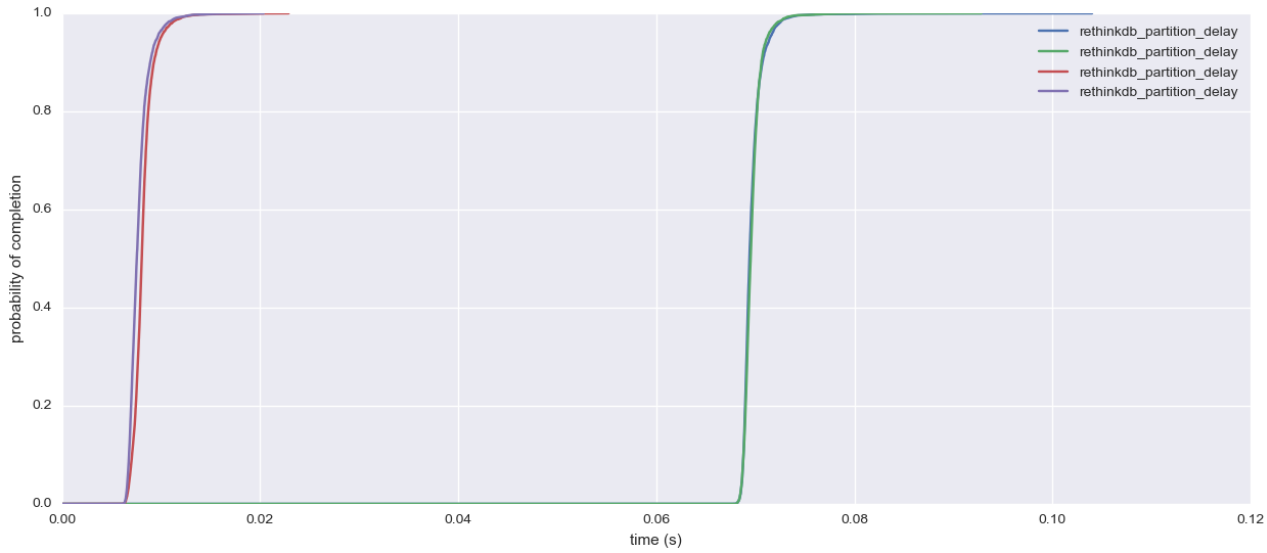

Figure 4.16: RethinkDB: cumulative density function with partitions.

The results in figure 4.15 shows two different behaviours. If the node having the problem is one which is not necessarily involved in the majority to reach consensus, the consequences are negligible (blue curve). On the other hand, if the faulty network card is the one of a node which is actually needed to obtain consensus, the performances deteriorate (all the other lines).
figure 4.16 explains how the Raft algorithm responds to a logically-partitioned cluster with groups made of 2 and 3 nodes respectively. More in detail:

- if the leader and a majority of the nodes are together, the algorithm is not delayed at all, since nobody in this partition needs to communicate with another node through a slow link. This situation is represented by the purple and the red lines on the left side.

- if the leader needs to contact any node afflicted by a slow link, it has to wait for him to respond and the overall performances are clearly worse (green and blue lines on the right).

4.3.4 Raft vs Paxos

This test is thought to compare the two consensus algorithms and show which of them suffers more packets delay, which is probably the most frequent issues on real systems.

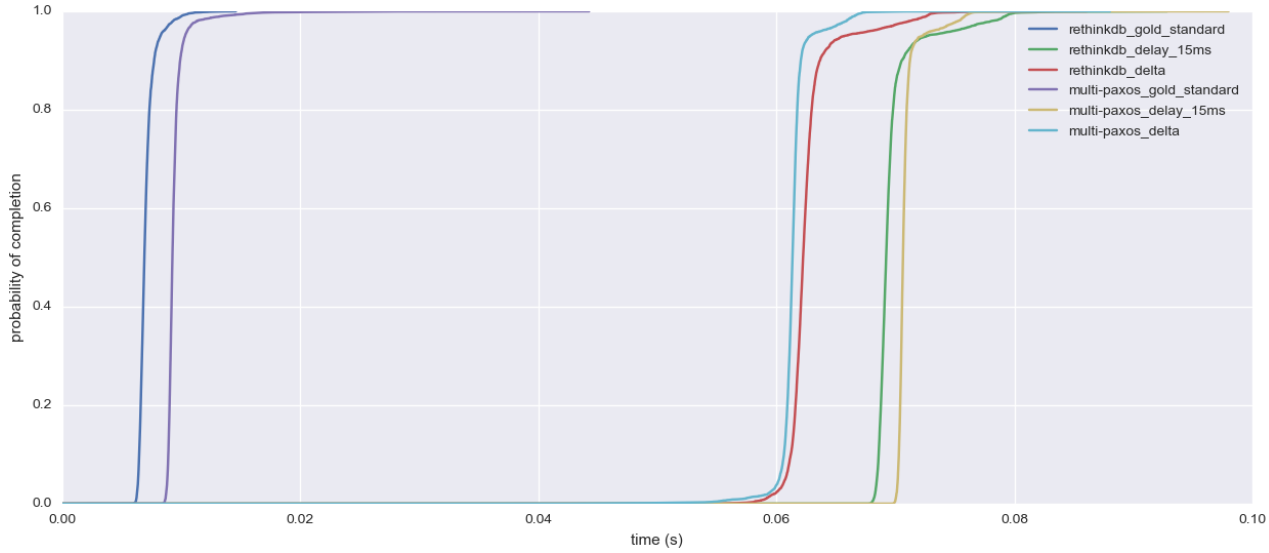

Figure 4.17: RethinkDB vs Multi-Paxos: cumulative density function comparison with delay.

The figure above illustrates how Multi-paxos performs slightly better than RethinkDB if the delay is quite small (15 ms). This can be easily noticed by looking at the two curves in the middle, showing the difference between the gold standards and the delayed executions, where Multi-paxos (light blue) is on the left with respect to RethinkDB (red).

4.3.5 Raft vs Raft

Different implementations relying upon the same consensus algorithms are compared in order to demonstrate how RethinkDB clearly outperforms PySyncObj. In this way it is possible to see how a commercial product is orders of magnitude better than its counterpart from any point of view.

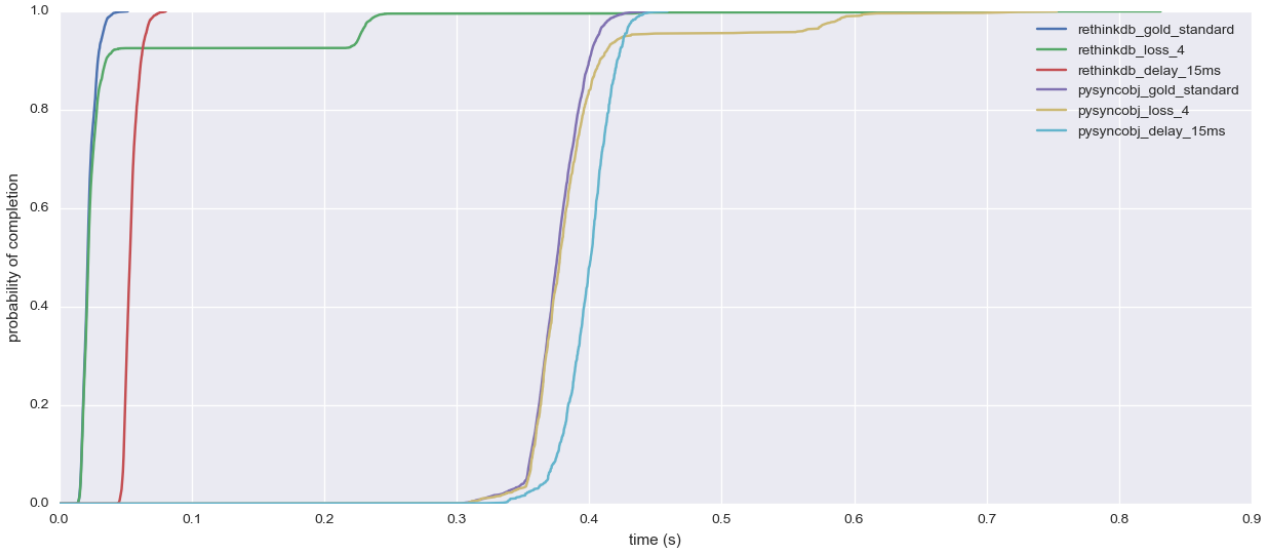

Figure 4.18: RethinkDB vs PySyncObj: cumulative density function comparison with delay and loss.

Chapter 5

Conclusion

Distributed algorithms are typically tested validating all the possible interactions between a set of nodes. This task is usually performed simulating different processes on local machines or by using different machines in a network. This phase typically requires many efforts, both for the wide range of scenarios produced by interleaving executions and for the difficulties in debugging remote machines. The testing approach proposed in this work is different since it directly manipulates the network underlying the system rather than acting on individual nodes. This solution allows the tester to model many different situations with little effort. The black-box approach suits this paradigm and such kind of tests can be easily performed using the developed platform. Even if network modifications affect the lower level message exchange, many useful insights on the upper application level logic can be inferred. For the purpose of this work, the platform has been used to test different implementations of Raft and Paxos consensus algorithms. The tests performed on Multi-paxos show it favours a distributed environment rather than a pseudo-distributed one. This is probably due to the CPU-intensive nature of this particular implementation. Furthermore, Multi-paxos is not performing failovers in an efficient way after a period of serious problems on the network.

RethinkDB seems to be the overall winner, even though other implementations reach slightly better results in some tests. This may be due to internal optimizations, but investigating them is out of the scope of this work. The product proved to be very well implemented and very robust, being capable of performing disaster recovery in a small amount of time. Moreover, RethinkDB is highly customizable and this flexibility well

suits different users' requirements.
Finally, as expected, commercial products turned out to be much better than their counterparts, especially from the safety point of view.