Clase 2 – Arquitectura Distribuida

Replicación de Datos

Replicación

- La replicación se realiza a través de estructuras denominadas **Replica Set**.
- Un **Replica Set** es un grupo de instancias **mongod** que mantienen el mismo conjunto de datos. Los replica sets proveen ***durabilidad*** *(Redundancia)* y ***alta disponibilidad*** *(Failover automático)* de los datos que almacenan.
- Utilizan un esquema derivado del Maestro-Esclavo.
- Un **nodo** (proceso mongod) se define como **Primario** (maestro) y recibe todas las operaciones de escritura.

Replicación

- No puede haber mas de **un nodo primario** simultáneamente, por lo tanto se proporciona una consistencia estricta en la escritura.
- Los **nodos secundarios replican** el **log de operaciones** del **nodo primario** y aplican las operaciones en sus datos **asincrónicamente**.
- Si el **nodo primario no** está **disponible**, se elige **un secundario como primario** (por votación).

Replicación

27058

27059

27060

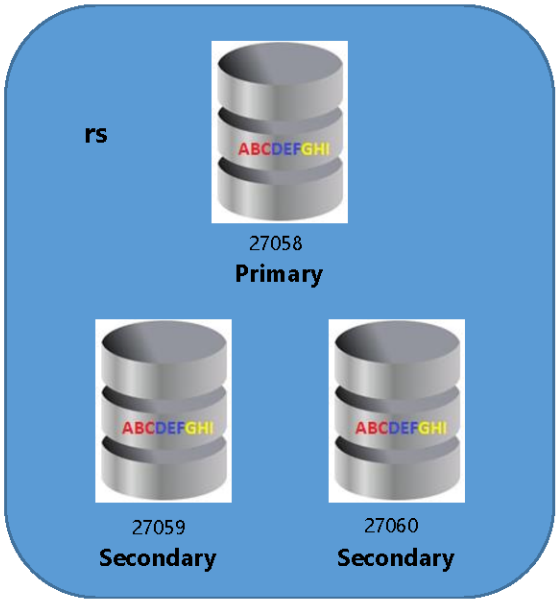
Creamos un servidor levantando una instancia

mongod **--replSet rs** --port 27058.....

Creamos 2 nuevas intancias de mongod

mongod **--replSet rs** --port 27059.....
mongod **--replSet rs** --port 27060.....

Replicación



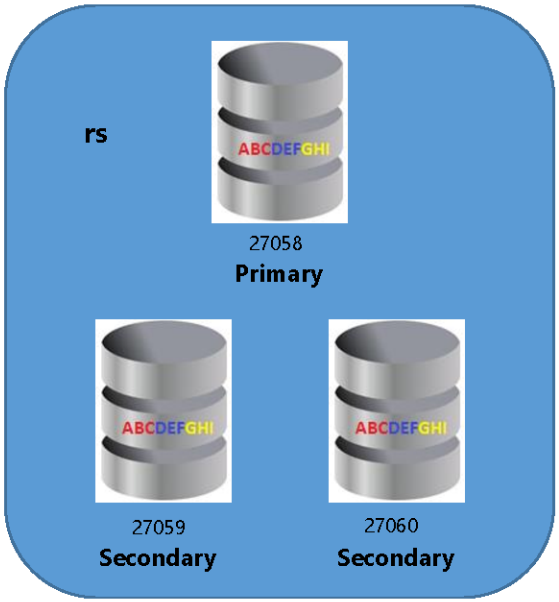
- Configuramos un ReplicaSet con un servidor

```
mongo --port 27058
      cfg={_id:"rs",
            members:[{_id:0, host:"localhost:27058"}]
      }
      rs.initiate( cfg )
```

- Agregamos al ReplicaSet nuevos servidores

```
      cfg={_id:"rs",
            members: [
                  {_id:0, host:"localhost:27058"},
                  {_id:1, host:"localhost:27059"},
                  {_id:2, host:"localhost:27060"}
            ]
      }
      rs.reconfig(cfg)
```

Replicación



- Configuramos un ReplicaSet con un servidor

```
mongo --port 27058
      cfg={_id:"rs",
            members:[{_id:0, host:"localhost:27058"}]
      }
      rs.initiate( cfg )
```

- Otra manera de agregar nuevos nodos al Replica Set

```
      rs.add("localhost:27059")
      rs.add("localhost:27060")
ó
      rs.add({_id:1,host:"localhost:27059"})
      rs.add({_id:2,host:"localhost:27060"})
```

Configuración del Replica Set

```
rs:PRIMARY> rs.conf()
{
        "_id" : "rs",
        "version" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "localhost:20000",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "slaveDelay" : 0,
                        "votes" : 1
                }
        ],
        "settings" : {
                "chainingAllowed" : true,
                "heartbeatTimeoutSecs" : 10,
                "getLastErrorModes" : {

                },
                "getLastErrorDefaults" : {
                        "w" : 1,
                        "wtimeout" : 0
                }
        }
}
```

Ejecutando `rs.conf()` veremos la configuración del Replica Set.

- _id muesta el nombre del Replica Set
- version irá cambiando cada vez que hagamos un `rs.reconfig()`.
- settings tiene configuración adicional sobre, como el tiempo a esperar para que los nodos respondan si están activos o no y el write concern utilizado por default, que es recibir la confirmación de escritura cuando el Primary almacenó el dato en memoria.
- members tiene un array con todos los miembros del Replica Set.

Configuración del Replica Set

```
rs:PRIMARY> rs.conf()
{
        "_id" : "rs",
        "version" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "localhost:20000",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "slaveDelay" : 0,
                        "votes" : 1
                }
        ],
        "settings" : {
                "chainingAllowed" : true,
                "heartbeatTimeoutSecs" : 10,
                "getLastErrorModes" : {

                },
                "getLastErrorDefaults" : {
                        "w" : 1,
                        "wtimeout" : 0
                }
        }
}
```

El subdocumento de cada miembro tiene:

- **_id** identificador único del miembro.
- **host** ip y puerto donde corre la instancia mongod.
- **arbiterOnly** indica si el miembro tendrá una réplica de los datos o si es un árbitro.
- **hidden** indica si el miembro será visible para los drivers que se comuniquen con el Replica Set.
- **priority** es la prioridad del miembro para ser elegido Primary. priority: 0 implicará que nunca será elegido.

Configuración del Replica Set

```
rs:PRIMARY> rs.conf()
{
        "_id" : "rs",
        "version" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "localhost:20000",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "slaveDelay" : 0,
                        "votes" : 1
                }
        ],
        "settings" : {
                "chainingAllowed" : true,
                "heartbeatTimeoutSecs" : 10,
                "getLastErrorModes" : {

                },
                "getLastErrorDefaults" : {
                        "w" : 1,
                        "wtimeout" : 0
                }
        }
}
```

El subdocumento de cada miembro tiene: (cont.)

- **slaveDelay** indica el retraso en segundos que tendrá el servidor con respecto al Primary.
- **votes** es la cantidad de votos que tendrá el miembro. No puede haber mas de 7 miembros votantes, por lo que si hay mas, deberán tener votes: 0.
- **tags** servirá, en conjunto con getLastErrorModes, para crear un write concern personalizado.

Atributos para un Miembro de un Replica Set

```
...
members: [
    {
        _id: <int>
        , host: <string>,
        arbiterOnly: <boolean>,
        buildIndexes: <boolean>,
        hidden: <boolean>,
        priority: <number>,
        tags: <document>,
        slaveDelay: <int>,
        votes: <number>
    },
     ...
],
...
```

Atributo arbiterOnly

- Es un atributo opcional de **tipo booleano (true/false)**.
- El árbitro puede votar en una elección de PRIMARY, pero no puede ser votado como PRIMARY.
- El nodo **Arbiter** no posee datos, ya que sólo sirve para aportar un voto válido en una elección de PRIMARY dando quorum.
- Permite al replica set tener un número impar de miembros. (recomendado).
- No tiene la sobrecarga (overhead) de un miembro Secondary que debe replicar los datos.
- Sólo añadir un árbitro a replica sets con números pares de miembros. Si añade un árbitro a un conjunto con un número impar de miembros, en el Replica Set podrán haber de elecciones empatadas.

rs.addArb("<host>")
rs.add ("<host>", **{arbiterOnly: true}**)
rs.add({ host: "<host>", **arbiterOnly: true** })

Atributo priority

Este atributo indica el grado de elegibilidad de un miembro del replica set para ser PRIMARY.

- *Es un atributo opcional* de **tipo numérico con un rango entre 0 y 1000**. *El valor* ***default es 1***.
- Conviene especificar un **valor más alto a los miembros más elegibes** para ser **PRIMARY**.
- Por el contrario, conviene especificar un **valor bajo** para hacer a un miembro **menos elegible**.
- Las prioridades sólo son usadas para compararse con otros miembros del mismo replica set.
- **Al cambiar el balance de prioridades de un replica set**, se disparará una **elección de primary**.
- Si a un miembro se le asigna **prioridad 0**, el mismo **nunca podrá ser primario**.

Nodos Secundarios con priority 0

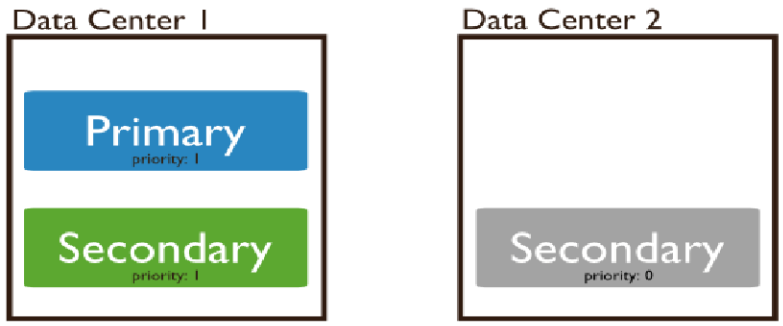
Se puede configurar un secundario con un atributo de Priority 0, esto significa que:

- **Nunca** podrá ser **PRIMAR**Y, **ni votado** como PRIMARIO.
- **Mantiene** una **copia** del conjunto de los **datos**.
- **Acepta** operaciones de **lectura**.
- **Vota** en elecciones.
- Es **particularmente usado** en Implementaciones con **Multi-data centers**.

Nodos Secundarios con priority 0

Posibles Usos

- ***Implementaciones Multi-Data Centers.***



Las escrituras están más cercanas al DC1, por lo que la latencia de escritura en el nodo secondary del DC2 no es la apropiada.

Nodos Secundarios con priority 0

Posibles Usos (Cont.)

- ***Nodos con Prioridad 0 como Nodos Standby.***
  - **En algunos** replica sets, **no es posible agregar** nuevos **miembros** en un **tiempo razonable**. Un **nodo standby** mantiene una copia actual de los datos y podría **reemplazar a un miembro no disponible.**

- ***Nodos con Prioridad 0 como Backup Server o Reporting Server.***
  - En algunos replica sets, se configuran nodos con prioridad 0, para ser nodos en los que se realiza el backup del replica Set ó se destinan para servidor de reportes.
  - Luego veremos **Hidden Members**.

Nodos Secundarios con priority 0

Ejemplo 1 – Pasamos a Priority 0 a un miembro del ReplicaSet

```
cfg = rs.conf()
cfg.members[0].priority = 0
rs.reconfig(cfg)
```

Ejemplo 2 – Agregamos un nuevo Miembro al ReplicaSet con Priority 0

```
rs.add({_id:3, host: "www.develop.net:27017", priority: 0 })
```

Atributo Hidden

El atributo Hidden en TRUE hace que un miembro del ReplicaSet esté invisible para las Aplicaciones Clientes.

Los nodos con Hidden en TRUE:

- **Deben** tener configurado **priority en 0**, ya que **no pueden ser PRIMARY**.
- **Votan** en elecciones.
- El método db.isMaster() no muestra los **miembros Hidden**.
- Usar **nodos Hidden para tareas dedicadas** como **reporting** o **backups**.
- **No se tendrán en cuenta** para la configuración de "**Read Preferences**". (Ver próximos slides)
- Los **miembros Delayed**, deben ser **miembros ocultos**. (Ver próximos slides)

Nodos Secundarios Hidden

Ejemplo 1 – Pasamos a Priority 0 y Hidden a un miembro del un ReplicaSet

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
rs.reconfig(cfg)
```

Ejemplo 2 – Agregamos un nuevo Miembro al ReplicaSet con Priority 0 y Hidden

```
rs.add({_id:3, host: "www.develop.net:27017", priority: 0, hidden: true })
```

Atributo slaveDelay

Los miembros "Delayed" contienen una copia del conjunto de datos del replica set. Sin embargo, la **copia** que tienen refleja un **retraso en la actualización de los datos**.

Por ejemplo, si la hora actual son las 10.00AM y el NODO tiene un delay de una hora, este nodo Delay tendrá operaciones anteriores a las 09:00AM.

El atributo que permite configurar a un nodo como Delayed es **slaveDelay.**

Un miembro Delayed:

- **Debe ser** un miembro de **Prioridad 0.**
- **Convendría que sea** un miembro **Hidden.** Para que **prevenir lecturas atrasadas** desde una aplicación cliente.
- **Vota** en una elección.

El tiempo de Delay:

- Debe ser **igual o mayor** que su **ventana de mantenimiento**.
- Debe ser **menor** que la **capacidad** del **oplog**.

Nodos Secundarios Delayed

Ejemplo 1 – Pasamos a Priority 0, hidden y Delayed 1 hora a un miembro del ReplicaSet

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600

rs.reconfig(cfg)
```

Ejemplo 2 – Agregamos un nuevo Miembro al ReplicaSet con Priority 0, Hidden y Delayed 1 hora.

```
rs.add({_id:3, host: "www.develop.net:27017", priority: 0,
hidden: true, slaveDelay:3600})
```

Atributo votes

El **número de votos** de un miembro **compite** en la **Elección del PRIMARY**.

El **número** de votos por **cada miembro** del Replica Set puede ser **1 ó 0**.

Un réplica set permite tener **hasta 50 miembros** (version **3.0** en adelante), pero **sólo 7** pueden ser miembros **con Voto**.

Si tuviésemos un **replicaset con más de 7 miembros**, se deberá **configurar votes en 0** para el resto de lo **miembros que no votarán**.

Los **miembros** con **votes = 0 puesen ser** votados como **PRIMARY**, **pero** esto no los habilita a **votar** en elecciones.

Atributo votes

Ejemplo 1 – Configuramos el voto en 0 de un miembro del ReplicaSet

```
cfg = rs.conf()
cfg.members[2].votes = 0
cfg.members[3].votes = 0

rs.reconfig(cfg)
```

Ejemplo 2 – Agregamos un nuevo Miembro al ReplicaSet con Voto 0.

```
rs.add({_id:3, host: "www.develop.net:27017", votes: 0})
```

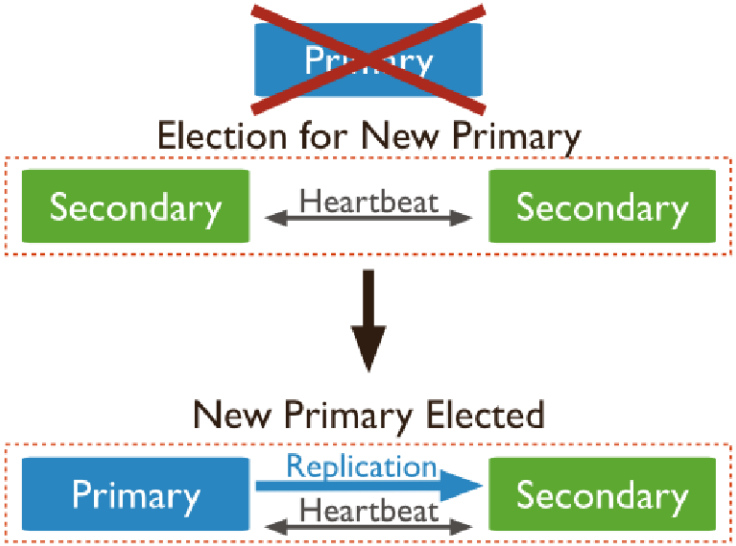
Primary

- El Primary del Replica Set es el **único nodo que puede recibir escrituras**, por lo que es necesario que, en caso que este deje de estar disponible, pueda **elegirse otro miembro** apto y transformarlo en Primary.

- Por default, el Primary es el único miembro del que se puede leer, pero esto puede ser cambiado especificándolo en la **Read Preference**.

- Para que un Primary continúe siéndolo, deberá poder **comunicarse** al menos con **la mayoría de los nodos** (la mitad de la cantidad de miembros + 1).

Failover automático



Cuando un **Secondary detecta** que el Replica Set **no tiene un Primary** y cumple las condiciones para serlo, llama a elección **postulándose** como Primary.
Si algún otro miembro votante conoce un **secondary más apto** para ser Primary (que esté más actualizado en cuanto a las operaciones del oplog o que tenga mayor prioridad), anulará la votación.
En caso de que consiga una **mayoría de votos** positivos, el miembro pasará a ser Primary.

Oplog

El **oplog** es un log donde se registran todas las operaciones de escritura que se van haciendo.
El método de replicación de MongoDB es asincrónico: los Secondaries son los responsables de mantenerse actualizados. Consultan periódicamente el oplog de otro miembro (Por default el Primary) para saber si hay nuevas operaciones de escritura que deban realizar.
El oplog se encuentra en la colección "*oplog.rs*" en la db "*local*", que es la única db que no se replica a otros miembros.

```
rs:PRIMARY> use local
switched to db local
rs:PRIMARY> show collections
me
oplog.rs
startup_log
system.indexes
system.replset
```

Oplog

Al insertar documentos, se crea una entrada por documento, indicando todos sus campos (incluido el _id). Tiene el timestamp de la operación, indica en que namespace se realizo y op:"i" significa que fue un insert.

```
rs:PRIMARY> db.repl.insert({x:1})
WriteResult({ "nInserted" : 1 })
rs:PRIMARY> db.repl.insert({x:2})
WriteResult({ "nInserted" : 1 })
```

```
"ts" : Timestamp(1457630972, 2),
"h" : NumberLong("7391950908998849541"),
"v" : 2,
"op" : "i",
"ns" : "ws.repl",
"o" : {
        "_id" : ObjectId("56e1aefbde47edb8d1ae8457"),
        "x" : 1
}


"ts" : Timestamp(1457630980, 1),
"h" : NumberLong("3126611757779888016"),
"v" : 2,
"op" : "i",
"ns" : "ws.repl",
"o" : {
        "_id" : ObjectId("56e1af04de47edb8d1ae8458"),
        "x" : 2
}
```

Oplog

Cuando se realizan operaciones que afectan multiples documentos, como un `remove` o un `update` con `{multi: true}`, en el oplog se guarda una entrada por cada documento. Sin importar el query usado dentro del update o remove, en el oplog se identifica cada documento unívocamente con su _id.

```
"ts" : Timestamp(1457631006, 1),
"h" : NumberLong("4289521851947198913"),
"v" : 2,
"op" : "u",
"ns" : "ws.repl",
"o2" : {
        "_id" : ObjectId("56e1af04de47edb8d1ae8458")
},
"o" : {
        "_id" : ObjectId("56e1af04de47edb8d1ae8458"),
        "x" : 1
}


"ts" : Timestamp(1457631059, 1),
"h" : NumberLong("-6900608960732677628"),
"v" : 2,
"op" : "u",
"ns" : "ws.repl",
"o2" : {
        "_id" : ObjectId("56e1aefbde47edb8d1ae8457")
},
"o" : {
        "$set" : {
                "x" : 0
        }
```

```
rs:PRIMARY> db.repl.update({x:2},{x:1})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
rs:PRIMARY> db.repl.update({x:1},{$inc:{x:-1}},{multi:true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

Journal

Para proveer **durabilidad** ante el caso de la caída repentina de una instancia, MongoDB usa archivos journal en el disco como un **write ahead log**, en los que, al llegar una escritura, se registrará la operación a realizarse antes de persistirse los datos reales en disco. La escritura en disco de los datos del journal se realiza con una frecuencia mucho mayor que la de los datos reales.

Si la instancia falla antes o mientras esta escribiendo datos a disco, tendrá en el journal las operaciones que deberá realizar para mantener la **integridad** y poseer escrituras mas recientes.