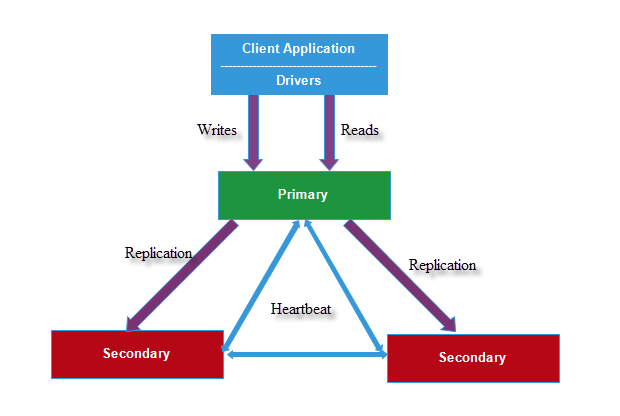
Replica Set

Our replica set has been implemented on a cluster in particular:

Replica sets provide redundancy and high availability, and are the basis for all production deployments.



The above figure illustrates how mongoDB manage a replica set on different servers (without arbiter).

“The primary node receives all write operations. A replica set can have only one primary capable of confirming writes with { w: "majority" } write concern; although in some circumstances, another mongod instance may transiently believe itself to also be primary. The primary records all changes to its data sets in its operation log, i.e. oplog.

The secondaries replicate the primary’s oplog and apply the operations to their data sets such that the secondaries’ data sets reflect the primary’s data set. If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.

Secondaries replicate the primary’s oplog and apply the operations to their data sets asynchronously. By having the secondaries’ data sets reflect the primary’s data set, the replica set can continue to function despite the failure of one or more members.”

[from mongoDB official documentation]

Read Availability over Consistency

We can see that mongoDB is more prone to consistency of the reads operation over the availability (when an application insert/query our database the operation is executed by default on the “primary” server). We can manage the default settings of MongoDB to permit read operations on the secondary. It’s important to notice that mongoDB implements an asynchronous replication to secondaries and this implies that reads from secondaries may return data that does not reflect the state of the data on the primary (from MongoDB manual).

Assuming that a consistent number of users will use our service, we don’t want to make an user wait to have a response in case of failure of the primary (the users will not have response for the time needed to elect a new primary) .

We managed the read preference and imposed the read option primaryPreferred

That according to the mongoDB official documentation this option “if the primary is unavailable, as is the case during failover situations, operations read from secondary members that satisfy the read preference’s maxStalenessSeconds and tag sets”. We have used the maxStalenessSeconds (time accepted since the database wasn’t updated by the primary) and tag sets used by default in MongoDB.

In case of the failure of the primary, if there was some un synchronized write, these operation will not be present in the secondaries (so the databases are not consistent) until the recovery of the old primary.

Replica set configuration

We have initialized three mongod instance for the replica set in particular

Server 172.16.1.5:

Listening port: 27017

Replica: /data/rs1

Server 172.16.1.7:

Listening port: 27018

Replica: /data/rs2

Server 172.16.1.8:

Listening port: 27019

Replica: /data/rs2

For each server we added in /etc/hosts the ip address and the associated host name (to permit an easier replica set configuration). We need to add to the DNS of each local machine using the application the ip and hostnames of the server. In the server 172.16.1.5 we wrote the configuration of our replica set (specifying the host members and the listening ports) and executed the command to activate the replica set and the command to allow read on the secondaries.

We simulated a failover on the primary server ( db.adminCommand({"shutdown" : 1})) and ensured that our application continued to work as expected.