**MongoDB Replica Set Configuration**

[**http://www.10gen.com/presentations/mongosv-2011/a-mongodb-replication-primer-replica-sets-in-practice**](http://www.10gen.com/presentations/mongosv-2011/a-mongodb-replication-primer-replica-sets-in-practice)

[**http://docs.mongodb.org/master/MongoDB-Manual-master.pdf**](http://docs.mongodb.org/master/MongoDB-Manual-master.pdf)

In the previous examples we started a cluster on one machine. We started a mongod process using parameters to specifying the dbpath and the log file location. In a production environment each mongod process is represented by a replica set.

**Background:**

A replica set is implemented in a master/slave architecture. The master is defined as the primary, the secondaries as the slave. This is implemented using an asynchronous protocol for better performance. Most cluster arechitectures use asynchronous replication including Hadoop.

A synchronous protocol where the write from the master is propagated to the slaves would be consistent. On the loss of the primary node you would not have to perform additional administration for recovery.

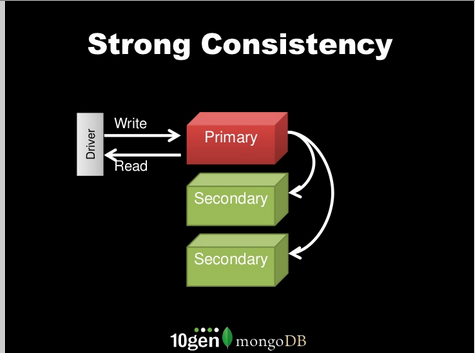
Asynchronous protocols require more maintenance.

When a primary goes down as detected by the failure of a heartbeat, a new primary from the secondary nodes is elected. All the writes now go to the new primary. We can see in the logs which node in the replica set is the primary and the driver will send writes to the primary.

There are 2 models in cluster theory, Strong Consistency and Eventual Consistency.

**Strong Consistency:**

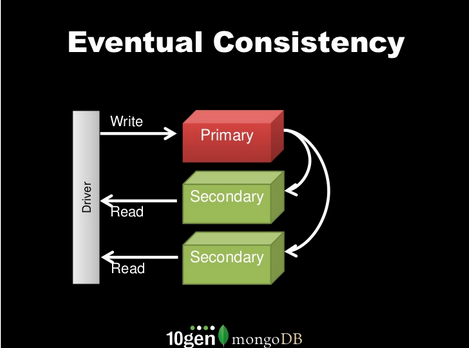
In the Strong consistency model all reads and writes go to the master. The advantage of this architecture is if I send a write to the db, I am guaranteed to see the most recent write when I do a read of that same data. Sending all writes and reads to the same node for concurrency is one of the dominant design choices because it allows the system to add a cache to improve read performance.



**Eventual Consistency:**

In an eventually consistent model the writes go to a master/primary but the reads come from a slave. The problem is if I do a write I am not guaranteed to see the most data because the write data from the master/primary might not have propagated to the seconedary/slaves nodes in time. For some applications this is ok. MongoDB can be configured to be in an eventually consistent mode as a strategy to try to increase read performance at the loss of viewing stale data. For some applications like shopping carts this model would not work.

There is an argument Eventual Consistency is for scale to get reads to perform faster. Most large scale systems such as the ones at Google use distributed locking (like the Google Chubby service) or MVCC (HBase solution) to guarantee consistency in a replicated cluster. Some systems like Cassandra don’t have this option and only offer the user an eventually consistent write model.

****

**Durability Models:**

Durability is the definition of when I do a write does it stay in the event of a failure? The primary problem is when data is placed in cache before it is written into cache; if the power goes away the data in the cache is lost before it is written to disk.

MongoDB contains several durability models:

1. Write and forget durability model: When you send a write to a database, the primary responds and you don’t wait to see if there is an acknowledgement. This is good for applications where you can lose writes. There is no log of error messages where these writes which didn’t complete are logged. You have to implement this logic in the application layer.
2. Write and wait for error: This is the most common mode, also called SAFE mode where the WriteConcern is set to safe mode. There is a getLastError() in the MongoDB driver which tells the DB to collect the error and send to the Application. This is the Safe mode flag. Tell the driver to send the getLastError() command, MongodB will wait and check if any error occurred.

There are 3 durability settings for getLastError(),

1. Journal Sync which guarantees the write is written to the journal which happens every 100ms or
2. FSYNC which is an operating systems call which happens every minute or so. FSYNC much slower but you don’t have to worry about managing the journal for recovery which is a separate log.
3. W flag which controls how many replicas the write is committed to. W=2 says 2 places have the write in memory to another node in the cluster. This is faster than writing to disk with either the Journal or FSYNC mode.
4. Majority in WriteConcern, automatically writes to 2/3 if there are 3 nodes. Don’t have to manage number of clusters.
5. Tagging: allows the definition of tags to specify replication factor across data centers. Can define these custom error modes to define durability across data centers.
   1. getLastErrorModes:{

veryImportant:{dc:3},

sortOfImportant:{dc:2}

}

Priorities: Each primary and secondary node has a priority setting which sets which secondary becomes the master first. A priority of 0 will mean the secondary will never become a primary. This is required for a delayed backup which is recommended for production situations for user rollback.

Slave Delay: specify how far behind the replica you want in back of the master. E.g. if a user accidentally drops a database and then this is replicated across the database and the data is gone. A slave delay member allows recovery.

Arbiters: determines a quorum, a majority in a 3 node cluster is 2. You need a >50% majority for a quorum. The arbiter is a separate server required to determine quorum.

Hidden: keep replica of data, can run this for backup. Never send application traffic to this node.

A production cluster should implement several or all of these options for operational stability and backup/recovery:

1. a replica set consisting of 3 servers, one primary and 2 secondary nodes. You need an arbiter to run in addition to this set of 3 servers.
2. A separate backup node which takes no application traffic.
3. A delayed replica.
4. Multi data center support in different regions

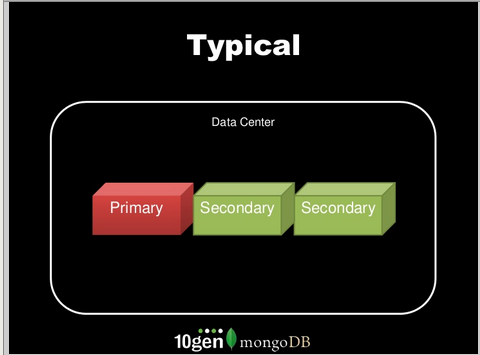
We can simulate all of these production configurations in the Amazon AWS Cluster.

The issue with the durability models is the impact it has on the recovery procedures. If the master fails and a new master is elected then there is a point in time where the new writes either have to be migrated from the replica to the new master or have to be undone because they can’t be verified.

**Replica Introduction:**

This is an introduction to creating a replica set on a set of servers on one data center. Replication scenarios across multiple data centers with sharding is in a separate document.

We will demonstrate the configuration of a typical replica set:



Create the subdirectories and log files for the replica processes:

mkdir -p srv/mongodb/rs0-0 srv/mongodb/rs0-1 srv/mongodb/rs0-2

**Note: this is modified to use the local directory instead of starting from /. In a production environment conventions have to be established which involve disk mounts and LVM partitions if they are used. A local path reduces the steps to make this demo easier to understand and with less questions around file permissions and sudo rights.**

On one machine start the following commands either in separate xterm windows. You can modify the commands to specify a log file, so you don’t see the log output directed to stdout. For easy debugging it is easier to display separate xterm windows so you can see if all replica window display the same status messags.

**NOTE: the mongodb documentation is incorrect. There are missing commands in there which will create error messages like;**

**NOTE: the instructions below are different and modified from the online MongoDB instructions.**

Mon Aug 2 11:30:19 [startReplSets] replSet can't get local.system.replset config from self or any seed (EMPTYCONFIG)

mongod --port 27017 --dbpath srv/mongodb/rs0-0 --replSet rs0

mongod --port 27018 --dbpath srv/mongodb/rs0-1 --replSet rs0

mongod --port 27019 --dbpath srv/mongodb/rs0-2 --replSet rs0

> config = {\_id: 'rs0', members: [

... {\_id: 0, host: 'localhost:27017'},

... {\_id: 1, host: 'localhost:27018'},

... {\_id: 2, host: 'localhost:27019'}]

... }

{

"\_id" : "rs0",

"members" : [

{

"\_id" : 0,

"host" : "localhost:27017"

},

{

"\_id" : 1,

"host" : "localhost:27018"

},

{

"\_id" : 2,

"host" : "localhost:27019"

}

]

}

> rs.initiate(config)

{

"info" : "Config now saved locally. Should come online in about a minute.",

"ok" : 1

}

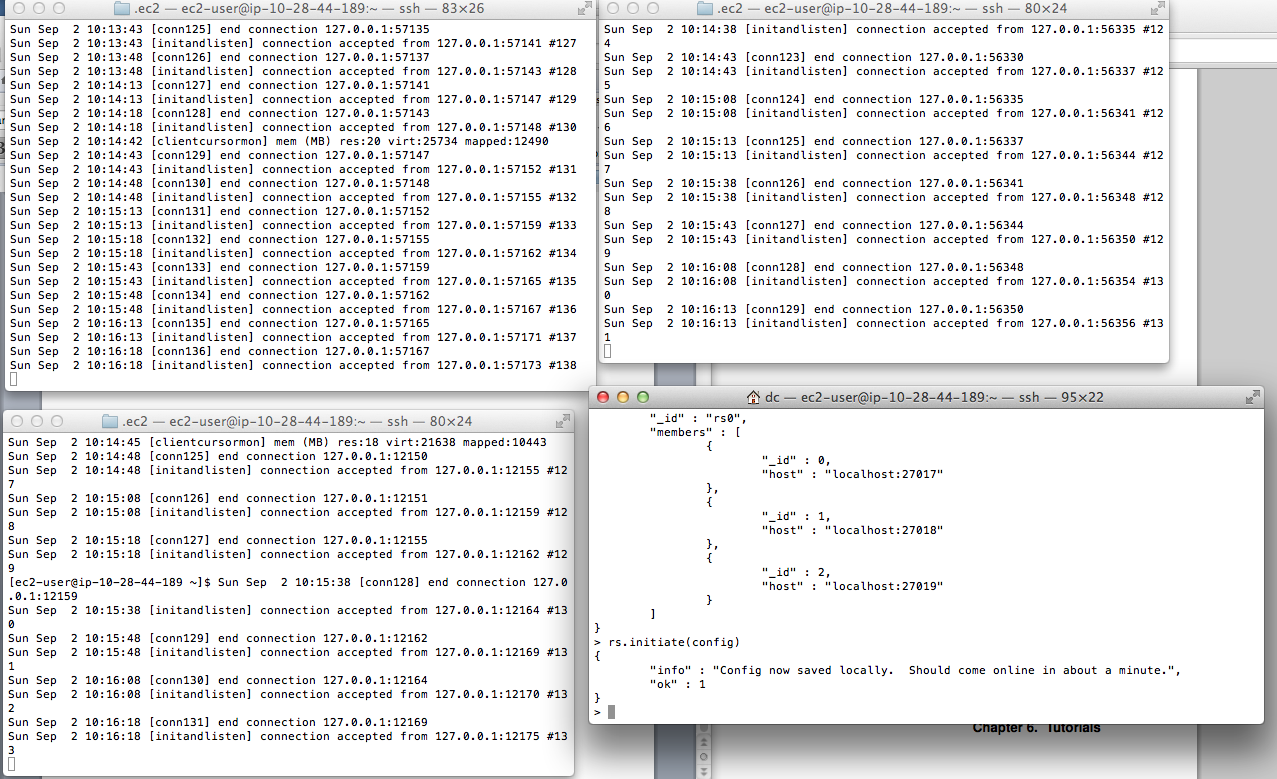
>

Once this running you should see all 3 nodes in a steady state waiting for data:



**Adding data and verifying the replication:**

**To facilitate debugging and verification you can open up terminals where all the replica sets are displayed on the screen and the Client is on a window. As we add and delete data we should see data propagate across the replica sets.**

****

**There is a 2 second heartbeat between the replica nodes:**

**Macintosh HD:Users:dc:Desktop:Screen Shot 2012-09-02 at 3.27.10 AM.png**

**Insert some data by pasting some JS code into the client.**

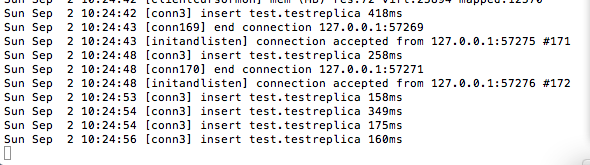
for(var i=0;i<10000000;i++){

db.testreplica.save({i:i+100});

}

**Runtime:**

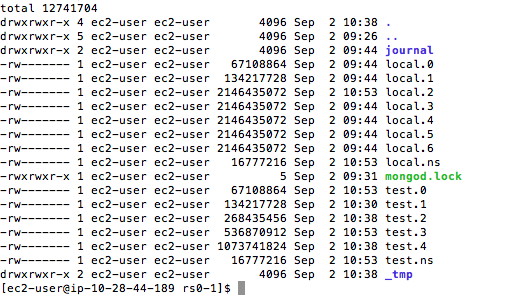
**You can see the writes being logged into the master:**

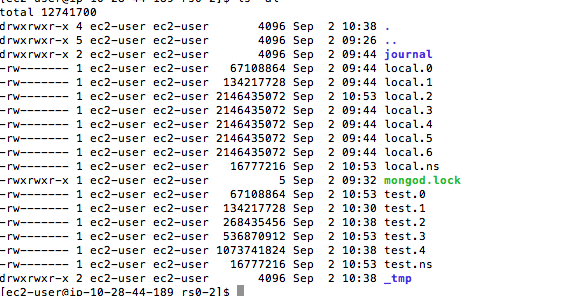
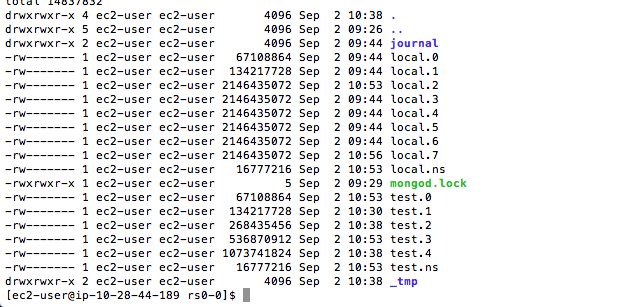
****

**In this configuration data is being replicated to the master. The writes don’t show up but when the replicas need more space they open a file and initialize with 0s before a write. This is mongodb-2.06, the later versions don’t do the init.**

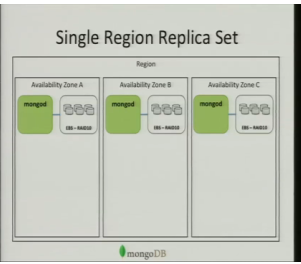
**Macintosh HD:Users:dc:Desktop:Screen Shot 2012-09-02 at 3.26.36 AM.png**

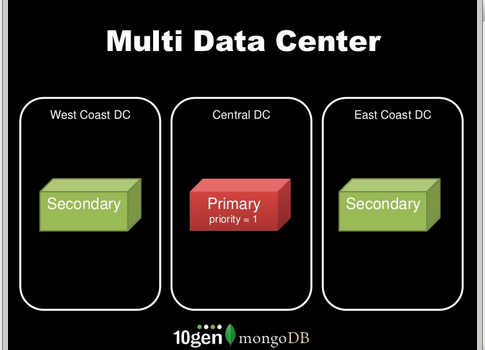
**We can look at the disk space in each of the replica sets and see they are identical:**

****

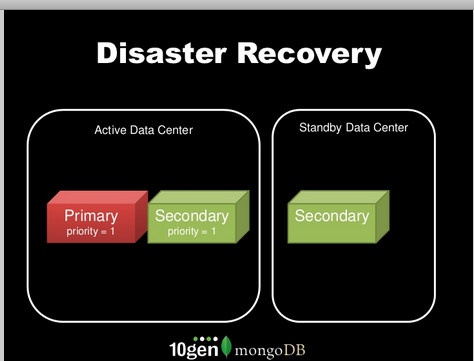
****

**Replica sets can be spread out over multiple data centers to prevent a single point of failure or over different availability zones in AWS.**

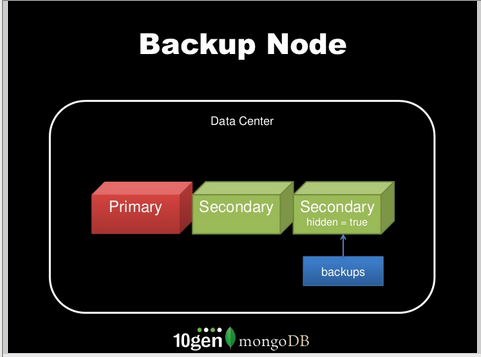
****

****

**Hot Standby:**

****

**Backup Nodes: A backup node is necessary to allow administrators to perform rollback of transactions (not ACID) which users want to undo. This capability is not like what is present in a database and is approximated by keeping a replica with a replication delay.**

****