Université Sorbonne Paris Nord
Département d'informatique

Travaux Pratiques

TP n°2 : Réplication et tolérance aux pannes avec MongoDB et Cassandra

Denis Linde

4 décembre 2025

Table des matières

1 Partie 1 : Réplication avec MongoDB

L'objectif de cette première partie est de reproduire pas à pas les manipulations présentées dans les vidéos du cours, tout en les expliquant de manière claire et détaillée.

1.1 Étape 1 : Lancer les serveurs MongoDB en mode réplication

Pour commencer, nous devons lancer trois instances de serveurs `mongod` pour simuler un cluster sur notre machine locale. Chaque serveur utilisera un port différent et un dossier de stockage de données distinct (`disk1`, `disk2`, `disk3`).

Nous utilisons l'option `-replSet monreplicatset` pour indiquer qu'ils appartiennent au même ensemble de réplication.

Voici les commandes exécutées dans trois terminaux différents :

Listing 1 – Lancement des 3 instances MongoDB

```
# Terminal 1
~$ mongod --replSet monreplicatset --port 27018 -dbpath disk1

# Terminal 2
~$ mongod --replSet monreplicatset --port 27019 -dbpath disk2

# Terminal 3
~$ mongod --replSet monreplicatset --port 27020 -dbpath disk3
```

1.2 Étape 2 : Connexion au premier serveur

Il faut ensuite se connecter via le terminal client au premier serveur (celui écoutant sur le port 27018) afin de pouvoir configurer le Replica Set.

Listing 2 – Connexion avec le client mongosh

```
~$ mongosh --port 27018
```

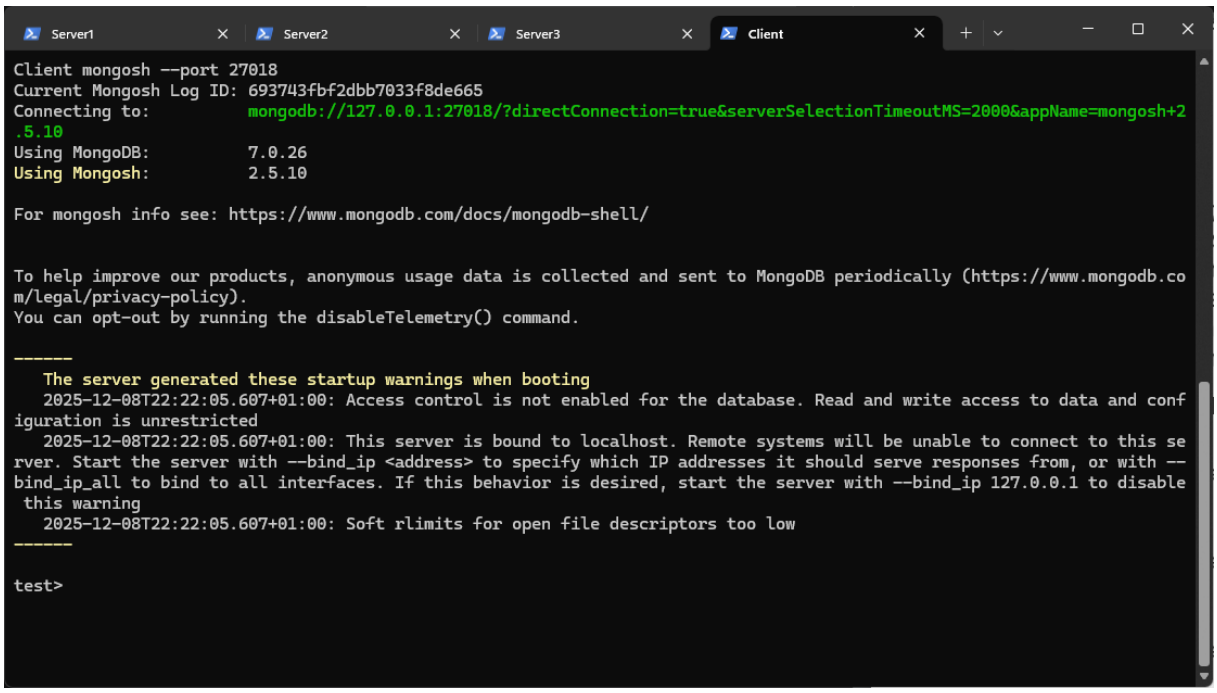

FIGURE 1 – Connexion réussie au serveur sur le port 27018

1.3 Étape 3 : Initialisation du Replica Set

Une fois connecté au serveur, nous devons initialiser le Replica Set à l'aide de la commande `rs.initiate()`. Cette commande configure le serveur actuel comme membre d'un nouveau jeu de réplication.

Nous observons sur la capture d'écran ci-dessous que le prompt change de `test>` à `monreplicatset [direct: primary]`. Cela indique que l'initialisation a réussi et que ce serveur a été élu comme nœud **Primary** (maître).

```
test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: 'localhost:27018',
  ok: 1
}
monreplicatset [direct: other] test>

monreplicatset [direct: primary] test>
```

FIGURE 2 – Le serveur passe en statut Primary après l'initialisation

1.4 Étape 4 : Ajout des autres membres au Replica Set

Maintenant que le Replica Set est actif avec un nœud primaire, nous devons ajouter les deux autres serveurs (qui écoutent sur les ports 27019 et 27020) pour assurer la redondance.

Nous utilisons la commande `rs.add("hostname:port")` pour ajouter un membre. Nous répétons cette opération pour les deux serveurs restants.

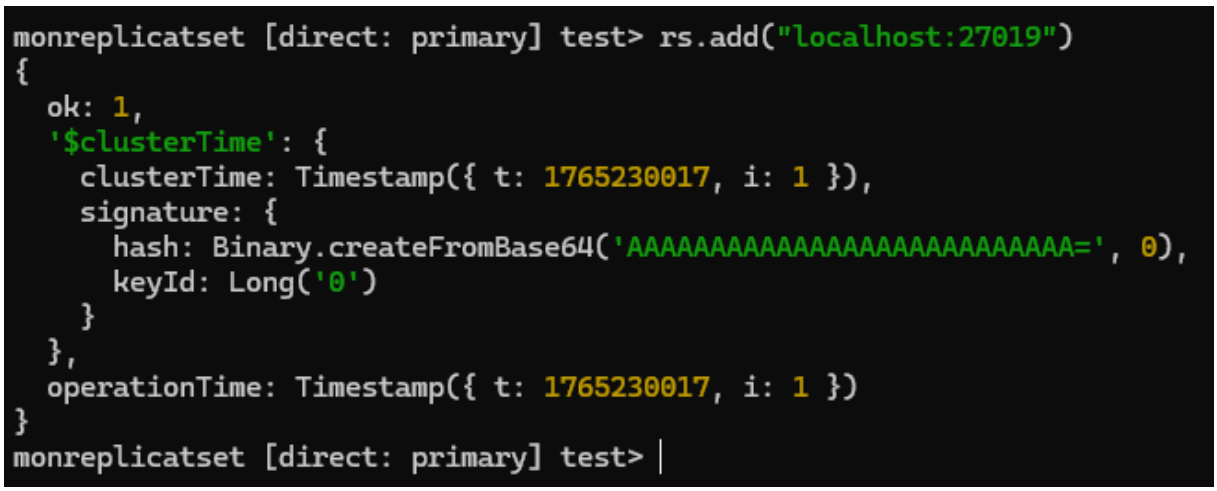

FIGURE 3 – Ajout du second serveur (port 27019) au Replica Set

1.5 Étape 5 : Vérification de la configuration

Pour s'assurer que tous les membres ont bien été ajoutés et que le Replica Set est correctement configuré, on utilise la commande `rs.config()`. Le résultat ci-dessous confirme la présence des trois membres (IDs 0, 1 et 2) fonctionnant sur les ports 27018, 27019 et 27020.

Listing 3 – Détails de la configuration (rs.config)

```
monreplicatset [direct: primary] test> rs.config()
{
  _id: 'monreplicatset',
  version: 5,
  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('6937447e7c62ac2c25560c23')
  }
}
```

1.6 Étape 6 : Vérification de l'état du cluster

Enfin, la commande `rs.status()` nous permet de vérifier l'état de santé de chaque nœud et leur rôle actuel (Primary ou Secondary). On y voit notamment que le serveur sur le port 27018 est bien `PRIMARY` (stateStr) et que les autres sont `SECONDARY`.

Listing 4 – État détaillé du cluster (rs.status)

```
monreplicatset [direct: primary] test> rs.status()
{
  set: 'monreplicatset',
  date: ISODate('2025-12-08T21:44:37.488Z'),
  myState: 1,
  term: Long('1'),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long('2000'),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1765230273, i: 1 }),
       t: Long('1') },
    lastCommittedWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1765230273, i:
       1 }), t: Long('1') },
```

```
  appliedOpTime: { ts: Timestamp({ t: 1765230273, i: 1 }), t:
     Long('1') },
  durableOpTime: { ts: Timestamp({ t: 1765230273, i: 1 }), t:
     Long('1') },
  lastAppliedWallTime: ISODate('2025-12-08T21:44:33.892Z'),
  lastDurableWallTime: ISODate('2025-12-08T21:44:33.892Z')
},
lastStableRecoveryTimestamp: Timestamp({ t: 1765230243, i: 1 }),
electionCandidateMetrics: {
  lastElectionReason: 'electionTimeout',
  lastElectionDate: ISODate('2025-12-08T21:34:54.247Z'),
  electionTerm: Long('1'),
  lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 1765229694,
      i: 1 }), t: Long('-1') },
  lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1765229694, i: 1
      }), t: Long('-1') },
  numVotesNeeded: 1,
  priorityAtElection: 1,
  electionTimeoutMillis: Long('10000'),
  newTermStartDate: ISODate('2025-12-08T21:34:54.310Z'),
  wMajorityWriteAvailabilityDate: ISODate('2025-12-08T21
     :34:54.349Z')
},
members: [
  {
    _id: 0,
    name: 'localhost:27018',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 1356,
    optime: { ts: Timestamp({ t: 1765230273, i: 1 }), t: Long('1'
       ) },
    optimeDate: ISODate('2025-12-08T21:44:33.000Z'),
    lastAppliedWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    lastDurableWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    electionTime: Timestamp({ t: 1765229694, i: 2 }),
    electionDate: ISODate('2025-12-08T21:34:54.000Z'),
    configVersion: 5,
    configTerm: 1,
    self: true,
    lastHeartbeatMessage: ''
  },
  {
    _id: 1,
    name: 'localhost:27019',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    uptime: 259,
```

```
    optime: { ts: Timestamp({ t: 1765230273, i: 1 }), t: Long('1'
       ) },
    optimeDurable: { ts: Timestamp({ t: 1765230273, i: 1 }), t:
       Long('1') },
    optimeDate: ISODate('2025-12-08T21:44:33.000Z'),
    optimeDurableDate: ISODate('2025-12-08T21:44:33.000Z'),
    lastAppliedWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    lastDurableWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    lastHeartbeat: ISODate('2025-12-08T21:44:37.193Z'),
    lastHeartbeatRecv: ISODate('2025-12-08T21:44:36.727Z'),
    pingMs: Long('0'),
    lastHeartbeatMessage: '',
    syncSourceHost: 'localhost:27018',
    syncSourceId: 0,
    infoMessage: '',
    configVersion: 5,
    configTerm: 1
  },
  {
    _id: 2,
    name: 'localhost:27020',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    uptime: 181,
    optime: { ts: Timestamp({ t: 1765230273, i: 1 }), t: Long('1'
       ) },
    optimeDurable: { ts: Timestamp({ t: 1765230273, i: 1 }), t:
       Long('1') },
    optimeDate: ISODate('2025-12-08T21:44:33.000Z'),
    optimeDurableDate: ISODate('2025-12-08T21:44:33.000Z'),
    lastAppliedWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    lastDurableWallTime: ISODate('2025-12-08T21:44:33.892Z'),
    lastHeartbeat: ISODate('2025-12-08T21:44:37.193Z'),
    lastHeartbeatRecv: ISODate('2025-12-08T21:44:37.058Z'),
    pingMs: Long('0'),
    lastHeartbeatMessage: '',
    syncSourceHost: 'localhost:27018',
    syncSourceId: 0,
    infoMessage: '',
    configVersion: 5,
    configTerm: 1
  }
],
ok: 1,
'$clusterTime': {
  clusterTime: Timestamp({ t: 1765230273, i: 1 }),
  signature: {
    hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=',
        0),
    keyId: Long('0')
  }
},
```

```
  operationTime: Timestamp({ t: 1765230273, i: 1 })
}
```

1.7 Étape 7 : Vérification détaillée avec rs.isMaster()

La commande `rs.isMaster()` fournit une vue d'ensemble de la topologie du Replica Set vue par le nœud actuel. Elle confirme notamment :

- Le nom du set (`setName: 'monreplicatset'`).
- Les hôtes membres (`hosts`).
- Le nœud primaire actuel (`primary: 'localhost:27018'`).
- Le fait que le nœud actuel est maître (`ismaster: true`).

Listing 5 – Détails du maître (rs.isMaster)

```
monreplicatset [direct: primary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId('693741797c62ac2c25560b99'),
    counter: Long('10')
  },
  hosts: [ 'localhost:27018', 'localhost:27019', 'localhost:27020'
     ],
  setName: 'monreplicatset',
  setVersion: 5,
  ismaster: true,
  secondary: false,
  primary: 'localhost:27018',
  me: 'localhost:27018',
  electionId: ObjectId('7fffffff0000000000000001'),
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1765230406, i: 1 }), t: Long('1')
       },
    lastWriteDate: ISODate('2025-12-08T21:46:46.000Z'),
    majorityOpTime: { ts: Timestamp({ t: 1765230406, i: 1 }), t:
       Long('1') },
    majorityWriteDate: ISODate('2025-12-08T21:46:46.000Z')
  },
  maxBsonObjectSize: 16777216,
  maxMessageSizeBytes: 48000000,
  maxWriteBatchSize: 100000,
  localTime: ISODate('2025-12-08T21:46:53.766Z'),
  logicalSessionTimeoutMinutes: 30,
  connectionId: 5,
  minWireVersion: 0,
  maxWireVersion: 21,
  readOnly: false,
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1765230406, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=',
          0),
      keyId: Long('0')
    }
```

```
  },
  operationTime: Timestamp({ t: 1765230406, i: 1 }),
  isWritablePrimary: true
}
```

1.8 Étape 8 : Création d'une base de données et insertion de données

Maintenant que le cluster est opérationnel, nous allons créer une base de données de test et y insérer quelques documents pour vérifier que l'écriture fonctionne correctement sur le nœud primaire.

Nous créons la base `demo1`, la collection `personnes`, et nous ajoutons trois pilotes de F1 (Verstappen, Norris, Piastri) avant de vérifier leur présence.

```
monreplicatset [direct: primary] test> use demo1
switched to db demo1
monreplicatset [direct: primary] demo1> db.createCollection("personnes")
{ ok: 1 }
monreplicatset [direct: primary] demo1> db.personnes.insert({"nom":"Verstappen"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693747cbf2dbb7033f8de666') }
}
monreplicatset [direct: primary] demo1> db.personnes.insert({"nom":"Norris"})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693747d1f2dbb7033f8de667') }
}
monreplicatset [direct: primary] demo1> db.personnes.insert({"nom":"Piastri"})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693747d7f2dbb7033f8de668') }
}
monreplicatset [direct: primary] demo1> db.personnes.find()
[
  { _id: ObjectId('693747cbf2dbb7033f8de666'), nom: 'Verstappen' },
  { _id: ObjectId('693747d1f2dbb7033f8de667'), nom: 'Norris' },
  { _id: ObjectId('693747d7f2dbb7033f8de668'), nom: 'Piastri' }
]
```

FIGURE 4 – Insertion et vérification des données sur le nœud primaire

1.9 Étape 9 : Vérification de la réplication sur un serveur secondaire

Nous nous connectons maintenant à l'un des serveurs secondaires (port 27019) pour vérifier deux choses :

— La réplication des données : les données insérées sur le primaire doivent être visibles.

— La restriction d'écriture : il doit être impossible de modifier les données sur un secondaire.

La commande `find()` confirme que les données sont bien répliquées. La commande `insert()` échoue avec l'erreur `NotWritablePrimary`, ce qui confirme que le nœud est bien en lecture seule.

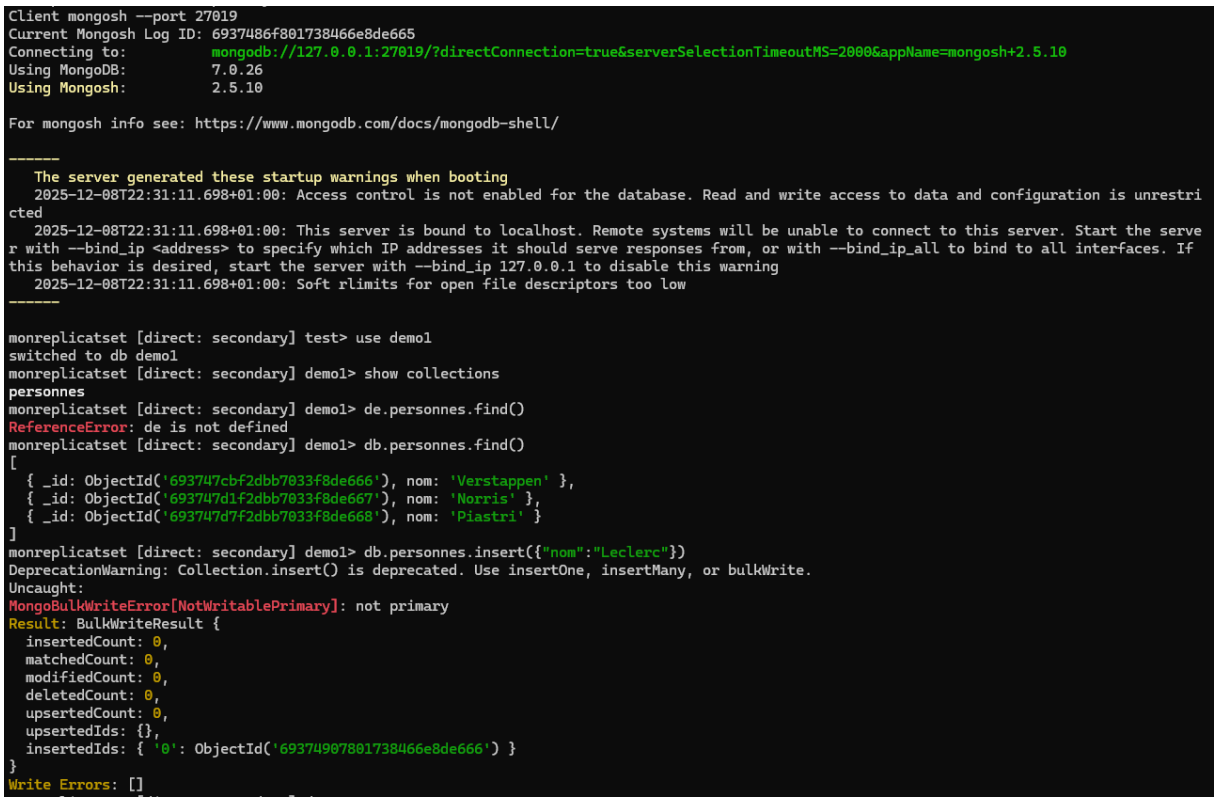

FIGURE 5 – Réplication et échec de l'écriture sur le secondaire

1.10 Étape 10 : Simulation d'une panne du serveur primaire

Pour tester la tolérance aux pannes et le mécanisme d'élection automatique, nous simulons un plantage du serveur primaire (celui sur le port 27018). Pour ce faire, nous utilisons la combinaison de touches `Ctrl+Z` dans le terminal où le serveur s'exécute, ce qui a pour effet de suspendre brutalement le processus.

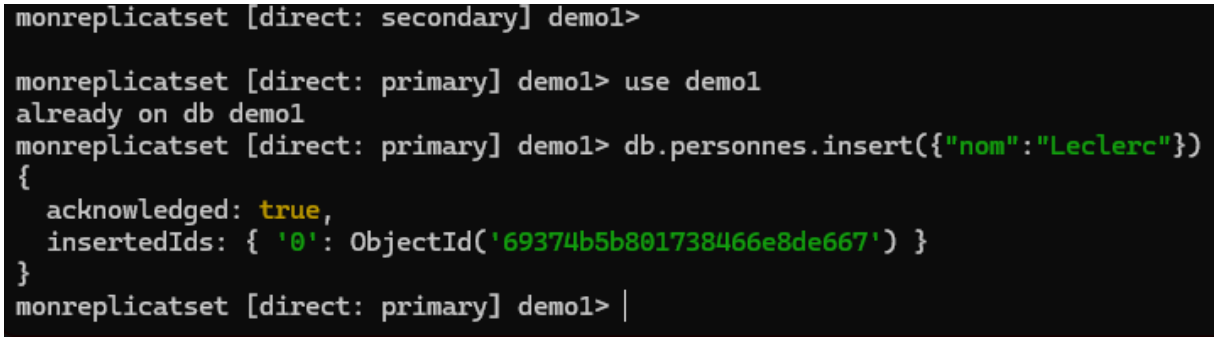

FIGURE 6 – Arrêt manuel du serveur primaire via Ctrl+Z

1.11 Étape 11 : Ajout d'un nœud arbitre

Après avoir lancé une nouvelle instance `mongod` sur le port 27021 (pour simuler une machine séparée), nous l'ajoutons au Replica Set depuis le nœud primaire en utilisant la commande `rs.addArb("localhost:27021")`.

Nous nous connectons ensuite à cet arbitre pour vérifier son comportement. Comme le montre la capture d'écran ci-dessous, toute tentative de lecture de données (via `find()`) échoue avec l'erreur `MongoServerError: node is not in primary or recovering state`. Cela confirme que l'arbitre participe aux votes d'élection mais ne stocke aucune donnée répliquée.

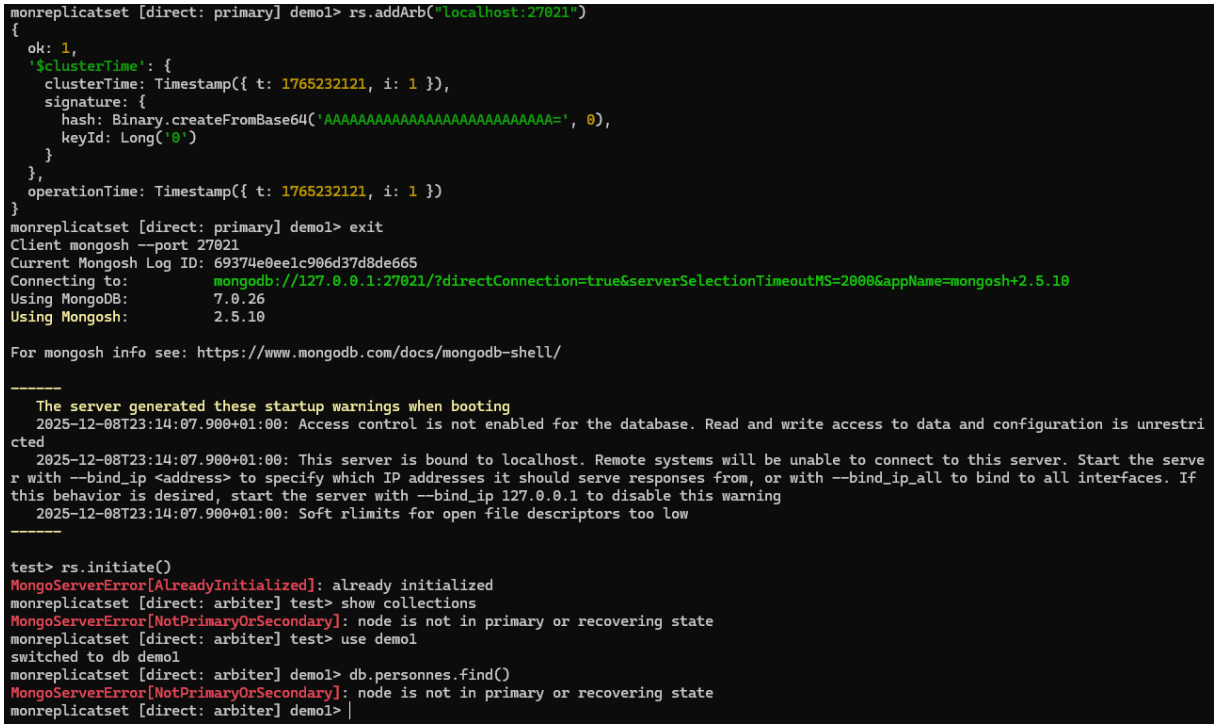

FIGURE 7 – Ajout de l'arbitre et preuve qu'il ne détient pas de données

Conclusion

Ce travail pratique nous a permis de mettre en œuvre et de valider les mécanismes de tolérance aux pannes de MongoDB via les Replica Sets. Nous avons réussi à :

- Configurer un cluster fonctionnel avec un nœud primaire et plusieurs secondaires.
- Vérifier la réplication automatique des données, assurant leur pérennité.
- Observer le basculement automatique (failover) lors de l'arrêt brutal du serveur principal.
- Comprendre le rôle d'un arbitre pour maintenir le quorum sans consommer de ressources de stockage.

Ces manipulations illustrent concrètement comment les bases de données NoSQL assurent la haute disponibilité dans des environnements distribués.