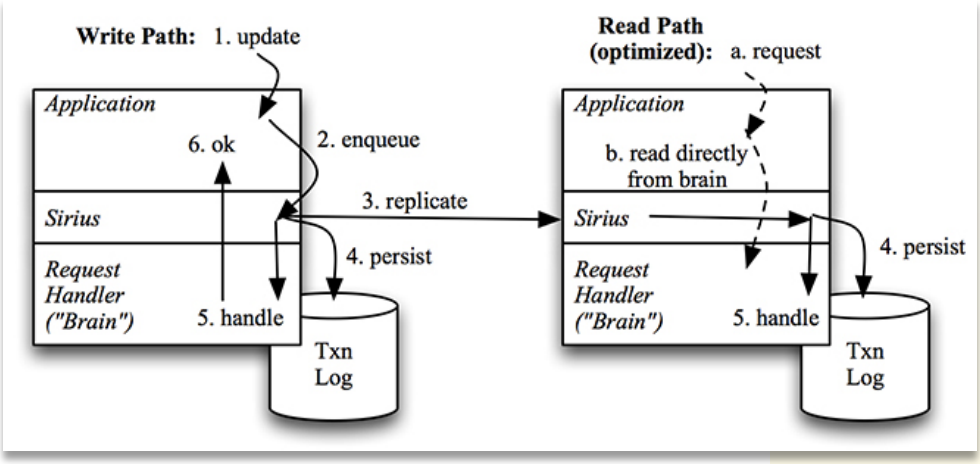
S I R I U S

Testing Sirius With Jepsen

Maulan Byron, James Wakemen & Andrew Wang



One of the "eight fallacies of distributed computing," famously coined by L. Peter Deutsch, is "the network is reliable." It's a fallacy because perhaps the hardest problem to solve, when architecting and maintaining any distributed application, is the detection and correction of network errors.

This is all the more important when considering the scale and overall heft of what is a video-centric cloud. For ours, we developed an open source and distributed system library, called Sirius. Its purpose is to deliver and coordinate data updates amongst a cluster of nodes. Sirius enables developers to build an absolute ordering for updates that arrive in the cluster, ensuring that cluster nodes eventually receive all updates, and persisting the updates on each node. Such updates are generally used to build in-memory data structures on each node, allowing applications using Sirius to have direct access to native data structures, representing up-to-date data.

Sirius does not, however, build these data structures itself -- instead, the client application supplies a callback handler. This allows developers using Sirius to build whatever structures are most appropriate for their application.

Said another way: Sirius enables a cluster of nodes to keep developer-controlled, in-memory data structures eventually consistent, allowing I/O-free access to shared information.

As noted in 'How Sirius Works' page, Sirius applies the updates in a totally ordered manner. This means that the order of the updates is the same for all the nodes. Sirius uses multi-paxos to establish this order. Nodes then distribute the updates and add them to an in-memory queue to be able to apply them in order. Sirius uses a catch up algorithm to check for and obtain any missing updates from neighboring nodes. To create a persistent state, the updates are written to a write-ahead transaction log before calling the application's callback function. This allows a node to be stopped and restarted, able to replay the transactions and get back to its original state before reconnecting to the cluster.

We wanted to test our library under conditions that would replicate the effects of a network that was malfunctioning in controlled ways. In particular, the tests would be run to determine how well the Sirius library performs while the underlying network is suffering from partitioning.

Which brings us to Jepsen -- an invaluable application tool that uses a set of virtual machines, attached to a virtual network, to perform various actions against the application to be tested -- while using its nemesis functionality to partition the network. This gives us an automated way to degrade or break the network out from under Sirius.

Because Sirius is a library and not a complete application, Jepsen tests were performed against the Sirius reference application written in Java. This reference application uses a REST-styled API to allow for basic operations against the datastore. The operations are limited to gets, puts, deletes, and listing of the keys.

For the purposes of these tests, the clients within Jepsen needed to be as simple as possible. For that reason, we first created a minimal HTTP-based client that performs operations using a specific Sirius node in the cluster. Jepsen then instantiates several of these clients, each using a different Sirius node.

Next, we created a set-client within Jepsen to call the REST services of the reference application. The set-client needed the functionality to both store items, and retrieve the list of items previously stored. This matched well with the REST API of the reference application. (The reference app did not expose a conditional put operation, so we did not create a check-and-set client.) Our set-based clients would then perform a series of puts to the datastore under various conditions, and then check the list of objects saved in the datastore. The results were then compared against the expected values from the operations, and the differences reported.

As you can see below, the client implements two operations: add and read. The add function sends an add request to the Sirius reference application instance that it is associated with at the time of its creation, and the read operation reads from the datastore and reports the list of keys as a set of values. Jepsen uses this set to perform the final comparison.

Figure 1a: *Client-initiated add and read requests are used by Jepsen to perform a final comparison.*

```
(defrecord CreateSetClient [url]
client/Client
  ( setup! [ _ test node ]
    ( let [ url ( str "http://" ( name node ) ":8000" ) ]
      ( info node "node url: " url )
      ( CreateSetClient. url )
    )
  )

  ( teardown! [ _ test ] )

  ( invoke! [ this test op ]
    ( case ( :f op )
      :add  (timeout 6000 (assoc op :type :info :value :timed-out)
            (let [r (webclient/put (str url "/storage/jepsen/" (:value op)) {:body (str (:value op))
                                                                             :throw-exceptions false})]

            (case (:status r)
                200 (assoc op :type :ok)
                (assoc op :type :info :value (:body r)))))
      :read ( timeout 6000 ( assoc op :type :info :value :timed-out )
              (let [r (webclient/get (str url "/keys") {:throw-exceptions false})]
              (case (:status r)
                200 ( assoc op :type :ok :value (set (map parse-int (str/split-lines (str/replace (:body r) #"jepsen/" "")))))
                404 ( assoc op :type :info :value :not-found)
                (assoc op :type :info :value (:body r)))))
    )
  )
)
```

Figure 1b: *The Test it self!*

```
(deftest create-test
    (let [test (run!
        ( assoc
            tst/noop-test
            :name "sirius-dummy"
            :os        debian/os
            :db        db
            :client    (create-set-client)
            :model     (model/set)
            :checker   (checker/compose {:html timeline/html
                                         :set  checker/set})
            ;:nemesis   (nemesis/partitioner nemesis/bridge)
            :nemesis   (nemesis/partition-random-halves)
            :generator (gen/phases
                        (->> (range)
                            (map (fn [x] {:type  :invoke
                                          :f     :add
                                          :value x}))
                             gen/seq
                            (gen/stagger 1/10)
                            (gen/delay 1)
                            (gen/nemesis
                                (gen/seq
                                    (cycle [(gen/sleep 60)
                                            {:type :info :f :start}
                                                (gen/sleep 300)
                                            {:type :info :f :stop}])))
                                (gen/time-limit 600))
                            (gen/nemesis
                                (gen/once {:type :info :f :stop}))
                            (gen/clients
                                (gen/once {:type :invoke :f :read}))
                        )))]
    (is (:valid? (:results test)))
    (pprint (:results test))))
```

The test we developed runs for a fixed length of time. Starting with a fully functional network, we then “break” the network in interesting ways.. To accomplish this, we used two of the built-in nemesis functions. The first nemesis is a bridge nemesis, which creates a situation in the network where the five nodes are divided into two pairs and a single node. The nodes in the pairs can see their companion and the single node, but not the nodes from the other pair. This allows the single node to act like a bridge between the two halves of the network.

The second nemesis is a partition-random-halves nemesis. It takes the five nodes in the network and divides them into two sets, roughly half sized. One set is a pair and the other is a triple. The nodes in one half cannot communicate with the nodes in the other half. In the code above, the bridge nemesis has been commented out and the random-halves nemesis has been put in its place.

Figure 2: *Two built-in nemesis functions enabled us to "break" the network in interesting ways.*

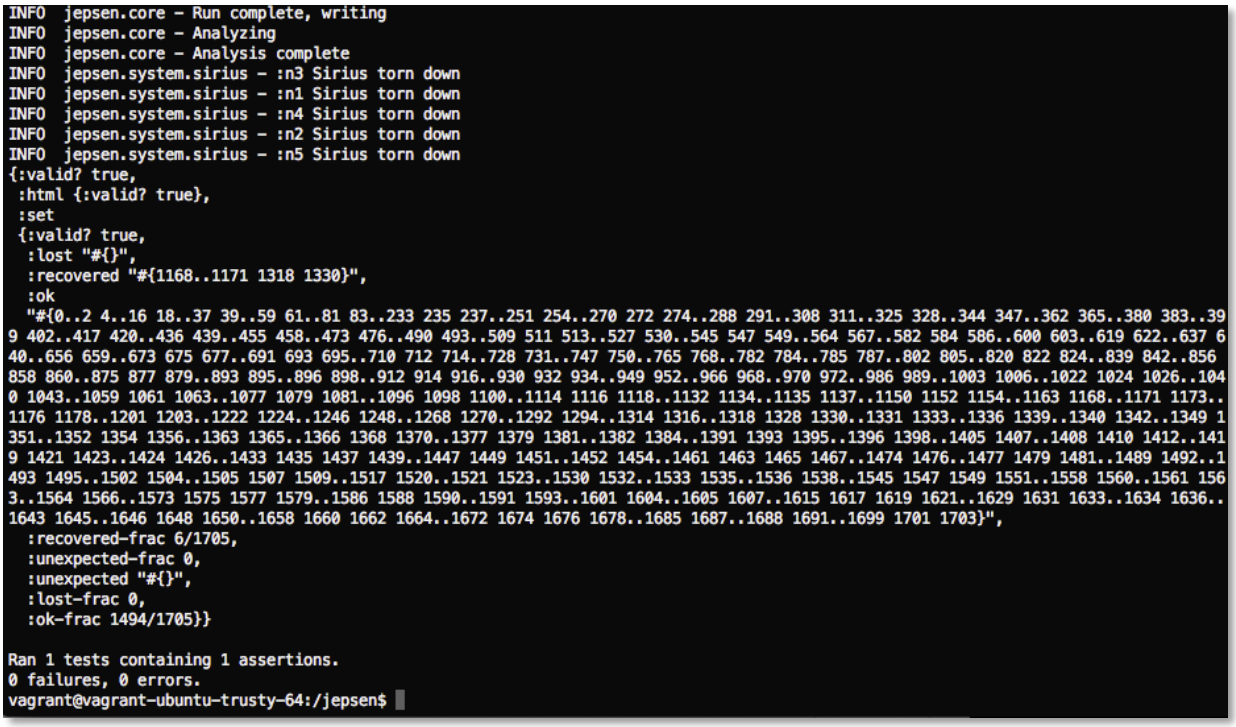
```
INFO  jepsen.core - Run complete, writing
INFO  jepsen.core - Analyzing
INFO  jepsen.core - Analysis complete
INFO  jepsen.system.sirius - :n4 Sirius torn down
INFO  jepsen.system.sirius - :n2 Sirius torn down
INFO  jepsen.system.sirius - :n1 Sirius torn down
INFO  jepsen.system.sirius - :n3 Sirius torn down
INFO  jepsen.system.sirius - :n5 Sirius torn down
{:valid? true,
 :html {:valid? true},
 :set
 {:valid? true,
  :lost "#{}",
  :recovered "#{}",
  :ok "#{1..15 17..37 39..59 61..83 85..2632}",
  :recovered-frac 0,
  :unexpected-frac 0,
  :unexpected "#{}",
  :lost-frac 0,
  :ok-frac 2628/2633}}

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
vagrant@vagrant-ubuntu-trusty-64:/jepsen$
```

The tests were run so that there were about 2600 insertions each. In neither case did Jepsen report that data was "lost" In Jepsen terms, "lost" data is data that has been sent to the datastore and acknowledged by a datastore write, but in the end, the data was not in the set of keys stored in the datastore.

The first test was performed using the bridge nemesis. In the bridge case, five of the inserts were not successful -- but because these were reported as being errors, they weren't "lost," which enabled the client to retry the operation.

As you can see from the results -- “0 failures, 0 errors”, once the cluster reaches stability, it doesn’t miss a beat -- it just works, even as the underlying network is having trouble.

Figure 3: *Once a cluster reaches stability, it just works - even if the underlying network is funky..*



The second test used the random-halves nemesis. This test was harder on the cluster and fewer of the additions were successful, but it didn’t lose any acknowledged writes. There were a few instances where the cluster was able to recover an add error. Jepsen considers it an error when a write is returned, but it was actually successful as “recovered” write. Since the writes of the same value are idempotent, and even without recovery taking place, any successful retry of the failed add results in an identical set of values to what would have been, had there been no initial failure.

The main point to take away from this is that the Sirius library is not acknowledging and then failing to persist additions to the dataset. If the Sirius library says that the addition was complete, it really was.

The other benefit to the library is that the entire dataset is replicated to each node eventually, which allows the retrieval of the data to incur a low to zero I/O penalty (Sirius' main use case holds the data in memory.) This makes the cluster's nodes perennially readable, even if the data is stale. The library was specifically designed to support applications that need to be available for reads under potentially adverse conditions, and a stale read is better than no read in those cases.

The latest update to Jepsen introduced a new analysis tool called Knossos. Knossos works by checking to see if the history of the parallel client operations is "linearizable" --- meaning it appears to the rest of the system to occur instantaneously. It does so by looking for possible permutations of the history, until a linearization is found. It doesn't try to find multiple versions. Once it finds one, it is complete. This type of check is essential to prove the functionality of any system that guarantees a high level of consistency.
Sirius does not offer such a high degree of consistency. Within Sirius, the updates/writes are sequenced using Paxos. This allows Sirius to guarantee that there is one order for all the updates, and that all nodes in the cluster apply the updates in that order. Sirius guarantees that each node is somewhere in the sequence of updates, not that the node is at the *end* of the sequence. Sirius doesn't use Paxos for reading data from the store, in any way. For the canonical use case, for instance, Sirius is bypassed for the reads. This design allows us to achieve eventual consistency -- which quickly led us to the conclusion that testing Sirius operations with Knossos did't make sense, as Sirius does not offer a level of consistency that Knossos would be testing.

As a result, Sirius succeeds in the following design goals for a highly available datastore for reference datasets:

1. The data needs to be in memory.
2. The application would tolerate eventual consistency of the slowly changing data.
3. The read throughput needs to be much higher than the write throughput.
4. Acknowledged writes could not be lost.

The Jepsen testing of the reference application showed that the library does indeed meet those requirements.

We've long admired how the Jepsen testing framework analyzes open source projects, often finding some pretty terrible bugs. That's why we opted to run it against our Sirius library -- to find and fix bugs, of course, especially because the Paxos protocol is notoriously difficult to implement. Getting the "badge of honor" that is zero bugs? Well, that was pretty nice too.

So, as it relates to the eight fallacies of distributed computing, it would appear that the network *is* reliable -- or can be, with careful testing using powerful open source tools, like Jepsen. Thank you Jepsen (and its creator, Kyle Kingsbury)!