Rapport TP2 - partie 1

Réplication dans un système maître-esclave

Addou Aicha Amira

Bombay Marc

3ème Année Ingénieur - INFO - Apprentissage

1. Introduction

La réplication est un mécanisme fondamental dans les systèmes distribués. Elle permet de maintenir plusieurs copies synchronisées d'un même ensemble de données afin d'assurer :

- **la tolérance aux pannes**,
- **la haute disponibilité**,
- **la continuité de service**,
- **la fiabilité des lectures**.

Dans ce rapport, nous décrivons **pas à pas** :

1. le principe général de la réplication ;
2. son fonctionnement dans une architecture maître–esclave ;
3. sa mise en œuvre pratique dans MongoDB à travers la création d'un Replica Set.

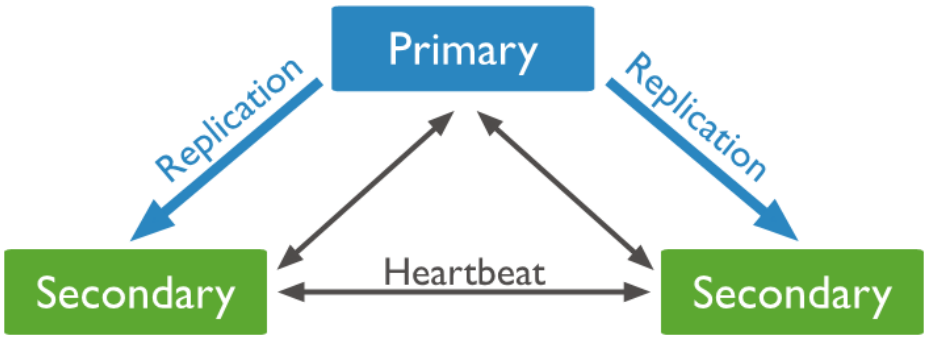
2. Principe général de la réplication

2.1 Communication permanente entre les nœuds

Dans un cluster distribué, tous les nœuds sont interconnectés. Chaque nœud envoie continuellement des messages aux autres afin de :

- vérifier leur état (heartbeats),
- propager les mises à jour,
- confirmer la bonne réception des opérations,
- maintenir la cohérence globale.

Cette communication ininterrompue garantit que l'ensemble reste synchronisé et réactif.



2.2 Gestion de la panne d'un nœud secondaire

Dans une architecture maître–esclave, le maître surveille l'état des esclaves.

Si un esclave tombe en panne :

- le maître détecte l'absence de réponse,
- il marque le nœud comme « inactif »,
- Il réaffecte les tâches à d'autres nœuds disponibles.

Ce mécanisme participe à la **tolérance aux pannes**.

2.3 Gestion de la panne du maître

Si le maître tombe en panne, le système ne peut plus coordonner les écritures. Une **élection automatique** est alors déclenchée via un algorithme de consensus distribué (ex. Paxos, Raft).

Étapes :

1. les nœuds restants échangent leur état ;
2. un vote est organisé ;
3. un nouveau maître est élu.

Ainsi, le système continue à fonctionner sans intervention humaine.

2.4 Le risque de partition réseau (split-brain)

Lorsqu'un cluster est divisé en deux groupes qui ne peuvent plus communiquer (à cause d'une panne réseau ou de la panne du maître), chaque groupe peut croire que l'autre est hors service.

Dans ce cas, chaque sous-groupe pourrait tenter d'élire un nouveau maître, ce qui risquerait de créer deux maîtres simultanés -> incohérence.

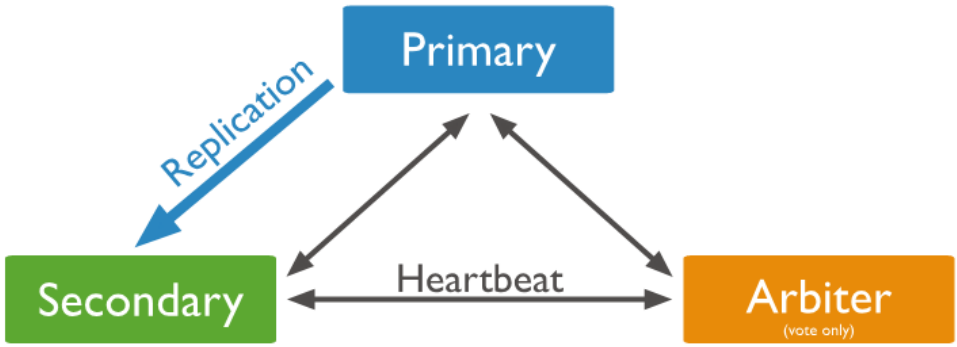
Solution classique (utilisée par MongoDB) :

- **seul le groupe qui détient la majorité des nœuds peut élire un maître**,
- l'autre groupe devient inactif.

Toutefois, si les deux groupes contiennent respectivement le même nombre de nœuds (pas de majorité absolue), cette règle ne fonctionne plus, et aucun maître ne peut être elu.

Pour pallier ce problème, il est courant d'ajouter un **arbitre**.

2.5 Rôle de l’arbitre dans un Replica Set



Un arbitre est un nœud particulier d’un Replica Set MongoDB qui **ne stocke aucune donnée**, mais participe uniquement au processus de vote lors de l’élection d’un maître.

Son rôle principal est d’éviter les situations où aucun groupe n’atteint la majorité, par exemple lorsque le cluster est divisé en deux sous-groupes de même taille.

L’arbitre ajoute **un vote supplémentaire** permettant de départager les deux groupes.

Ainsi, même si les nœuds sont répartis équitablement, le groupe contenant l’arbitre obtient automatiquement la majorité, et peut donc élire un maître.

3. La réplication dans MongoDB

MongoDB repose sur une architecture **primaire / secondaire**, équivalente à maître–esclave.

3.1 Rôle du primaire

Le nœud primaire :

- reçoit **toutes les écritures**,
- reçoit la majorité des lectures,
- enregistre chaque opération dans un journal `oplog`,
- propage les modifications aux secondaires.

Les écritures sont **centralisées** pour éviter tout conflit.

3.2 Rôle des secondaires

Les nœuds secondaires :

- répliquent les données du primaire,
- sont en lecture seule par défaut,
- peuvent être lus **uniquement si la préférence de lecture est modifiée**.

Remarque :

MongoDB utilise une **réplication asynchrone** : Le secondaire peut être **en retard** par rapport au primaire.

3.3 Gestion de la cohérence

Par défaut, MongoDB applique une **cohérence forte** : toutes les lectures et écritures sont effectuées sur le nœud primaire. Cela garantit que chaque lecture renvoie toujours la version la plus récente des données.

Il est toutefois possible d'autoriser les lectures sur les nœuds secondaires pour répartir la charge et améliorer les performances. Cependant, comme la réplication est asynchrone, les secondaires peuvent avoir un léger retard. Une lecture effectuée sur un secondaire peut donc retourner une donnée non mise à jour.

Ce mécanisme illustre un compromis classique des systèmes distribués : **plus de performance et de scalabilité**, mais **moins de cohérence stricte**.

4. Mise en place d’un Replica Set MongoDB : étapes pratiques

Dans cette partie, nous reproduisons la manipulation réalisée dans la vidéo pour créer un Replica Set de trois nœuds, totalement sur une seule machine.

Nous utiliserons :

- 3 terminaux serveur (nœuds),
- 1 terminal client,
- 3 ports différents,
- un répertoire par serveur.

4.1 Préparation de l'environnement

- Création des répertoires de données

```
mkdir d1
mkdir d2
mkdir d3
```

- Nom du Replica Set :

```
“rs0”
```

- Ports utilisés :

```
27018
27019
27020
```

4.2 Démarrage des trois nœuds MongoDB

Terminal 1 :

```
docker exec -it mongo1 mongod --replSet rs0 --port 27018
--dbpath d1
```

Terminal 2 :

```
docker exec -it mongo1 mongod --replSet rs0 --port 27019
--dbpath d2
```

Terminal 3 :

```
docker exec -it mongo1 mongod --replSet rs0 --port 27020
--dbpath d3
```

Chaque instance signale : "Replica set not initialized" - ce qui est normal à ce stade.

4.3 Initialisation du Replica Set

Dans un quatrième terminal :

```
docker exec -it mongo1 mongosh --port 27018
```

Puis :

```
rs.initiate()
```

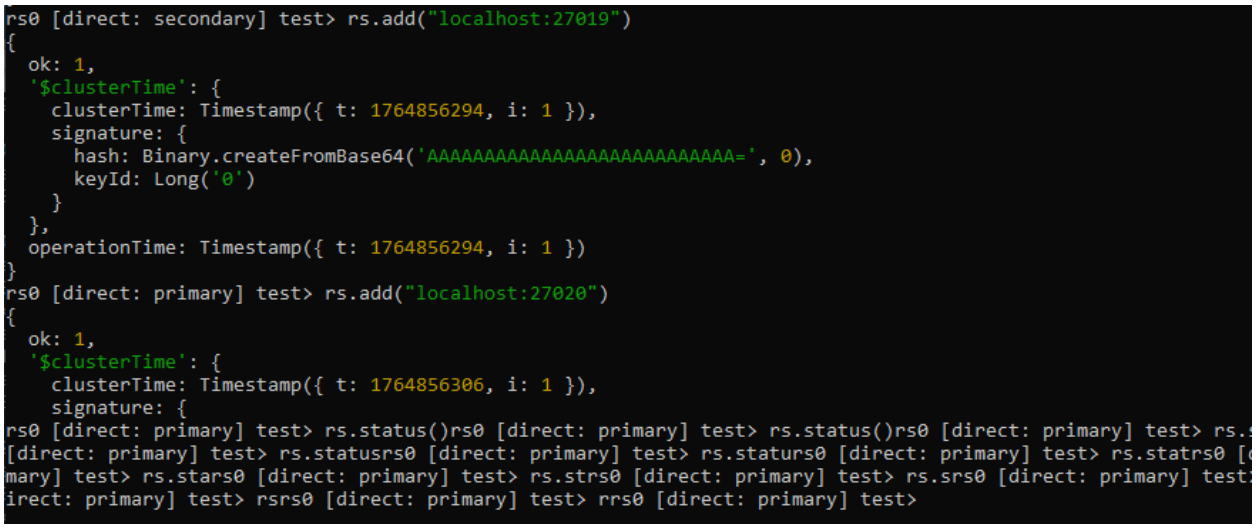
MongoDB crée alors automatiquement un premier nœud **primaire**.

```
test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: 'localhost:27018',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764856268, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764856268, i: 1 })
}
rs0 [direct: secondary] test> rs.add("localhost:27019")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764856294, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764856294, i: 1 })
}
rs0 [direct: primary] test> rs.add("localhost:27020")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764856306, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764856306, i: 1 })
}
```

4.4 Ajout des secondaires au cluster

Toujours dans le shell Mongo :

```
rs.add("localhost:27019")
rs.add("localhost:27020")
```



MongoDB commence immédiatement la réplication.

4.5 Vérification de la configuration

1. Configuration statique

`rs.config()`

```
rs0 [direct: primary] test> rs.config()
{
  _id: 'rs0',
  version: 7,
  term: 1,
  members: [
    {
      _id: 0,
      host: 'localhost:27018',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'localhost:27019',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 2,
      host: 'localhost:27020',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ],
  protocolVersion: Long('1'),
  writeConcernMajorityJournalDefault: true,
  settings: {
    chainingAllowed: true,
    heartbeatIntervalMillis: 2000,
    heartbeatTimeoutSecs: 10,
    electionTimeoutMillis: 10000,
    catchUpTimeoutMillis: -1,
    catchUpTakeoverDelayMillis: 30000,
    getLastErrorModes: {},
    getLastErrorDefaults: { w: 1, wtimeout: 0 },
    replicaSetId: ObjectId('693191cc09c8b19645608e50')
  }
```

On y trouve :

Champ	Signification
`_id`	identifiant interne du nœud
`host`	adresse + port
`priority`	capacité à devenir primaire
`votes`	participation à l'élection
`slaveDelay`	retard volontaire
`members`	liste des membres

2. État dynamique en temps réel

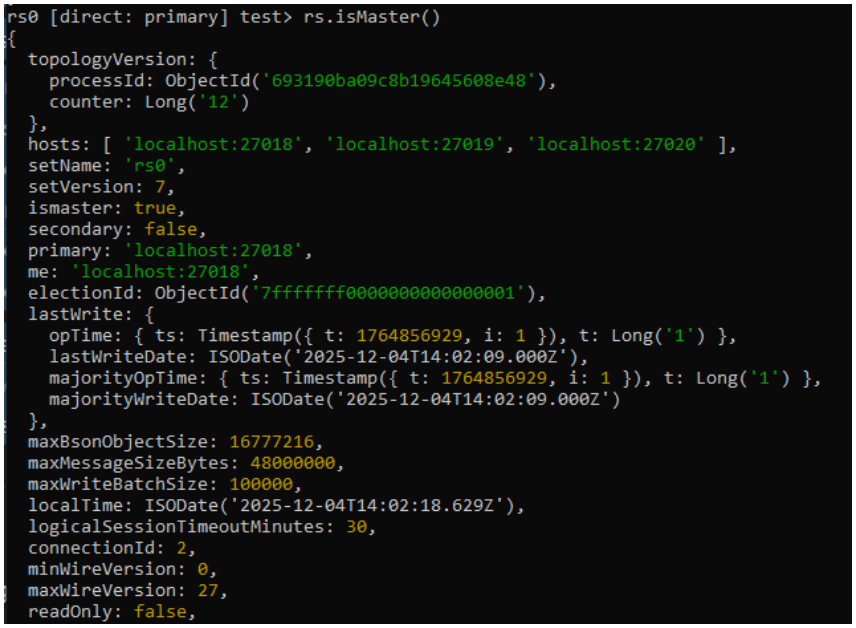
```
rs.status()
```

Informations obtenues :

- quel nœud est primaire,
- quel nœud est secondaire,
- état de santé [notamment: le serveur est-il joignable ?],
- temps de fonctionnement,
- date de dernière opération synchronisée.
- etc.

3. Vérifier si le nœud est maître

```
db.isMaster()
```



`db.isMaster()` renvoie un objet contenant, entre autres :

Champ	Signification
`ismaster`	true ou false selon si c'est un primaire ou pas
`setName`	nom du set
`hosts`	liste des membres
`primary`	l'adresse du Primary [maître]
`me`	l'adresse du serveur depuis lequel on a appelé

4.6 Lecture sur les secondaires

Par défaut, interdit.
Pour autoriser :

```
rs.slaveOk()
```

Comme nous avons téléchargé une version plus récente de mongodb nous avons utilisé la commande suivante:

```
db.getMongo().setReadPref("secondary")
```

4.7 Ajout d’un arbitre

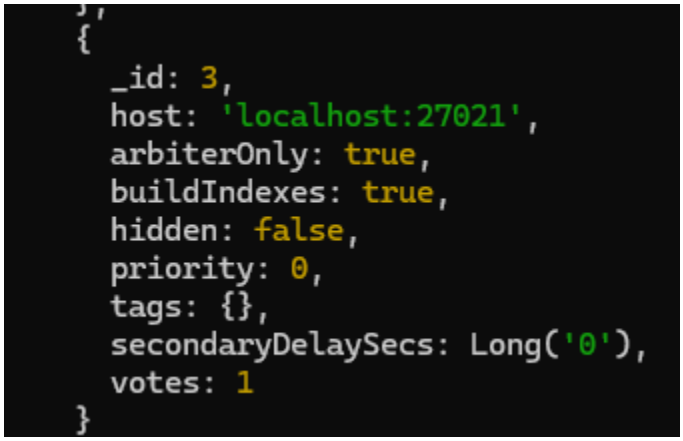
Lancer un arbitre :

après avoir créé un dossier darb,

```
mongod --replSet rs0 --port 27021 --dbpath darb
```

Puis, dans un shell mongo :

```
rs.addArb("localhost:27021")
```



Résultat de `rs.config()` montrant l’ajout de `localhost:27021` en arbitre (`arbiterOnly: true`)

4.8 Simulation d’une panne

- **Arrêter le primaire.**

un secondaire devient primaire après une élection

Essayer d’écrire sur un secondaire

```
rs0 [direct: secondary] lesfilms> db.films.drop()
MongoServerError[NotWritablePrimary]: not primary
```

Reconnexion au nouveau primaire

L'écriture fonctionne :

```
rs0 [direct: primary] lesfilms> db.films.drop()
true
```

5. Conclusion

Dans cette première partie, nous avons :

- rappelé le fonctionnement fondamental de la réplication dans les systèmes distribués,
- expliqué le modèle primaire–secondaire de MongoDB,
- mis en place un Replica Set complet comprenant 3 nœuds (et éventuellement un arbitre),
- observé la synchronisation, les élections, et la tolérance aux pannes.

Cette expérimentation permet de comprendre de manière concrète comment MongoDB assure la **cohérence**, la **haute disponibilité**, et la **résilience** grâce à son système de réplication.