TP2 — Introduction à la réplication sur MongoDB

Base de Données NoSQL

Nom : Haiballa El Varougou

Année : 2025–2026

Table des matières

1 Initialisation du mode Replica Set sur MongoDB

1.1 MongoDB Replica Set, c'est quoi ?

Un système de gestion de base de données comme MongoDB vise à conserver des données de façon *durable*, *fiable* et *accessible*. Dans l'écosystème MongoDB, le mécanisme natif pour la **haute disponibilité** et la **tolérance aux pannes** s'appelle le **Replica Set**.

Un Replica Set correspond à un ensemble de processus `mongod` maintenant **une même base** grâce à la reproduction d'un **journal d'opérations** (l'**oplog**). En pratique, on y trouve :

- un **nœud primaire** (*Primary*) qui réceptionne les écritures ;
- un ou plusieurs **nœuds secondaires** (*Secondaries*) qui rejouent les opérations du Primary ;
- éventuellement un ou plusieurs **arbitres** (*Arbiters*) qui ne stockent pas de données mais participent aux votes.

Bonne pratique : prévoir un **nombre impair de membres votants** (nœuds de données + arbitres) afin de faciliter l'obtention d'une majorité lors des élections.

1.2 Fonctionnement général d'un Replica Set

Le principe est le suivant :

- les clients envoient les écritures au **Primary** ;
- le Primary enregistre les changements dans ses collections **et** dans l'**oplog** ;
- chaque Secondary lit l'oplog et applique les opérations **dans le même ordre**.

En cas d'indisponibilité du Primary (crash, maintenance, coupure réseau), les membres restants déclenchent une **élection** : si une majorité est atteinte, un Secondary suffisamment à jour devient le nouveau Primary et accepte les écritures. Lorsque l'ancien Primary revient, il se resynchronise puis reprend un rôle de Secondary.

En production, on recommande au minimum **trois membres votants** (par exemple 2 nœuds de données + 1 arbitre) pour conserver une majorité même en cas de panne d'un membre.

1.3 Avantages principaux d'un Replica Set

Les Replica Sets apportent plusieurs garanties importantes :

- **Haute disponibilité** : bascule automatique vers un Secondary élu Primary si le Primary tombe.
- **Tolérance aux pannes** : les données étant recopiées, la perte d'un nœud n'implique pas la perte des données.
- **Maintenance à chaud** : possibilité de redémarrer / mettre à jour un Secondary sans interrompre le service global.
- **Répartition de charge en lecture** : selon les besoins, certaines lectures peuvent être dirigées vers des Secondaries (rapports, analytics, sauvegardes, etc.).

Schéma simplifié

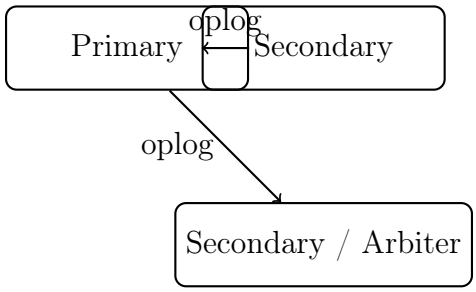

Figure 1 — Schéma simplifié d'un Replica Set MongoDB

1.4 Installation de MongoDB en mode Replica Set via Docker

Les commandes ci-dessous reprennent une mise en place simple : lancer 3 conteneurs MongoDB dans un même réseau Docker et les regrouper dans un Replica Set nommé `rs0`.

```
# Cration du rseau virtuel pour le cluster
docker network create mongo-cluster

# Lancement de 3 instances MongoDB dans ce rseau virtuel, avec des ports diffrents
# --replSet rs0 indique que l'instance fera partie du Replica Set "rs0"
docker run -d --name mongo1 --net mongo-cluster \
  -p 27017:27017 \
  mongo \
  --replSet rs0 --bind_ip_all

docker run -d --name mongo2 --net mongo-cluster \
  -p 27018:27017 \
  mongo \
  --replSet rs0 --bind_ip_all

docker run -d --name mongo3 --net mongo-cluster \
  -p 27019:27017 \
  mongo \
  --replSet rs0 --bind_ip_all

# Vrification du lancement des conteneurs
docker ps
```

Récupération de l'IP hôte (important : `localhost` ne convient pas car chaque conteneur a son propre localhost) :

```
ifconfig | grep "inet " # sur MacOS par exemple
```

Puis initialisation depuis le nœud qui jouera le rôle de Primary (ici `mongo1`) :

```
docker exec -it mongo1 mongosh
```

Dans `mongosh`, initier `rs0` avec les adresses `ip:port` :

```
rs.initiate({
  _id: "rs0",
  members: [
```

```
    { _id: 0, host: "<host_ip>:27017" },
    { _id: 1, host: "<host_ip>:27018" },
    { _id: 2, host: "<host_ip>:27019" }
  ]
})
```

Si l'IP a changé, on peut mettre à jour la configuration :

```
cfg = rs.conf()
cfg.members[0].host = "<new_host_ip>:27017"
cfg.members[1].host = "<new_host_ip>:27018"
cfg.members[2].host = "<new_host_ip>:27019"
rs.reconfig(cfg, { force: true })
```

Vérifier l'état des membres :

```
rs.status().members.map(m => ({ name: m.name, stateStr: m.stateStr }))
```

1.5 Connexion au Replica Set

On peut se connecter au cluster de deux manières :

```
# Connexion directe  un noeud (ici le Primary prsum sur 27017)
mongosh "mongodb://<host_ip>:27017/?directConnection=true"

# Connexion au Replica Set complet rs0 (recommand en production)
mongosh "mongodb://<host_ip>:27017,<host_ip>:27018,<host_ip>:27019/?replicaSet=rs0
    "

# Exemple sur une machine donne
mongosh "mongodb://192.0.0.2:27017,192.0.0.2:27018,192.0.0.2:27019/?replicaSet=rs0
    "
```

Le driver MongoDB ou `mongosh` détecte automatiquement le Primary et dirige les écritures vers ce nœud. En cas de bascule, il se reconnecte au nouveau Primary sans modifier la chaîne de connexion.

2 Questions / Réponses sur la réplication MongoDB

Partie 1 — Compréhension de base

1. **Qu'est-ce qu'un Replica Set dans MongoDB ?**
   Un Replica Set est un ensemble de processus `mongod` qui maintiennent la même base en répliquant un journal d'opérations (oplog). Il apporte redondance et disponibilité via un Primary, des Secondaries et éventuellement des Arbiters.

2. **Quel est le rôle du Primary ?**
   Le Primary reçoit **toutes** les écritures, les applique, les inscrit dans l'oplog ; les Secondaries les rejouent. Par défaut, les lectures partent aussi sur le Primary (`readPreference: "primary"`).

3. **Quel est le rôle essentiel des Secondaries ?**
   Les Secondaries appliquent l'oplog et gardent une copie quasi à jour. Ils peuvent être élus Primary, et servir des lectures si la politique de lecture le permet (`"secondary"`, `"secondaryPreferred"`, etc.).

4. **Pourquoi MongoDB n'autorise-t-il pas les écritures sur un Secondary ?**
   Pour éviter les conflits et garantir un seul point d'écriture : un unique flux d'opérations est produit par le Primary puis répliqué vers les Secondaries.

5. **Qu'appelle-t-on "cohérence forte" dans MongoDB ?**
   On parle de cohérence forte lorsqu'une lecture reflète les dernières écritures confirmées, typiquement via lecture sur Primary et un couple `readConcern` adapté (`"majority"` ou `"linearizable"`) combiné à un `writeConcern` approprié (souvent `"majority"`).

6. **Différence entre `readPreference: "primary"` et `"secondary"` ?**
   `"primary"` : lectures sur le Primary (cohérence maximale, charge concentrée).
   `"secondary"` : lectures sur les Secondaries (décharge le Primary, mais risque de données légèrement en retard).

7. **Quand lire sur un Secondary malgré les risques ?**
   Quand une faible obsolescence est acceptable : rapports, analytics, exports, sauvegardes, batch, ou lectures géolocalisées proches d'un Secondary.

Partie 2 — Commandes & configuration

8. **Commande d'initialisation d'un Replica Set :**

```
docker exec -it mongo1 mongosh
```

```
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "<host_ip>:27017" },
    { _id: 1, host: "<host_ip>:27018" },
    { _id: 2, host: "<host_ip>:27019" }
  ]
})
```

9. **Ajouter un nœud au Replica Set (ex : `mongo4`) :**

```
# 1) Lancer un nouveau conteneur
docker run -d --name mongo4 --net mongo-cluster \
  -p 27020:27017 \
  mongo \
  --replSet rs0 --bind_ip_all

# 2) Depuis le Primary, l'ajouter
docker exec -it mongo1 mongosh
```

```
rs.add("<host_ip>:27020")
```

10. **Afficher l'état actuel du Replica Set :**
`rs.status()` dans `mongosh`. Version Docker :

```
docker exec -it mongo1 mongosh --eval "rs.status()"
```

11. **Identifier le rôle (Primary / Secondary / Arbiter) d'un nœud :**
Localement : `db.isMaster()` (ou `rs.isMaster()`). Globalement : `rs.status().members` et `stateStr`.

```
docker exec -it mongo1 mongosh --eval "db.isMaster()"
docker exec -it mongo2 mongosh --eval "db.isMaster()"
```

12. **Forcer un basculement du Primary (step down) :**

```
docker exec -it mongo1 mongosh --eval "rs.stepDown(60)"
```

13. **Déclarer un arbitre, et pourquoi :**

```
# 1) Lancer un conteneur arbitre
docker run -d --name mongo-arb --net mongo-cluster \
  mongo \
  --replSet rs0 --bind_ip_all

# 2) Ajouter l'arbitre au Replica Set
docker exec -it mongo1 mongosh --eval 'rs.addArb("mongo-arb:27017")'
```

Un arbitre ne stocke pas de données : il sert à atteindre une majorité lors des votes.

14. **Configurer un Secondary retardé (`slaveDelay`) :**

```
docker run -d --name mongo-delayed --net mongo-cluster \
  -p 27021:27017 \
  mongo \
  --replSet rs0 --bind_ip_all

docker exec -it mongo1 mongosh
```

```
rs.add({
  host: "mongo-delayed:27017",
  priority: 0,
```

```
  hidden: true,
  slaveDelay: 120
})
```

Partie 3 — Résilience et tolérance aux pannes

15. **Si le Primary tombe et qu'il n'y a pas de majorité : que se passe-t-il ?**
Sans majorité, aucun nouveau Primary n'est élu : les écritures échouent ; seules des lectures sur Secondaries sont possibles (si autorisées).

16. **Comment MongoDB choisit-il un nouveau Primary ?**
Éligibilité dans la majorité de votes, puis `priority` la plus haute ; à égalité, nœud le plus à jour (oplog le plus avancé).

17. **Définir une élection.**
Mécanisme automatique de vote : un candidat sollicite les votes des membres et devient Primary s'il obtient la majorité.

18. **Auto-dégradation : définition et cas typique.**
Un Primary se rétrograde en Secondary s'il perd la majorité (ex : partition réseau) afin d'éviter un split-brain.

19. **Pourquoi un nombre impair de nœuds ?**
Pour faciliter l'obtention d'une majorité et limiter les situations bloquantes.

20. **Conséquences d'une partition réseau ?**
La partition majoritaire garde/élit un Primary ; les partitions minoritaires deviennent en lecture seule (pas de Primary).

Partie 4 — Scénarios pratiques

21. **3 nœuds : Primary (27017), Secondary (27018), Arbiter (27019). Si le Primary devient injoignable ?**
Secondary + Arbiter = majorité (2 votes sur 3), donc le Secondary peut être élu Primary.

22. **Secondary avec `slaveDelay = 120` secondes : utilité ?**
Il applique les opérations avec 120 secondes de retard (rattrapage d'erreurs humaines, audit, analyse "dans le passé proche"). À éviter pour des lectures critiques.

23. **Lecture toujours à jour même en cas de bascule : que recommander ?**
`writeConcern: "majority"`, `readPreference: "primary"`, `readConcern: "majority"` (ou `"linearizable"`).

24. **Écriture confirmée par au moins deux nœuds : quel `writeConcern` ?**
`{ w: 2 }` ou `"majority"` (sur un RS à 3 membres).

25. **Lecture obsolète depuis un Secondary : pourquoi et comment éviter ?**
Réplication asynchrone ⇒ retard possible. Pour éviter : lire sur Primary, `readConcern: "majority"`, surveiller le lag.

26. **Commande pour voir quel nœud est Primary :**

```
docker exec -it mongo1 mongosh --eval \
```

```
'rs.status().members.map(m => ({ name: m.name, stateStr: m.stateStr }))'
```

27. Forcer une bascule manuelle du Primary (avec impact limité) :

```
docker exec -it mongo1 mongosh --eval "rs.printSlaveReplicationInfo()"
docker exec -it mongo1 mongosh --eval "rs.stepDown(60)"
```

28. Ajouter un Secondary (procédure complète) :

```
docker run -d --name mongo4 --net mongo-cluster \
  -p 27020:27017 \
  mongo \
  --replSet rs0 --bind_ip_all

docker exec -it mongo1 mongosh
```

```
rs.add("mongo4:27017")
```

29. Retirer un nœud défectueux :

```
docker exec -it mongo1 mongosh --eval 'rs.remove("mongo4:27017")'
docker stop mongo4
docker rm mongo4
```

30. Rendre un Secondary caché (hidden) et pourquoi :

```
cfg = rs.conf()
cfg.members[2].hidden = true
cfg.members[2].priority = 0
rs.reconfig(cfg)
```

Utile pour réserver un nœud à des usages internes (sauvegardes/analytics) sans qu'il soit choisi pour servir des lectures.

31. Augmenter la priorité d'un nœud (Primary préféré) :

```
cfg = rs.conf()
cfg.members[0].priority = 10
cfg.members[1].priority = 1
cfg.members[2].priority = 0
rs.reconfig(cfg)
```

32. Vérifier le délai de réplication d'un Secondary :

```
docker exec -it mongo1 mongosh --eval "rs.printSlaveReplicationInfo()"
```

33. Que fait `rs.freeze()` et quand l'utiliser ?
`rs.freeze(N)` empêche un nœud d'être élu Primary pendant `N` secondes (utile pour contrôler une élection).

```
docker exec -it mongo2 mongosh --eval "rs.freeze(300)"
```

34. **Redémarrer un Replica Set sans perdre la configuration :**
La configuration est stockée dans la base `local`. Tant que le `dbPath` est conservé et que `-replSet` reste identique, la configuration persiste.

```
docker restart mongo1
docker restart mongo2
docker restart mongo3
```

35. **Surveiller la réplication en temps réel :**
Commandes : `rs.status()`, `rs.printSlaveReplicationInfo()`, `db.printReplicationInfo()`.
Logs :

```
docker logs -f mongo1
docker exec -it mongo1 mongosh --eval "rs.printSlaveReplicationInfo()"
```

Questions complémentaires

37. **Qu'est-ce qu'un Arbiter et pourquoi ne stocke-t-il pas de données ?**
Un Arbiter vote mais ne stocke rien, ne devient pas Primary, et sert uniquement à obtenir une majorité.

38. **Comment vérifier la latence de réplication Primary → Secondaries ?**
`rs.printSlaveReplicationInfo()`.

39. **Commande pour afficher le retard de réplication :**
`rs.printSlaveReplicationInfo()`.

40. **Réplication asynchrone vs synchrone : quel type utilise MongoDB ?**
MongoDB réplique surtout en asynchrone, avec contrôle via `writeConcern`.

41. **Peut-on modifier la configuration sans redémarrer ?**
Oui, via `rs.conf()` et `rs.reconfig()`.

42. **Secondary en retard de plusieurs minutes : conséquences ?**
Retard important : pas/peu éligible comme Primary, et risque de resynchronisation complète si l'oplog ne suffit plus.

43. **Gestion des conflits de données pendant la réplication ?**
Un seul nœud accepte les écritures (Primary), ce qui évite les conflits. Cas extrêmes : rollback.

44. **Peut-on avoir plusieurs Primary en même temps ? Pourquoi ?**
En fonctionnement normal non : élections + majorité empêchent le split-brain.

45. **Pourquoi éviter les écritures sur Secondary même si on lit dessus ?**
Les écritures applicatives sur Secondary sont interdites par les drivers et seraient incohérentes : la réplication risquerait d'écraser ces changements.

46. **Conséquences d'un réseau instable sur un Replica Set ?**
Élections fréquentes, *step-down* répétés, erreurs d'écriture, augmentation du retard de réplication, risque accru de rollback.

Fin du document