RethinkDB - Jepsen Tests

Ayush Jain (aj2672)
Computer Engineering, Columbia University
Prakhar Srivastav (ps2894)
Computer Science, Columbia University

Abstract—**Jepsen is a Clojure library that provides functionality to setup a distributed system, run a bunch of operations against that system and verify that the history of those operations make sense. This history can be used to arrive at a consistency model for the system.**

This paper discusses the use of Jepsen to evaluate consistency semantics of RethinkDB - an upcoming real-time open-source database. RethinkDB claims to provide linearizablity over atomic operations. We tried to understand how strictly does RethinkDB adhere to this claim and this paper represents a discussion of our finding.

I. Introduction

RethinkDB is a scalable document-oriented database which came out of a need to make building real-time applications easier. Apart from being one of the few NoSQL databases that allow joins, RethinkDB is a distributed database that is easy to scale. With multiple shards and replicas, a RethinkDB clusters provides high availability and robust fault-tolerance. In this section, we will first go over consistency semantics in general. We will then look at the architecture of RethinkDB and its consistency guarantees.

A. Consistency Semantics

In any concurrent system that has multiple actors working on a shared memory, one of the key concerns is how do writes and reads behave. More explicitly, in the absence of a global real-time clock, how does the ordering of these operations affect the outcome of the system. Hence consistency is the idea of assigning meaning to concurrent reads and writes on shared, possibly replicated, state.

For distributed systems we define the term consistency model. We say that given some rules which relate the operations and state, the history of operations in the system and their result should always follow those rules. We call these rules a consistency model. These rules give us a basic direction of what the value returned in case of every write should be. These rules can be anything. These could be very strict based on our intuition above, or these could be very weak. Like a no rule statement, where we say that every operation is permitted is also a consistency model. The principle is that once a system defines these rules, they should be adhered to, no matter what.

The strongest consistency model defined is the linearizability model. We call this model strongest because it maps most closely to our intuitive model. Effectively, linearizability enforces that once a write completes, all later reads should return the value of that write or the value of a later write. Once a read returns a particular value, all later reads should return that value or the value of a later write.

We can use the atomic constraint of linearizability to mutate state safely. We can define an operation like compare-and-set, in which we set the value of a register to a new value if, and only if, the register currently has some other value. Linearizability guarantees us the safe interleaving of changes and that those changes will be visible to other participants after the operation. In essence, this model prohibits stale reads. Each read will see some current state between invocation and completion; but not a state prior to the read. It also prohibits non-monotonic reads in which one reads a new value, then an old one.

Linearizability is important for distributed systems and databases because of two reasons. First, it is synonymous with the term atomic consistency. Second, it is the C, or consistency, in Gilbert and Lynchs CAP theorem, considered the fundamental of building any distributed system.

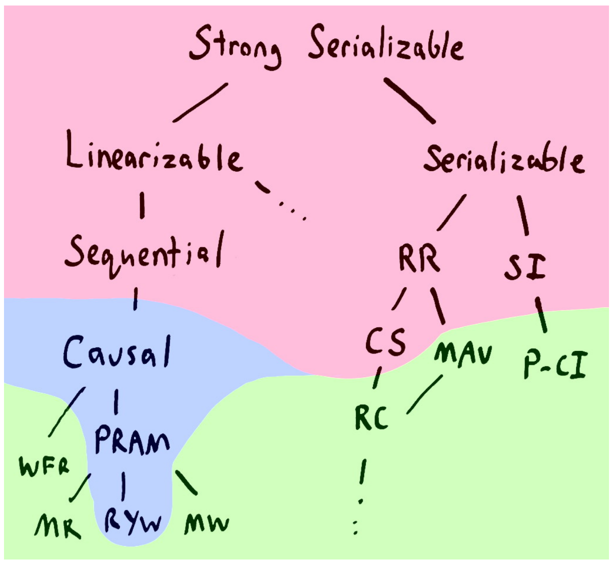


Fig. 1. Consistency Models in order of strength.

B. RethinkDB

RethinkDB is an open source, NoSQL, distributed document-oriented database. It stores JSON documents with dynamic schemas. RethinkDB identifies itself as a CP (consistent and partition tolerant) system. What this means is that in times of a network partition, a RethinkDB cluster chooses to maintain data consistency over unavailability. In its performance, it is more similar to MongoDB, which also seeks to maintain consistency, while different from Riak or Dynamo which define themselves as AP systems.

Figure 2 shows what a RethinkDB cluster looks like in general. The data can be sharded at each table level with variable number of shards for different tables. Each shard can have multiple replicas, out of which one is selected to be the primary. All reads and writes to any key in a given shard always get routed to its respective primary, where theyre ordered and evaluated. This architecture helps RethinkDB achieve its consistency goals.

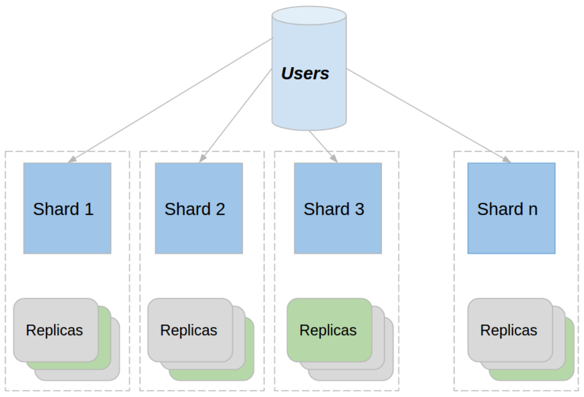


Fig. 2. RethinkDB cluster

If the primary replica for a table fails, as long as more than half of the voting replicas for each shard remain available, one of those voting replicas will be arbitrarily selected as the new primary. There will be a brief period of unavailability, but no data will be lost. If the primary replica specified in a tables configuration comes back online after a failure, it will return to being the primary.

If there is a network partition, the behavior of the system from any given clients perspective depends on which side of the netsplit that client is on. If the client is on the same side of the netsplit as the majority of voting replicas for the shard, it will continue operating without any problems. However, if the client is on the opposite side of the netsplit, the clients up-to-date queries and write queries will encounter a failure of availability.

Apart from the default state performance, RethinkDB also provides various configurations to tune its availability and consistency performance according to particular requirements. These configurations are based on the following three parameters.

- Write acknowledgements are set per table with the write_acks setting, either using the config command or by writing to the table_config system table. The default is **majority**, meaning writes will be acknowledged when a majority of (voting) replicas have confirmed their writes. The other possible option is single, meaning writes will be acknowledged when a **single** replica acknowledges it.

- Durability is set per table with the durability setting, again using either reconfigure or writing to the table_config system table. In **hard** durability mode, writes are committed to disk before acknowledgements are sent; in **soft** mode, writes are acknowledged immediately after being stored in memory. The soft mode is faster but slightly less resilient to failure. The default is hard.

- Read mode is set per query via an optional argument, read_mode (or readMode), to table. It has three possible values: **single** returns values that are in memory (but not necessarily written to disk) on the primary replica. This is the default. **majority** will only return values that are safely committed on disk on a majority of replicas. This requires sending a message to every replica on each read, so it is the slowest but most consistent. Lastly, **outdated** will return values that are in memory on an arbitrarily-selected replica. This is the fastest but least consistent.

According to the above settings based on the three parameters, various levels of availability and consistency guarantees can be achieved. However, RethinkDB claims that, we can achieve linearizability of individual atomic operations on individual documents with the following settings:

- write_acks: majority
- durability: hard
- read_mode: majority

Since, the write and reads are acknowledged from a majority of nodes and hard durability ensures writes in both memory and disk before a confirmation, the RethinkDB promises seem valid. The goal of our project is to validate these guarantees not in favorable conditions or under high throughput but under stressful conditions of system and network failures.

II. Test Setup

Our test primarily consists of setting up of a RethinkDB cluster of five nodes, creating a table with appropriate write settings and then performing a bunch of read, write and CAS operations. We then record this history of operations and try to arrive at a linearity model from the same. However, this is harder than it sounds.

This calls for high computation and memory requirements and easy setup and tear down of our system on each test run apart from an appropriate framework to run operations, induce system and network partitions and recover from them. It should also provide for sufficient logging to note down the history of events and then make sense of it in order to arrive at any consistency model. Following are the different components of our test framework along with their configurations. We have discussed the last two in greater detail in the following section.

1) **Docker** - 6 Docker containers running Debian Jessie operating system. 1 parent container hosting 5 inner containers as RethinkDB nodes.
2) **Amazon Web Services** - 1 EC2 Instance - 160 GB Memory and 40 vCPUs
3) **RethinkDB** - Version Number: 2.2.0
4) **Jepsen** - Version Number: 0.0.7-SNAPSHOT
5) **Knossos** - Version Number: 0.2.4-SNAPSHOT

III. Jepsen

Jepsen is a clojure library that provides functionality to setup a distributed system, run a bunch of operations against that system and verify that the history of those operations make sense.

A Jepsen test runs as a Clojure program on a control node. That program uses SSH to log into a bunch of db nodes, where it sets up the distributed system you're going to test using the test's pluggable os and db.

Once the system is running, the control node spins up a set of logically single-threaded processes, each with its own client for the distributed system. A generator generates new operations for each process to perform. Processes then apply those operations to the system using their clients. The start and end of each operation is recorded in a history. While performing operations, a special nemesis process introduces faults into the system–also scheduled by the generator.

Finally, the DB and OS are torn down. Jepsen uses a checker to analyze the test's history for correctness, and to generate reports, graphs, etc. The test, history, analysis, and any supplementary results are written to the filesystem under $store/<test-name>/<date>/$ for later review. Symlinks to the latest results are maintained at each level for convenience.

Jepsen tests are made of three fundamental parts

- **Generator** - A stateful object that generates operations for processes to run.
- **Client** - Runs operations on the distributed system. Processes call clients with operations to be performed.
- **Checker** - Validates whether a history of operations is correct with respect to some model.

A. Generators

A generator is a single stateful object that yields :invoke operations to processes. Each process asks the generator for an operation, applies it to the client, then comes back for another operation. Once a generator returns nil, it's empty and a process won't request any more from it.

The job of the generator is to generate a bunch of randomly ordered operations for the processes to execute. The high-level idea is to mix, randomly permute and stagger these operations over a time limit so that these can be executed by the processes.

In our tests, we have used three operations namely - **write**, **read**, **cas**. These operations correspond to the **atomic** operations over which RethinkDB guarantees linearizability over a single row.

For configuring the generation of these operations, the Jepsen library provides a bunch

```
; Generators
(defn w   [_ _] {:type :invoke, :f :write, :value (rand-int 5)})
(defn r   [_ _] {:type :invoke, :f :read})
(defn cas [_ _] {:type :invoke, :f :cas, :value [(rand-int 5) (rand-int 5)]})
```

Fig. 3.   Generators

of helper functions to tweak the sequence and ordering. Our tests make use of a few of these functions namely -

1) gen/stagger - introduces a time delay between operations
2) gen/mix - returns a mix of operations randomly chosen for a list
3) gen/limit - the time limit for which the operations are selected
4) r,w,cas - read, write and cas operations performed on the database

B. Client

Clients take :invoke operations, apply them to the system being tested (e.g. by making a network call) and return a corresponding completion operation with type :ok, :fail, or :info).

In our tests, the client is responsible for interfacing with the database servers. In other words, all db commands such as create, update etc are executed by the client. Internally each of the processes call these client for db related tasks.

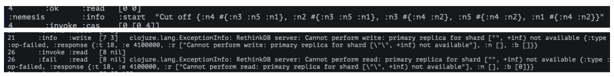
Our tests first begin by bootstrapping the database and the tables with the appropriate write-mode and read-mode setting. This is made configurable so that the testing for various combinations of read and write settings can be made easy. More importantly, there is also a teardown! function that deals with cleaning up after the test is complete to ensure that each test result starts with a clean slate.

```
(case (:f op)
  :read (assoc op
               :type  :ok
               :value (independent/tuple id
                        (query/run (term :DEFAULT
                                     [(query/get-field row "val") nil])
                               (:conn this))))
  :write (do (query/run (query/insert (query/table (query/db db) table)
                               {:id id, :val value}
                               {"conflict" "update"})
                    (:conn this))
             (assoc op :type :ok))
  :cas (let [[value value'] value
             res (query/run
                   (query/update
                     row
                     (query/fn [row]
                       (query/branch
                         (query/eq (query/get-field row "val") value)
                         {:val value'}
                         (query/error "abort"))))
                   (:conn this))]
         (assoc op :type (if (and (= (:errors res) 0)
                                  (= (:replaced res) 1))
                           :ok
                           :fail))))))
```

Fig. 4.   Client Operations - Read, Write and CAS

C. Nemesis

Nemesis is a part of the Jepsen library that is responsible for causing network partitions in the distributed system that is being tested. It works by removing IPtables on hosts and modifying the /etc/hosts file so that hosts are unable to communicate to each other.



```
4        :ok     :read   [0 0]
:nemesis         :info   :start  "Cut off {:n4 #{:n3 :n5 :n1}, :n2 #{:n3 :n5 :n1}, :n3 #{:n4 :n2}, :n5 #{:n4 :n2}, :n1 #{:n4 :n2}}"
4        :invoke :cas    [0 [0 4]]
```

```
21      :info   :write  [7 3]   clojure.lang.ExceptionInfo: RethinkDB server: Cannot perform write: primary replica for shard ["", +inf) not available {:type
:op-failed, :response {:t 18, :e 4100000, :r ["Cannot perform write: primary replica for shard [\"\", +inf) not available"], :n [], :b []}}
26      :invoke :read   [8 nil]
26      :fail   :read   [8 nil] clojure.lang.ExceptionInfo: RethinkDB server: Cannot perform read: primary replica for shard ["", +inf) not available {:type :
op-failed, :response {:t 18, :e 4100000, :r ["Cannot perform read: primary replica for shard [\"\", +inf) not available"], :n [], :b [0]}}
```

Fig. 5.   Nemesis output

In our tests we use nemesis to partition our 5-node network into two halves. Depending on which half the primary replica gets partitioned into and the side of the partition that the client deals with, the client can encounter failed shard errors.

D. Checker

The checker is the part of the library that deals with verifying a history of operations with respect to a model. This functionality is implemented under another library - **knossos**. Knossos is linearizability checker that given a history of operations by a set of clients, and some single-threaded model, attempts to show that the history is not linearizable with respect to that model.

CAS Register

The model that we used in our tests is a simple CAS register. A cas register is an abstract model to mimic database behavior. It supports only three operations

- Read: Read value from register
- Write(x): Write value x to register
- CAS(x,y): If x set y, else raise an exception

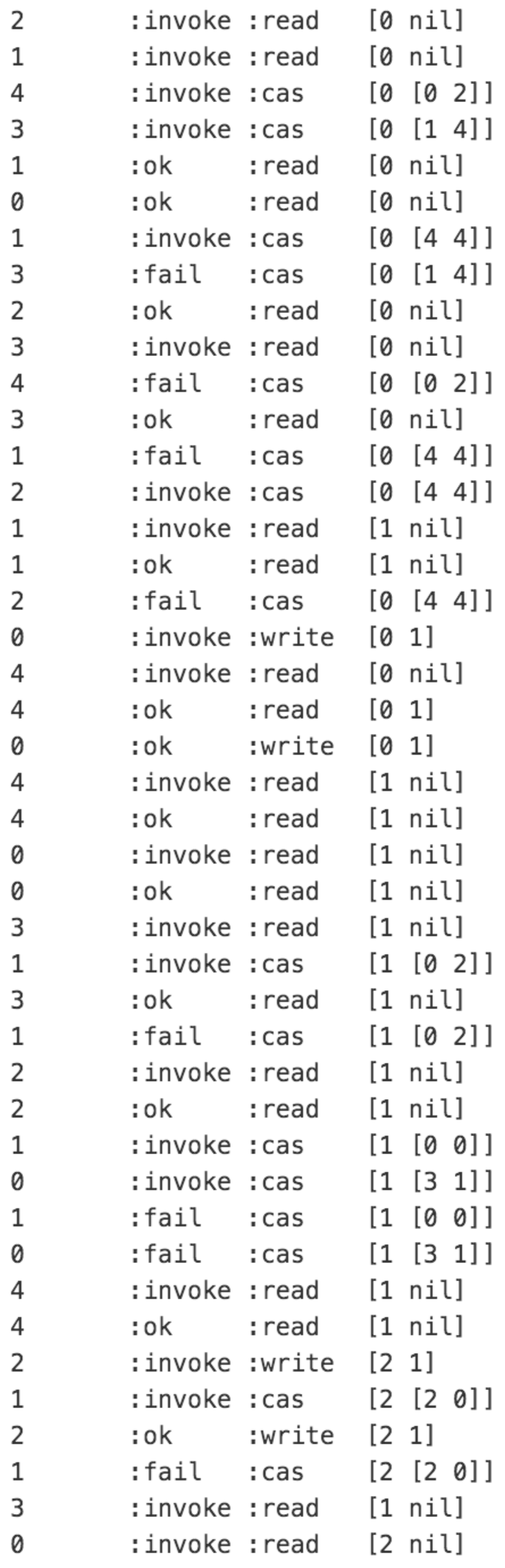
Apart from its simplicity and strong similarity to database, the CASRegister model is used by Jepsen to test similar properties of other databases as well. Hence, we decided to use the CASRegister model in our linearizablity checker.

Validating Linearizablity with Knossos

Linearizability is a guarantee about single operations on single objects. It provides a real-time (i.e., wall-clock) guarantee on the behavior of a set of single operations (often reads and writes) on a single object (e.g., distributed register or data item).

Under linearizability, writes should appear to be instantaneous. Imprecisely, once a write completes, all later reads (where later is defined by wall-clock start time) should return the value of that write or the value of a later write. Once a read returns a particular value, all later reads should return that value or the value of a later write.

Now that we have defined linearizability we can dive deeper into how knossos evaluates whether a given history of operations is linearizable with respect to a given model.



```
2       :invoke :read   [0 nil]
1       :invoke :read   [0 nil]
4       :invoke :cas    [0 [0 2]]
3       :invoke :cas    [0 [1 4]]
1       :ok     :read   [0 nil]
0       :ok     :read   [0 nil]
1       :invoke :cas    [0 [4 4]]
3       :fail   :cas    [0 [1 4]]
2       :ok     :read   [0 nil]
3       :invoke :read   [0 nil]
4       :fail   :cas    [0 [0 2]]
3       :ok     :read   [0 nil]
1       :fail   :cas    [0 [4 4]]
2       :invoke :cas    [0 [4 4]]
1       :invoke :read   [1 nil]
1       :ok     :read   [1 nil]
2       :fail   :cas    [0 [4 4]]
0       :invoke :write  [0 1]
4       :invoke :read   [0 nil]
4       :ok     :read   [0 1]
0       :ok     :write  [0 1]
4       :invoke :read   [1 nil]
4       :ok     :read   [1 nil]
0       :invoke :read   [1 nil]
0       :ok     :read   [1 nil]
3       :invoke :read   [1 nil]
1       :invoke :cas    [1 [0 2]]
3       :ok     :read   [1 nil]
1       :fail   :cas    [1 [0 2]]
2       :invoke :read   [1 nil]
2       :ok     :read   [1 nil]
1       :invoke :cas    [1 [0 0]]
0       :invoke :cas    [1 [3 1]]
1       :fail   :cas    [1 [0 0]]
0       :fail   :cas    [1 [3 1]]
4       :invoke :read   [1 nil]
4       :ok     :read   [1 nil]
2       :invoke :write  [2 1]
1       :invoke :cas    [2 [2 0]]
2       :ok     :write  [2 1]
1       :fail   :cas    [2 [2 0]]
3       :invoke :read   [1 nil]
0       :invoke :read   [2 nil]
```

Fig. 6. History of operations

The image on the right shows one such snapshot of the history of operations. The first column corresponds to the process id, which is an integer between 0 and value of concurrency setting. In our configuration, this was set to 5. The second column is the result of operation that was carried out, followed by the type of the operation (one of :read, :write or :cas).The last column indicates the input that is passed to each of these operations.

Given such a history of operations, the problem can be more precisely described as so -

Problem: Taking a history with pairs of (invoke, ok) operations, and finding an equivalent history of operations which is consistent with the model. This equivalent single-threaded history is called a **linearization**; a system is **linearizable** if at least one such history exists.

Internally, knossos uses the techinque of **automatic linearization** to find a concurrent history.

Automatic Linearization: *A straightforward way to automatically check whether a concurrent history has a corresponding linearization is to simply try all possible permutations of the concurrent history until we either find a linearization and stop, or fail to find one and report that the concurrent history is not linearizable.*

Despite its inherent complexity costs, this method is used by Knossos for checking concurrent histories of small length (e.g. less than 20). In practice, the space used for concurrent algorithms is typically small because incorrect algorithms often exhibit an incorrect concurrent history which is almost sequential.

IV. Test Results

For our tests, we chose two different sets of cofigurations of RethinkDB:

1) read_mode: **single**, durability: **hard**, write_mode: **single**
2) read_mode: **majority**, durability: **hard**, write_mode: **majority**

To reiterate, the first configuration means that all reads will be done from the primary replica, data is written to permanent storage of nodes before sending a commit but it will be written only to the primary replica, before it is said to be committed.

The second configuration on the other hand means that all reads will be returned from the majority of voting replicas, data is written to permanent storage of nodes before sending a commit and it will be written to a majority of replicas, before it is confirmed as committed.

read_mode	**write_mode**	**linearizible?**
single	single	false
majority	majority	false

Fig. 7. Summary of results

As the above table suggests our tests show non-linearizable output for both the configurations. While, this is expected for our first set of configuration, it is not in line for our second set and does not match the guarantees made by RethinkDB documentation. RethinkDB promises linearizability for atomic operations on a single document.

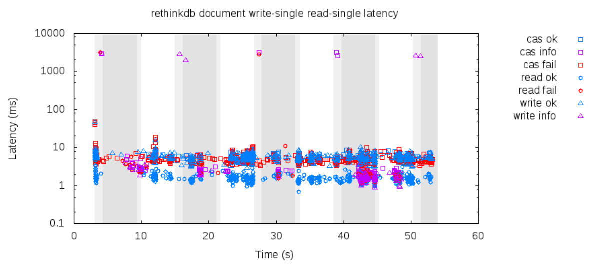


Fig. 8. Obesrved Latencies for Single-Hard-Single Configuration

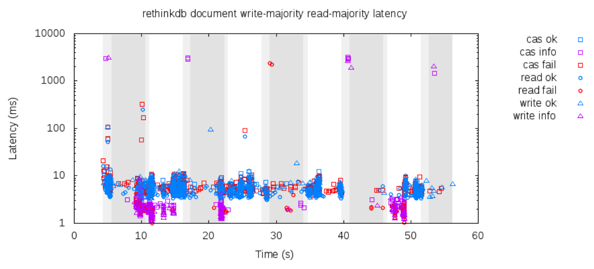


Fig. 9. Obesrved Latencies for Majority-Hard-Majority Configuration

The figures 8 and 9 show the latencies graph obtained for our two set of configurations. It is easy to see that read latencies are consistently lower than that of write latencies. While the blue points show successful operations, red show failures and purple are the indeterminate which also essentially depicts failures. The greyed out regions of the graphs show periods of network partition and hence show considerably higher failures as compared to the white ones.

Between the two graphs, it can also be seen that read and write latencies are lower for single-hard-single configuration than what is seen for majority-hard-majority. since, majority-hard-majority achieves a quorum of majority of its voting replicas before returning a successful result, this is expected.

One other observation to note is that we see a considerably higher number of failures during times of network partitions for the second configuration as compared to what is seen for the first configuration. This is also expected as the cluster seeks to obtain a quorum of majority replicas in the second configurations which may or may not always be possible.

V. Conclusion

This paper describes our project which uses the Jepsen library for devising tests in order to validate consistency semantics (linearizablity) for RethinkDB.

In summary we've found that RethinkDB does not hold up to its stated promise of being strongly consistent i.e. linearizable for atomic single-row operations at the stated configuration setting - read_mode: majority and write_mode: majority. Lastly, for a weaker configuration setting - read_mode: single and write_mode: single our tests showed that the atomic operations are again not linearizable. These results, however, are expected.

VI. Future Work

Although our tests do validate the linearizability of the two specific configurations of a RethinkDB cluster there is scope for substantial future work. In particular, we would like to work on the following:

- Isolate a smaller history which could be used to validate linearizability manually. Currently, the subset of operations outputted by knossos is too large to validate manually by hand.
- Validate more configurations and map them to other consistency semantics. Apart from single/single and majority/majority there are various other combinations of configurations possible with RethinkDB, e.g. outdated. It would be interesting to map various combinations into other weaker semantics e.g. read committed, eventual consistency etc.
- The RethinkDB documentation talks about a particular procedure for ensuring no split-brains are obtained after a network partition heals. Tests to validate these claims would make an interesting case study that would in turn help in understanding how robust the system is to failures.

- The current tests generate operation inputs at random i.e. read/write/cas operations take input values at random. We would want to evaluate whether generating input in a certain order could help us in culling down the search space of all possible histories.

- Lastly this project does not dive into the reason why the system is **not** linearizable. Looking deeper into the system, such as the leader election protocol or the replication methodology to isolate the causes of achieving linearizablity should be the logical next step.

Acknowledgment

We would like to thank our Prof. Roxana Geambasu for providing us feedback and guidance on our project. Her suggestions regarding evaluating consistency models and narrowing down the scope of the project were invaluable.

We would also like to thank Kyle Kingsbury aka Aphyr, the author of Jepsen and Knossos, in clarifying our doubts regarding the setup of the system.

References

[1] RethinkDB Architecture Documentation

[2] Linearizability versus Serializability by Peter Bailis

[3] Class lectures by Prof Roxana Geambasu

[4] Highly Available Transactions: Virtues and Limitations, Peter Bailis, Aaron Davidson, Alan Fekete , Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica

[5] Consistency Models by Kyle Kingsbury

[6] MongoDB Analysis by Kyle Kingsbury

[7] Linearizablity with Knossos by Kyle Kingsbury

[8] Jepsen Codebase by Kyle Kingsbury