Tutoriel : Mise en place d'un Replica Set MongoDB

Ce tutoriel a pour objectif de vous aider à comprendre et à mettre en place une grappe de serveurs (cluster) MongoDB, appelée "Replica Set". Cette architecture est fondamentale pour assurer la haute disponibilité et la tolérance aux pannes de vos données.

Partie 1 : Concepts Fondamentaux de la Réplication

Avant de passer à la pratique, il est crucial de comprendre les principes théoriques qui régissent un système distribué comme MongoDB.

1. Interconnexion et Communication Constante

Dans une grappe de serveurs, tous les nœuds sont connectés et échangent des messages en permanence ("heartbeats"). Cette communication constante est vitale pour :

- Répliquer les données.
- Transmettre des confirmations d'écriture.
- Vérifier l'état de santé de chaque nœud. Cela permet au système de rester cohérent et réactif.

2. Architecture Maître/Esclave (Primaire/Secondaire)

MongoDB utilise une architecture de type Maître/Esclave, terminologie actuellement remplacée par **Primaire/Secondaire (Primary/Secondary)**.

- **Si un Secondaire (un esclave) tombe en panne :** Le Primaire (le maitre)s'en aperçoit rapidement grâce à l'absence de réponse aux messages

réguliers. Il marque le nœud comme inactif et réaffecte la charge si nécessaire.

3. Élection Automatique en cas de panne du Primaire

Le point critique est la panne du Maître (Primaire), car c'est lui qui centralise les écritures et la coordination.

- **Le problème :** Le système ne peut plus fonctionner normalement.
- **La solution :** Un processus d'élection automatique (algorithme distribué type Paxos ou Raft) est déclenché. Les nœuds restants négocient pour désigner un nouveau Primaire sans intervention humaine.

4. Le risque de "Split-Brain" et la Règle de Majorité

Imaginez que le réseau se coupe en deux groupes qui ne peuvent plus communiquer.

- **Le risque :** Chaque groupe pourrait croire que l'autre est en panne et tenter d'élire son propre Maître. On se retrouverait avec deux Maîtres simultanés, entraînant une incohérence des données inacceptable.
- **La solution (adoptée par MongoDB) :** Seul le groupe possédant la **majorité absolue** des nœuds est autorisé à élire un Maître et à continuer de fonctionner. L'autre groupe passe en mode inactif (lecture seule ou déconnecté) jusqu'au rétablissement du réseau.

5. Le Rôle de l'Arbitre : Assurer la Majorité

La règle de majorité décrite ci-dessus pose problème si vous avez un nombre pair de nœuds de données.

- **Le scénario problématique :** Imaginez un cluster avec seulement deux nœuds (1 Primaire et 1 Secondaire). Si le Primaire tombe en panne, le Secondaire restant ne représente que 50% des voix. Il n'a pas la majorité

absolue (>50%). Il ne peut donc pas s'élire lui-même Primaire, et le cluster devient inaccessible en écriture.

- **La solution : L'Arbitre.** Pour garantir qu'une majorité puisse toujours être atteinte sans avoir à ajouter un coûteux troisième serveur de données, MongoDB permet l'ajout d'un nœud spécial appelé "Arbitre".
- **Qu'est-ce qu'un Arbitre ?** C'est une instance MongoDB légère qui :
    1. **Participe aux votes** lors d'une élection.
    2. **Ne stocke aucune donnée** (très peu de ressources CPU/Disque nécessaires).
    3. **Ne peut jamais être élu Primaire**.
- **Son utilité :** Son seul rôle est d'apporter une voix supplémentaire pour briser les égalités et assurer qu'un nombre impair de votants est disponible, permettant ainsi d'atteindre une majorité même dans de petites configurations.

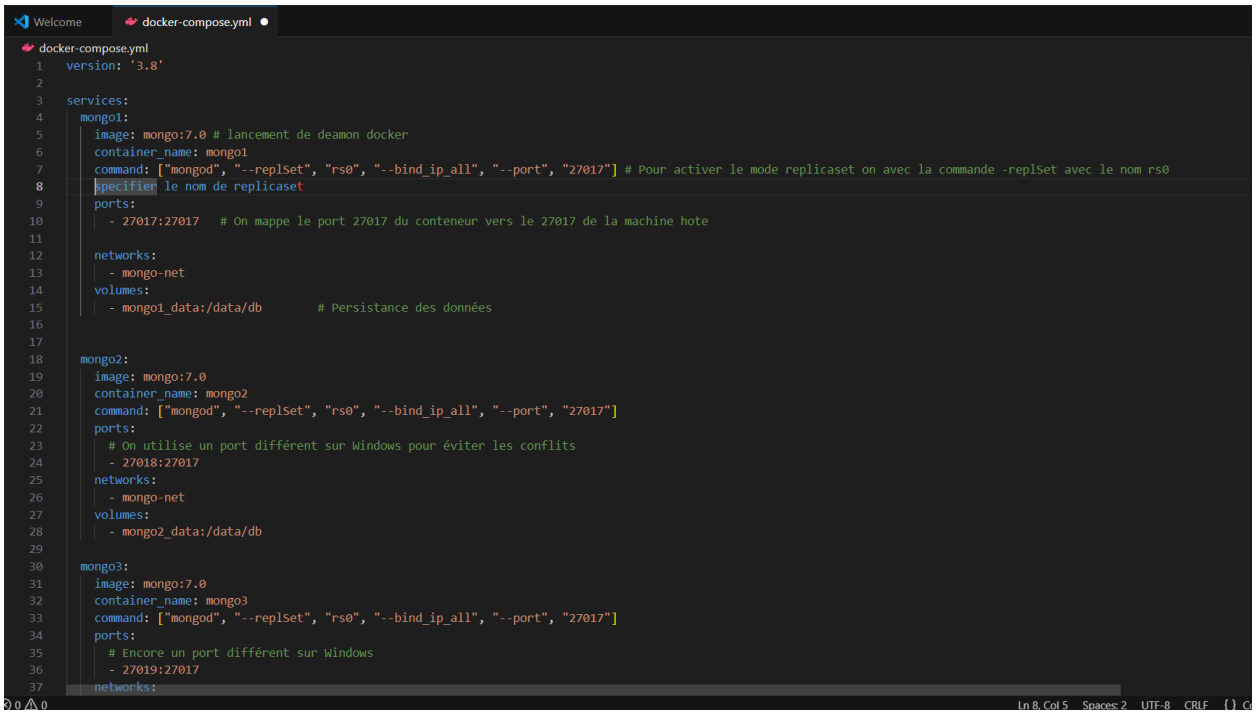
5. Spécificités de MongoDB

- **Rôle de la Réplication :** La réplication sert principalement à la **tolérance aux pannes (Haute Disponibilité)**, et *non* à la montée en charge des écritures (pour cela, on utilise le Sharding/Partitionnement).
- **Flux d'écriture :** Les écritures se font **toujours sur le nœud Primaire** pour garantir la cohérence et gérer les conflits.
- **Flux de lecture et Cohérence :**
    - Par défaut, les lectures se font aussi sur le Primaire pour assurer une **cohérence forte** (on lit toujours la dernière version de la donnée).
    - Il est possible de configurer des lectures sur les Secondaires pour répartir la charge, mais cela introduit un risque de lire des données obsolètes, car la réplication est **asynchrone** (il y a un léger délai entre l'écriture sur le primaire et sa copie sur les secondaires).
- **Mécanisme d'écriture :** Le Primaire écrit d'abord dans un journal séquentiel (Oplog) pour la performance, envoie un acquittement au client, puis la donnée est répliquée vers les Secondaires.

Partie 2:manipulation de tp Mongo replica sets

Le but de ce Tp est de de simuler une architecture Maître-Esclave (Primary-Secondary) sur une seule machine en utilisant différents ports.

Conformément à l'objectif de simuler une grappe de serveurs (cluster), nous avons instancié trois conteneurs MongoDB (`mongo1`, `mongo2`, `mongo3`) via **Docker Compose**.



1) Définition de l'architecture avec Docker compose

Le fichier `docker-compose.yml` agit comme un véritable plan d'architecte, permettant de simuler un **Replica Set** complet sur une seule machine physique. Il orchestre le lancement de trois conteneurs indépendants mais interconnectés, configurés pour fonctionner en équipe grâce aux mécanismes suivants :

- **Isolation et Persistance des Données :** La bonne pratique évoquée dans la vidéo, consistant à séparer physiquement les données (création manuelle de `disk1`, `disk2`...), est ici assurée par l'utilisation de **volumes Docker nommés** (`mongo1_data`, etc.). Cette approche garantit une persistance fiable des données et des journaux de transaction (logs), éléments cruciaux pour la récupération séquentielle en cas de panne.
- **Interconnexion et Réseau :** L'option de démarrage `--replSet rs0` est injectée à chaque conteneur pour indiquer son appartenance à la grappe "rso", bien que la communication effective ne débute qu'après l'étape d'initialisation. L'architecture réseau repose sur une astuce de **port mapping** : alors que les nœuds communiquent entre eux sur un réseau privé Docker via leur port standard sans conflit, ils sont exposés sur la machine hôte (Windows) via des ports distincts (27017, 27018, 27019), permettant ainsi une administration externe ciblée.

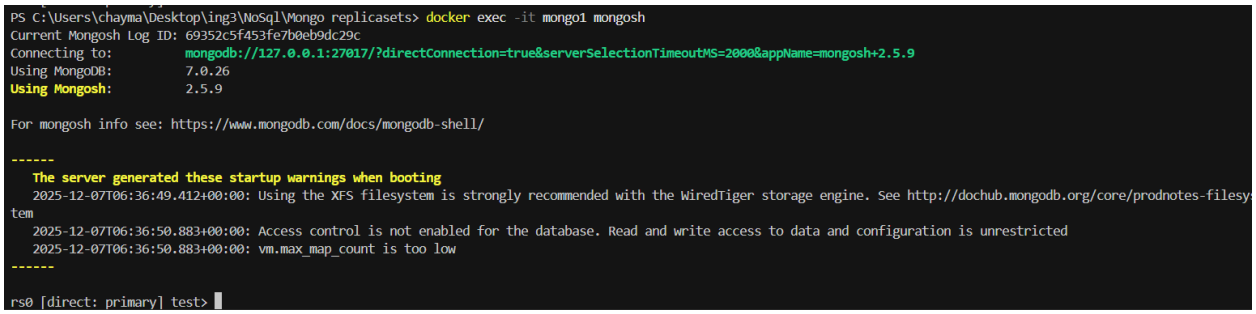
```
test> rs.initiate({
...   _id: "rs0",
...   members: [
...     { _id: 0, host: "mongo1:27017" },
...     { _id: 1, host: "mongo2:27017" },
...     { _id: 2, host: "mongo3:27017" }
...   ]
... })
{ ok: 1 }
rs0 [direct: other] test>
```

2) Initialisation et Configuration du Replica Set

Une fois les conteneurs lancés via Docker Compose, les instances MongoDB sont fonctionnelles mais isolées : elles ne "savent" pas encore qu'elles appartiennent à la même équipe. L'étape d'initialisation est donc indispensable pour activer la communication inter-nœuds.

- **Connexion au conteneur principal**

Contrairement à une installation locale où l'accès se fait via un terminal classique, nous devons ici pénétrer directement dans le conteneur maître pour administrer la grappe. Nous utilisons la commande suivante pour ouvrir le shell MongoDB moderne (`mongosh`, remplaçant l'outil déprécié `mongo`) :



- **Configuration de la grappe (Cluster)**

L'initialisation est réalisée via la commande `rs.initiate()`. Contrairement à la démonstration vidéo qui procède par étapes (initialisation puis ajouts successifs avec `rs.add`), nous avons opté pour l'injection immédiate de la configuration complète.

Adaptation pour l'environnement Docker : Cette méthode est cruciale dans un environnement conteneurisé pour deux raisons :

1. **Adressage réseau :** Dans la vidéo, les nœuds communiquent via `localhost` sur des ports différents. Dans notre architecture Docker, les conteneurs communiquent via leurs **noms de service** (`mongo1`, `mongo2`, `mongo3`) sur le port interne standard (`27017`).
2. **Stabilité :** Définir la configuration complète dès le départ force l'utilisation de ces noms de service persistants. Cela empêche MongoDB

de s'auto-configurer avec une adresse IP interne Docker aléatoire, ce qui briserait la configuration du cluster à chaque redémarrage des conteneurs.

-> Dès l'exécution de la commande, le Replica Set est officiellement créé. Les nœuds entament une phase de "négociation" via un processus d'élection (basé sur des algorithmes de consensus comme **Paxos** ou **Raft**). Ce processus désigne automatiquement un nœud **Primary** (Maître) responsable des écritures, et bascule les autres en **Secondary** (Esclaves), assurant ainsi la haute disponibilité du système.

3) Surveillance et Monitoring

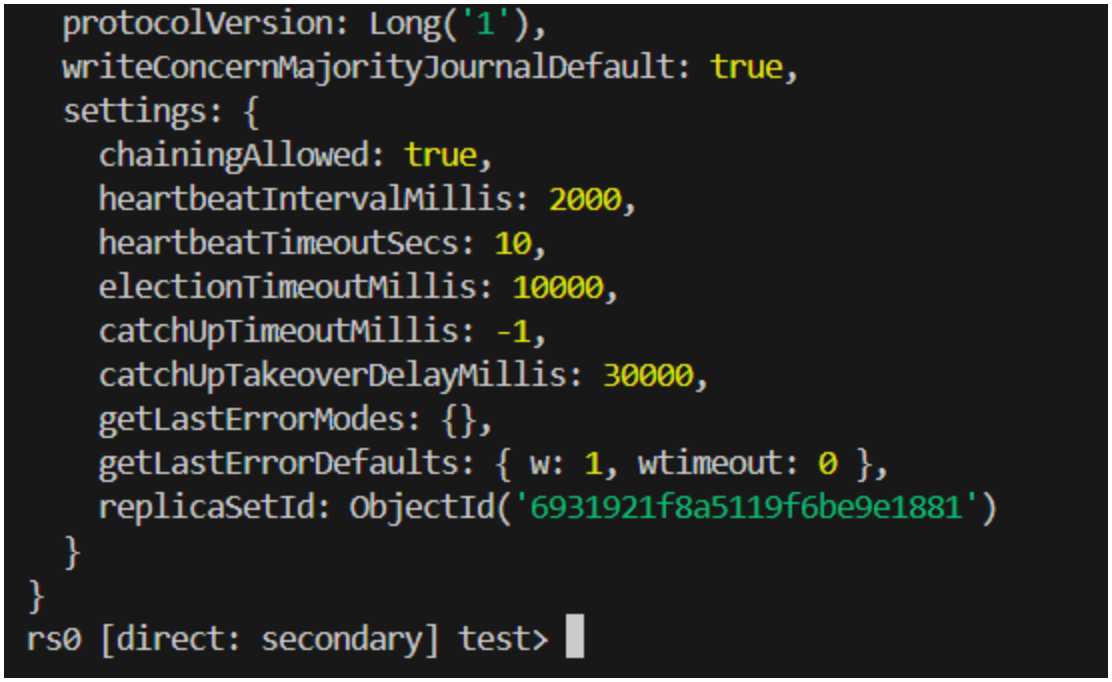
Nous avons utilisé plusieurs commandes pour vérifier l'état de la grappe, conformément aux recommandations de diagnostic de la vidéo.

A. Configuration Statique (`rs.conf()`)

Cette commande nous a permis de voir la configuration "froide" du cluster.

- **Observation :** On y retrouve la liste des membres (`members`), leur `_id`, et surtout leur `priorité` et `votes`.
- **Remarque :** Comme souligné, ces paramètres sont modifiables. Par exemple, mettre `priority: 0` empêcherait un nœud de devenir Primary, et `votes: 0` l'empêcherait de participer à l'élection (utile pour les nœuds passifs).

```
rs0 [direct: secondary] test> rs.conf()
{
  _id: 'rs0',
  version: 1,
  term: 4,
  members: [
    {
      _id: 0,
      host: 'mongo1:27017',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'mongo2:27017',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 2,
      host: 'mongo3:27017',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ],
  protocolVersion: Long('1'),
```



B. État Dynamique (`rs.status()`)

C'est la commande la plus critique pour le monitoring en temps réel.

- **Information clé (Health) :** Le champ `health` (1 ou 0) indique si le nœud est en ligne.
- **Information clé (State) :** Permet d'identifier immédiatement qui est `PRIMARY` et qui est `SECONDARY`.
- **Information clé (Sync) :** Permet de voir si les esclaves ont du retard (replication lag) sur le maître via l'`optime`.

C. Identification du Rôle (`rs.hello()` ou `db.isMaster()`)

Cette commande est utilisée côté client pour savoir à qui l'on parle.

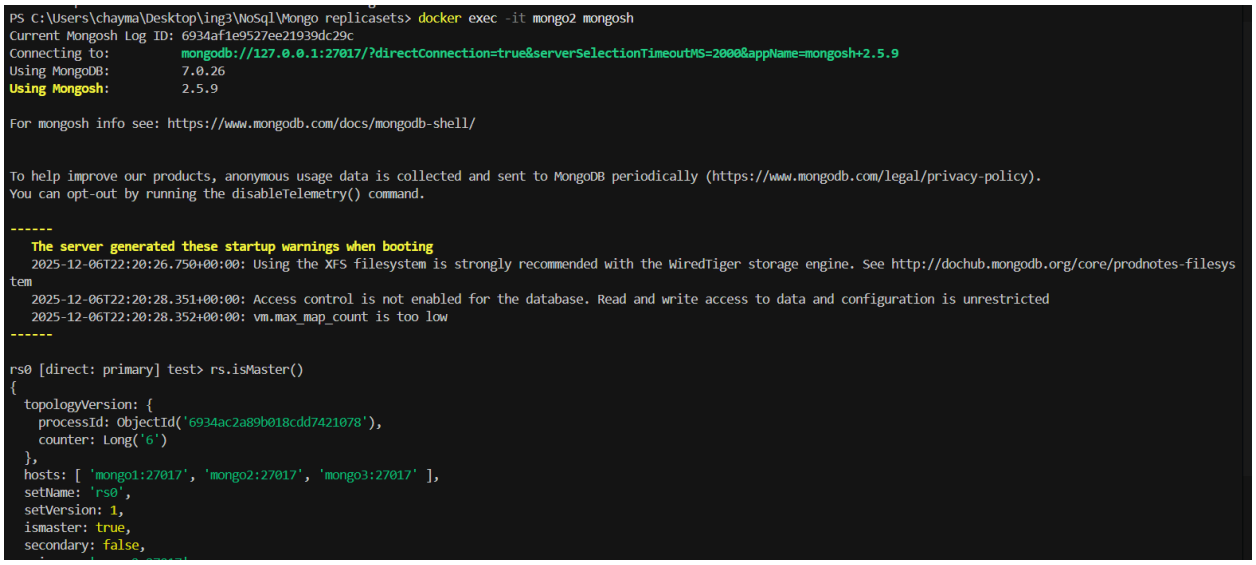
- **Remarque :** C'est essentiel car, par défaut, **MongoDB n'autorise les écritures que sur le Primary** pour garantir une cohérence forte des données.

```
rs0 [direct: secondary] test> rs.status()
{
  set: 'rs0',
  date: ISODate('2025-12-06T22:29:57.372Z'),
  myState: 2,
  term: Long('4'),
  syncSourceHost: 'mongo2:27017',
  syncSourceId: 1,
  heartbeatIntervalMillis: Long('2000'),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1765060188, i: 1 }), t: Long('4') },
    lastCommittedWallTime: ISODate('2025-12-06T22:29:48.678Z'),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1765060188, i: 1 }), t: Long('4') },
    appliedOpTime: { ts: Timestamp({ t: 1765060188, i: 1 }), t: Long('4') },
    durableOpTime: { ts: Timestamp({ t: 1765060188, i: 1 }), t: Long('4') },
    lastAppliedWallTime: ISODate('2025-12-06T22:29:48.678Z'),
    lastDurableWallTime: ISODate('2025-12-06T22:29:48.678Z')
  },
  lastStableRecoveryTimestamp: Timestamp({ t: 1765060158, i: 1 }),
  electionParticipantMetrics: {
    votedForCandidate: true,
    electionTerm: Long('4'),
    lastVoteDate: ISODate('2025-12-06T22:20:38.657Z'),
    electionCandidateMemberId: 1,
    voteReason: '',
    lastAppliedOpTimeAtElection: { ts: Timestamp({ t: 1764865287, i: 1 }), t: Long('3') },
    maxAppliedOpTimeInSet: { ts: Timestamp({ t: 1764865287, i: 1 }), t: Long('3') },
    priorityAtElection: 1,
    newTermStartDate: ISODate('2025-12-06T22:20:38.681Z'),
    newTermAppliedDate: ISODate('2025-12-06T22:20:38.713Z')
  },
  members: [
    {
      _id: 0,
      name: 'mongo1:27017',
```

4) Les noeuds workers

```
rs0 [direct: secondary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId('6934ac2a2134990206l9e906'),
    counter: Long('4')
  },
  hosts: [ 'mongo1:27017', 'mongo2:27017', 'mongo3:27017' ],
  setName: 'rs0',
  setVersion: 1,
  ismaster: false,
  secondary: true,
  primary: 'mongo2:27017',
  me: 'mongo1:27017',
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1765060268, i: 1 }), t: Long('4') },
    lastWriteDate: ISODate('2025-12-06T22:31:08.000Z'),
    majorityOpTime: { ts: Timestamp({ t: 1765060268, i: 1 }), t: Long('4') },
    majorityWriteDate: ISODate('2025-12-06T22:31:08.000Z')
  },
```

5) Le Noeud master



4. Gestion de la Réplication et Cohérence

Nous avons testé l'insertion de données sur le nœud `mongo1` (Primary).

1. **Écriture :** Les données sont écrites sur le journal du Primary puis répliquées de manière asynchrone vers les Secondaries.

2. **Lecture :** Par défaut, nous ne pouvons pas lire sur `mongo2` ou `mongo3`.
3. **Forçage de lecture :** Pour lire sur un esclave, il faut utiliser `rs.secondaryOk()` (anciennement `rs.slaveOk()`).
    - **Avertissement :** Lire sur un secondaire permet de répartir la charge (scalabilité), mais présente un risque de **cohérence éventuelle** (lire une donnée obsolète si la réplication n'est pas encore terminée).

5. Mise en place d'un Arbitre (Exercice demandé)

Comme suggéré à la fin de la démonstration pour traiter les cas de **partitionnement réseau** (où un sous-groupe de serveurs perd la connexion avec les autres), nous avons ajouté un **nœud Arbitre**.

Rôle de l'Arbitre : L'arbitre ne stocke **aucune donnée**. Son unique rôle est de participer aux votes pour permettre d'atteindre une majorité stricte lors d'une élection. Cela empêche les situations de blocage ou de "split-brain" (deux maîtres simultanés).

- **Manipulation effectuée :**

1. Ajout dans le `docker-compose.yaml` : Nous avons ajouté un service léger pour l'arbitre qui ne nécessite pas de volume de données important.

```yaml
services:
  mongo-arbiter:
    image: mongo:7.0
    container_name: mongo_arb
    command: ["mongod", "--replSet", "rs0", "--bind_ip_all", "--port", "27017"]
    networks:
      - mongo-net
```

2. Ajout au Replica Set : Depuis le shell du nœud Primary (`mongo1`), nous avons exécuté :

```
rs0 [direct: primary] test> rs.addArb("mongo_arb:27017")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1765092078, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1765092078, i: 1 })
}
rs0 [direct: primary] test>
```

Vérification : Via la commande `rs.status()`, le nouveau nœud apparaît avec l'état `ARBITER`. En cas de panne d'un des nœuds de données, cet arbitre apportera la voix nécessaire pour élire un nouveau Primary et maintenir le cluster fonctionnel en mode dégradé, sans risque de perte de quorum.

```
  {
    _id: 3,
    name: 'mongo_arb:27017',
    health: 1,
    state: 7,
    stateStr: 'ARBITER',
    uptime: 94,
    lastHeartbeat: ISODate('2025-12-07T07:22:52.406Z'),
    lastHeartbeatRecv: ISODate('2025-12-07T07:22:52.405Z'),
    pingMs: Long('0'),
    lastHeartbeatMessage: '',
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    configVersion: 2,
    configTerm: 7
  }
```

Partie 3: Validation Fonctionnelle et Tolérance aux Pannes

Maintenant que l'architecture est en place, le but est de vérifier le respect du contrat **"Maître-Esclave"**. Nous devons valider que les données s'écrivent uniquement sur le Maître (Primary) et se propagent bien vers les Esclaves (Secondaries) pour assurer la cohérence. Enfin, nous devons prouver la **Haute Disponibilité** du système en simulant une panne critique pour observer la capacité du cluster à s'auto-réparer via l'élection d'un nouveau leader.

1. Test de Réplication et de Lecture :

- **Insertion :** Connexion au Primary (`docker exec -it mongo1 mongosh`) et insertion d'un document : `db.test.insert({nom: "ReplicaTest"})`.

```
rs0 [direct: primary] test> db.test.insert({nom: "ReplicaTest"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('6935308a453fe7b0eb9dc29d') }
}
rs0 [direct: primary] test>
```

- **Vérification sur Esclave :** Connexion à un Secondaire (`docker exec -it mongo2 mongosh`).
- **Activation Lecture :** La commande `db.test.find()` échoue par défaut. Exécution de `rs.secondaryOk()` pour autoriser la lecture et valider la présence de la donnée répliquée

```
PS C:\Users\chayma\Desktop\ing3\NoSql\Mongo replicasets> docker exec -it  mongo2 mongosh
Current Mongosh Log ID: 693530bf5e709ae2ea9dc29c
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.9
Using MongoDB:          7.0.26
Using Mongosh:          2.5.9

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-12-07T06:36:49.413+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesys
tem
   2025-12-07T06:36:50.883+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
   2025-12-07T06:36:50.884+00:00: vm.max_map_count is too low
------

rs0 [direct: secondary] test>  db.test.find()
[ { _id: ObjectId('6935308a453fe7b0eb9dc29d'), nom: 'ReplicaTest' } ]
rs0 [direct: secondary] test>
```

2. Simulation de Panne (Failover) :

- **Arrêt brutal :** Arrêt du conteneur Maître depuis Windows :

```
C:\Users\chayma\Desktop\ing3\NoSql\Mongo replicasets>docker stop mongo1
mongo1

C:\Users\chayma\Desktop\ing3\NoSql\Mongo replicasets>docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS             PORTS
                  NAMES
66dd0b4ff00f   mongo:7.0   "docker-entrypoint.s…"   34 minutes ago   Up 34 minutes      27017/tcp
                  mongo_arb
920bbe473362   mongo:7.0   "docker-entrypoint.s…"   2 days ago       Up About an hour   0.0.0.0:27019->27017/tcp, [::]:2
7019->27017/tcp   mongo3
6dd7d8e9cd4d   mongo:7.0   "docker-entrypoint.s…"   2 days ago       Up About an hour   0.0.0.0:27018->27017/tcp, [::]:2
7018->27017/tcp   mongo2
```

Observation : Sur le terminal du nœud restant (`mongo2`), exécution de `rs.status()`. On constate que le cluster a détecté la panne et a automatiquement élu `mongo2` comme nouveau **PRIMARY**.

```
  members: [
    {
      _id: 0,
      name: 'mongo1:27017',
      health: 0,
      state: 8,
      stateStr: '(not reachable/healthy)',
      uptime: 0,
      optime: { ts: Timestamp({ t: 0, i: 0 }), t: Long('-1') },
      optimeDurable: { ts: Timestamp({ t: 0, i: 0 }), t: Long('-1') },
      optimeDate: ISODate('1970-01-01T00:00:00.000Z'),
      optimeDurableDate: ISODate('1970-01-01T00:00:00.000Z'),
      lastAppliedWallTime: ISODate('2025-12-07T07:50:52.873Z'),
      lastDurableWallTime: ISODate('2025-12-07T07:50:52.873Z'),
      lastHeartbeat: ISODate('2025-12-07T07:51:43.721Z'),
      lastHeartbeatRecv: ISODate('2025-12-07T07:50:59.497Z'),
      pingMs: Long('0'),
      lastHeartbeatMessage: 'Error connecting to mongo1:27017 :: caused by :: Could not find address for mongo1:27017: SocketException: onInvoke :: caused by :: Host not f
ound (authoritative)',
      syncSourceHost: '',
      syncSourceId: -1,
      infoMessage: '',
      configVersion: 2,
      configTerm: 7
    },
```

```
{
  _id: 1,
  name: 'mongo2:27017',
  health: 1,
  state: 1,
  stateStr: 'PRIMARY',
  uptime: 4495,
  optime: { ts: Timestamp({ t: 1765093901, i: 1 }), t: Long('9') },
  optimeDate: ISODate('2025-12-07T07:51:41.000Z'),
  lastAppliedWallTime: ISODate('2025-12-07T07:51:41.395Z'),
  lastDurableWallTime: ISODate('2025-12-07T07:51:41.395Z'),
  syncSourceHost: '',
  syncSourceId: -1,
  infoMessage: 'Could not find member to sync from',
  electionTime: Timestamp({ t: 1765093869, i: 1 }),
  electionDate: ISODate('2025-12-07T07:51:09.000Z'),
  configVersion: 2,
  configTerm: 9,
  self: true,
  lastHeartbeatMessage: ''
},
```

Retour à la normale : Redémarrage du nœud (`docker start mongo1`), qui réintègre le groupe en tant que **SECONDARY**.

```
C:\Users\chayma\Desktop\ing3\NoSql\Mongo replicasets>docker start mongo1
mongo1

C:\Users\chayma\Desktop\ing3\NoSql\Mongo replicasets>docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS             PORTS
                NAMES
66dd0b4ff00f   mongo:7.0   "docker-entrypoint.s…"   36 minutes ago   Up 36 minutes      27017/tcp
                mongo_arb
7e41def443f3   mongo:7.0   "docker-entrypoint.s…"   2 days ago       Up 2 seconds       0.0.0.0:27017->27017/tcp, [::]:2
7017->27017/tcp   mongo1
920bbe473362   mongo:7.0   "docker-entrypoint.s…"   2 days ago       Up About an hour   0.0.0.0:27019->27017/tcp, [::]:2
7019->27017/tcp   mongo3
6dd7d8e9cd4d   mongo:7.0   "docker-entrypoint.s…"   2 days ago       Up About an hour   0.0.0.0:27018->27017/tcp, [::]:2
7018->27017/tcp   mongo2
```

=> **Validation de la tolérance aux pannes.** Suite à l'arrêt volontaire du conteneur `mongo1` (ancien Maître), on observe via la commande `rs.status()` que le mécanisme d'élection s'est déclenché automatiquement. Le nœud `mongo2` a changé de statut pour devenir le nouveau **PRIMARY**, garantissant ainsi la continuité de service (écritures toujours possibles) malgré la perte d'un serveur.