

---

**TP2 MongoDB**  
**Compte rendu du TP**

---

Réalisé par :

Célian WIRTZ ([celian.wirtz@edu.univ-paris13.fr](mailto:celian.wirtz@edu.univ-paris13.fr))

# **TP2**

## **Partie 0**

Principes généraux d'un cluster de serveurs (rappel)

- Interconnexion et échanges : tous les nœuds d'un cluster de gestion des données sont connectés et s'échangent des messages. La cohérence et la réactivité du système reposent sur cette communication continue (réPLICATION, accusés, vérifications d'état).
- Détection des pannes de réplica (esclave/secondary) : si un nœud secondaire tombe en panne, le maître/primary s'en aperçoit rapidement (absence de réponse, timeout) et peut marquer le nœud comme inactif, retenter plus tard ou réaffecter la charge.
- Panne du maître (primary) : si le primary tombe, le cluster ne peut plus coordonner les écritures ; il faut alors déclencher une *élection* automatique (algorithmes de consensus distribués — ex. Paxos, Raft) pour désigner un nouveau primary. C'est ce mécanisme qui permet la résilience et l'auto-organisation.
- Partition réseau (split-brain) : si le cluster est scindé en deux sous-ensembles incapables de communiquer, chaque sous-ensemble peut croire l'autre mort et tenter d'élire un primary, cela peut créer deux primaries simultanés (inacceptable pour la cohérence).
  - Solution classique : n'autoriser à fonctionner que la partie qui détient la majorité des votes / nœuds. Le groupe majoritaire continue de traiter les requêtes ; l'autre groupe reste en attente jusqu'au rétablissement du réseau.
- Rôle de la réPLICATION vs. montée en charge : la réPLICATION sert surtout la tolérance aux pannes et la haute disponibilité ; pour monter en charge en écriture on utilise le partitionnement (sharding), pas la réPLICATION.

Concepts spécifiques à MongoDB (réPLICATION / replica set)

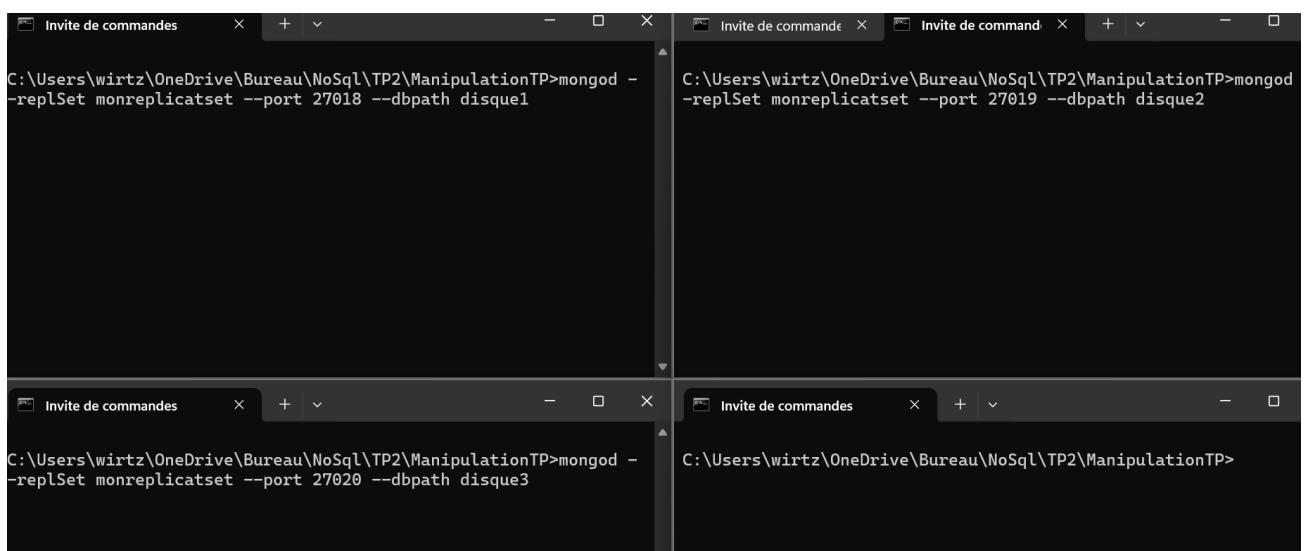
- Architecture : MongoDB utilise un modèle *primary / secondary* (termes équivalents à maître / esclave).
  - Toutes les écritures vont vers le primary.
  - Par défaut, les lectures se font aussi sur le primary (cohérence forte).
  - On peut configurer des lectures sur des secondaries (via *readPreference*), mais la réPLICATION est asynchrone, donc les secondaries peuvent être en retard et retourner des données obsolètes.
- Durabilité & journal (oplog / journal) :
  - Les écritures sont d'abord consignées dans un journal (append-only), opération séquentielle — ce qui minimise la latence de disque (pas de recherches mécaniques), et assure performance et durabilité.

- Le primary envoie un accusé de réception au client lorsque l'écriture est acceptée/consignée. Ensuite la réPLICATION vers les secondaries s'effectue via le oplog (journal d'opérations).
- Élection automatique : si le primary échoue, MongoDB lance une élection pour désigner un nouveau primary parmi les membres du replica set.
  - Si aucun sous-ensemble ne détient la majorité, aucune élection ne peut se tenir et le cluster passe en mode dégradé ; pour contourner ce cas on peut ajouter un arbitre (arbiter) qui vote mais ne stocke pas de données.
- Arbitre (arbiter) :
  - Nœud qui ne stocke pas de données, mais qui participe au vote lors des élections : utile pour obtenir une majorité sans ajouter un second replica complet.
  - Même si l'arbitre n'a pas de données, il persiste certaines informations de configuration sur disque (donc on lui donne un dbpath).
- Paramètres des membres (dans la config du replica set) :
  - host (adresse:port), priority (préférence d'élection — priority = 0 empêche un membre d'être élu), votes (si le membre peut voter ou pas), hidden (si le membre est caché), slaveDelay (délai volontaire de réPLICATION pour simuler une replica en retard), etc.
  - Le numéro de version de la configuration est incrémenté à chaque modification (ajout/suppression/reconfig).

#### Commandes de monitoring et d'administration (mentionnées)

- rs.initiate() — initialise le replica set (après avoir lancé les mongod avec la même option --replicaSet).
- rs.add("host:port") — ajoute un membre au replica set.
- rs.addArb("host:port") — (ou rs.add({ host:"host:port", arbiterOnly:true })) ajoute un arbitre qui vote mais ne stocke pas de données.
- rs.config() — affiche la configuration statique du replica set (liste des membres, priorités, etc.).
- rs.status() — affiche l'état dynamique et en temps réel de chaque nœud (qui est primary, secondaries, état de synchronisation, health, uptime, optime (dernière opération répliquée), etc.).
- isMaster() — permet de savoir si le nœud interrogé est primary ou non ; utile côté client pour connaître à qui on parle.
- Remarques importantes :
  - rs.config() donne la configuration définie statiquement ; rs.status() donne un état d'exécution live — les deux sont complémentaires.

## Initialisation du replica set



```
C:\Users\wirtz\OneDrive\Bureau\NoSql\TP2\ManipulationTP>mongod --replSet monreplicatset --port 27018 --dbpath disque1
C:\Users\wirtz\OneDrive\Bureau\NoSql\TP2\ManipulationTP>mongod --replSet monreplicatset --port 27019 --dbpath disque2
C:\Users\wirtz\OneDrive\Bureau\NoSql\TP2\ManipulationTP>mongod --replSet monreplicatset --port 27020 --dbpath disque3
C:\Users\wirtz\OneDrive\Bureau\NoSql\TP2\ManipulationTP>
```

```
C:\Users\wirtz\OneDrive\Bureau\NoSql\TP2\ManipulationTP>mongosh --port 27018
```

```
-----  
monreplicatset [direct: primary] test> rs.initiate()
```

### Ajout des autres membres

- Depuis le shell connecté à l'instance initiale (primary) :

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

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

```
monreplicatset [direct: primary] test> rs.add("localhost:27019")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1765235683, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAA='),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1765235683, i: 1 })
}
monreplicatset [direct: primary] test> |
```

## Vérifications / monitoring (exemples)

- rs.conf() — inspecter la configuration (membres, host:port, priorités, votes).
- rs.status() — voir l'état réel (qui est primary, synchronisation, délais, health=1 si en ligne, up-time, optime etc.).
- db.isMaster() — demander si le nœud courant est primary.

```
monreplicaset [direct: primary] test> rs.config()
{
  _id: 'monreplicaset',
  version: 5,
  term: 2,
  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
    }
  ]
}
```

```
,  
    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('693757fd92631fd6faeb44de')  
    }  
},
```

```
mongosh mongodb://127.0.0.1:27017/test> rs.status()  
{  
  set: 'monreplicaset',  
  date: ISODate('2025-12-08T23:24:33.913Z'),  
  myState: 1,  
  term: Long('2'),  
  syncSourceHost: '',  
  syncSourceId: -1,  
  heartbeatIntervalMillis: Long('2000'),  
  majorityVoteCount: 2,  
  writeMajorityCount: 2,  
  votingMembersCount: 3,  
  writableVotingMembersCount: 3,  
  optimes: {  
    lastCommittedOpTime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },  
    lastCommittedWallTime: ISODate('2025-12-08T23:24:31.211Z'),  
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },  
    appliedOpTime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },  
    durableOpTime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },  
    writtenOpTime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },  
    lastAppliedWallTime: ISODate('2025-12-08T23:24:31.211Z'),  
    lastDurableWallTime: ISODate('2025-12-08T23:24:31.211Z'),  
    lastWrittenWallTime: ISODate('2025-12-08T23:24:31.211Z')  
  },  
  lastStableRecoveryTimestamp: Timestamp({ t: 1765236251, i: 1 }),  
  electionCandidateMetrics: {  
    lastElectionReason: 'electionTimeout',  
    lastElectionDate: ISODate('2025-12-08T23:07:20.619Z'),  
    electionTerm: Long('2'),  
    lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 0, i: 0 }), t: Long('1') },  
    lastSeenWrittenOpTimeAtElection: { ts: Timestamp({ t: 1765235055, i: 1 }), t: Long('1') },  
    lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1765235055, i: 1 }), t: Long('1') },  
    numVotesNeeded: 1,  
    priorityAtElection: 1  
},
```

```
wMajorityWriteAvailabilityDate: ISODate('2025-12-08T23:07:20.723Z')
},
members: [
{
_id: 0,
name: 'localhost:27018',
health: 1,
state: 1,
stateStr: 'PRIMARY',
uptime: 1034,
optime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },
optimeDate: ISODate('2025-12-08T23:24:31.000Z'),
optimeWritten: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },
optimeWrittenDate: ISODate('2025-12-08T23:24:31.000Z'),
lastAppliedWallTime: ISODate('2025-12-08T23:24:31.211Z'),
lastDurableWallTime: ISODate('2025-12-08T23:24:31.211Z'),
lastWrittenWallTime: ISODate('2025-12-08T23:24:31.211Z'),
syncSourceHost: '',
syncSourceId: -1,
infoMessage: '',
electionTime: Timestamp({ t: 1765235240, i: 1 }),
electionDate: ISODate('2025-12-08T23:07:20.000Z'),
configVersion: 5,
configTerm: 2,
self: true
}
```

```
_id: 1,
name: 'localhost:27019',
health: 1,
state: 2,
stateStr: 'SECONDARY',
uptime: 589,
optime: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },
optimeDurable: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },
optimeWritten: { ts: Timestamp({ t: 1765236271, i: 1 }), t: Long('2') },
optimeDate: ISODate('2025-12-08T23:24:31.000Z'),
optimeDurableDate: ISODate('2025-12-08T23:24:31.000Z'),
optimeWrittenDate: ISODate('2025-12-08T23:24:31.000Z'),
lastAppliedWallTime: ISODate('2025-12-08T23:24:31.211Z'),
lastDurableWallTime: ISODate('2025-12-08T23:24:31.211Z'),
lastWrittenWallTime: ISODate('2025-12-08T23:24:31.211Z'),
lastHeartbeat: ISODate('2025-12-08T23:24:32.782Z'),
lastHeartbeatRecv: ISODate('2025-12-08T23:24:32.765Z'),
pingMs: Long('0'),
lastHeartbeatMessage: '',
syncSourceHost: 'localhost:27018',
syncSourceId: 0,
infoMessage: '',
configVersion: 5,
configTerm: 2
},
{
_id: 2,
name: 'localhost:27020',
health: 1,
state: 2,
stateStr: 'SECONDARY',
```

```

monreplicatset [direct: primary] test> rs.isMaster()
{
    topologyVersion: {
        processId: ObjectId('69375a2783bf8c9e0b6b2b67'),
        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('7fffffff0000000000000002'),
    lastWrite: {
        opTime: { ts: Timestamp({ t: 1765237352, i: 1 }), t: Long('2') },
        lastWriteDate: ISODate('2025-12-08T23:42:32.000Z'),
        majorityOpTime: { ts: Timestamp({ t: 1765237352, i: 1 }), t: Long('2') },
        majorityWriteDate: ISODate('2025-12-08T23:42:32.000Z')
    },
    maxBsonObjectSize: 16777216,
    maxMessageSizeBytes: 48000000,
    maxWriteBatchSize: 100000,
    localTime: ISODate('2025-12-08T23:42:41.438Z'),
    logicalSessionTimeoutMinutes: 30,
    connectionId: 10,
    minWireVersion: 0,
    maxWireVersion: 27,
    readOnly: false,
    ok: 1,
    '$clusterTime': {
        clusterTime: Timestamp({ t: 1765237352, i: 1 })
    }
}

```

Champs importants dans la configuration (rs.conf() / members[])

- \_id du replica set (nom du replica set)
- version — incrémenté à chaque reconfiguration
- members — tableau contenant chaque membre ; pour chaque membre on trouve :
  - host (adresse:port)
  - priority (influence l'éligibilité à devenir primary ; priority = 0 empêche l'élection)
  - votes (1 ou 0 — si le membre peut voter)
  - hidden, arbiterOnly (flags)
  - slaveDelay (délai de réPLICATION)
  - id numérique du membre (0, 1, 2, ...)
- Ces champs servent à contrôler qui peut devenir primary, qui vote, quels nœuds sont cachés, etc.

Connexion au primary et opérations effectuées

1. Connexion au shell Mongo du primary :

2. Création d'une base / collection de démonstration :
3. Insertion de documents dans demo1.personne :
4. Lecture sur le primary :

```
monreplicatset [direct: primary] test> use demo1
switched to db demo1
monreplicatset [direct: primary] demo1> db.create
db.createUser          db.createCollection      db.createEncryp
db.createRole

monreplicatset [direct: primary] demo1> db.createCollection("gens")
{ ok: 1 }
monreplicatset [direct: primary] demo1> db.gens
db.gens

monreplicatset [direct: primary] demo1> db.gens.insert({ "nom": "Philippe" })
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany or bulkWrite instead.
{
  acknowledged: true,
  insertedIds: { '_id': ObjectId('69377bb4627c2179dd1e2621') }
}
monreplicatset [direct: primary] demo1> db.gens.insert({ "nom": "Paul" })
{
  acknowledged: true,
  insertedIds: { '_id': ObjectId('69377bc7627c2179dd1e2622') }
}
monreplicatset [direct: primary] demo1> db.gens.find()
[
  { '_id': ObjectId('69377bb4627c2179dd1e2621'), nom: 'Philippe' },
  { '_id': ObjectId('69377bc7627c2179dd1e2622'), nom: 'Paul' }
]
monreplicatset [direct: primary] demo1> |
```

Connexion au secondary et comportement observé

Tentative de lecture et d'affichage des collections :

- Message observé : le shell indique clairement que ce nœud n'est pas primary donc certaines opérations sont restreintes.

```
monreplicatset [direct: secondary] demo1> db.gens.insert({ "nom": "Charlie" })
Uncaught:
MongoBulkWriteError[NotWritablePrimary]: not primary
Result: BulkWriteResult {
  insertedCount: 0,
  matchedCount: 0,
  modifiedCount: 0,
  deletedCount: 0,
  upsertedCount: 0,
  upsertedIds: {},
  insertedIds: { '_id': ObjectId('69377e03b2ae7d22d41e2622') }
}
Write Errors: []
monreplicatset [direct: secondary] demo1> |
```

Forcer la lecture sur un secondary

Remarque conceptuelle : lire sur un secondary peut renvoyer des données en retard (replication asynchrone), donc risque d'obsolescence.

Dans la démonstration on a utilisé/utilisé l'idée d'autoriser la lecture sur secondary (commande/option côté client ou shell).

```
monreplicaset [direct: secondary] demo1> show collections
gens
monreplicaset [direct: secondary] demo1> db.gens.find()
[
  { _id: ObjectId('69377bb4627c2179dd1e2621'), nom: 'Philippe' },
  { _id: ObjectId('69377bc7627c2179dd1e2622'), nom: 'Paul' }
]
monreplicaset [direct: secondary] demo1>
```

Simulation d'une panne du primary et comportement du cluster

1. Simulation : arrêt brutal du primary (instance sur 27018) via Ctrl+C :
  - o Arrêt du processus mongod sur 27018.
2. Effet observé :
  - o Les deux autres nœuds (27019 et 27020) « s'agitent » : ils échangent des messages de coordination et commencent une procédure d'élection automatique.
  - o Tentative de connexion à 27018 après l'arrêt échoue (pas d'écoute).
3. Résultat :
  - o Un des secondaries (ici 27019) a été élu nouveau primary. Après l'élection, on a pu se connecter sur 27019 et lire/écrire :

# Questions

## Partie 1

### 1. Qu'est-ce qu'un Replica Set dans MongoDB ?

Un Replica Set est un groupe de serveurs MongoDB qui contiennent tous les mêmes données, répliquées automatiquement.

### 2. Quel est le rôle du Primary dans un Replica Set ?

Le Primary est le nœud central qui garantit la cohérence forte, il permet de lire et écrire, c'est le seul à pouvoir écrire, il est aussi celui responsable de maintenir à jour les secondaires.

### 3. Quel est le rôle essentiel des Secondaries ?

Ils répliquent les données du Primary de manière asynchrone et peuvent potentiellement servir pour faire une lecture (mais risque pas à jour). Enfin ils peuvent prendre le relais en cas de panne.

### 4. Pourquoi MongoDB n'autorise-t-il pas les écritures sur un Secondary ?

L'objectif est d'éviter les conflits, si chacun écrit sur son secondary en même temps on aurait un risque de concurrence.

### 5. Qu'est-ce que la cohérence forte dans le contexte MongoDB ?

La cohérence forte signifie que la lecture renvoie toujours la dernière version des données.

Par défaut, MongoDB lit et écrit sur le Primary, ce qui impose cette cohérence.

### 6. Quelle est la différence entre readPreference: "primary" et "secondary" ?

Cela définit où on va faire notre lecture, sachant que la cohérence forte est maintenue uniquement sur le primary (donc lecture via secondary peut-être pas à jour).

### 7. Dans quel cas pourrait-on souhaiter lire sur un Secondary malgré les risques ?

Lorsque l'on veut éviter au Primary des requêtes contraignantes cela peut être une bonne idée de lire sur un Secondary, d'autant plus si la cohérence forte n'est pas impérative (données qui changent peu, etc...)

## **Partie 2**

8. Quelle commande permet d'initialiser un Replica Set ?

`rs.initiate()`

9. Comment ajouter un nœud à un Replica Set après son initialisation ?

Simple :

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

(Remplacer le 27019 par le numéro de port)

10. Quelle commande permet d'afficher l'état actuel du Replica Set ?

`rs.status()`

11. Comment identifier le rôle actuel (Primary / Secondary / Arbitre) d'un nœud ?

Exécuter sur le nœud :

`db.isMaster()`

(il y a une valeur ismaster qui sera ou true ou false)

12. Quelle commande permet de forcer le basculement du Primary ?

`rs.stepDown()`

13. Comment peut-on désigner un nœud comme Arbitre ? Pourquoi le faire ?

`rs.addArb("host:port")`

- Pourquoi ajouter un arbitre ?
  - Pour donner un vote supplémentaire dans les élections et ainsi permettre d'atteindre une majorité sans ajouter un replica complet (utile si on ne veut pas ou ne peut pas ajouter un nœud lourd en stockage).
  - Aussi, un arbitre ne stocke pas de données.

14. Donnez la commande pour configurer un nœud secondaire avec un délai de réPLICATION (slaveDelay).

`cfg.members[1].slaveDelay = 3600`

`rs.reconfig(cfg)`

## Partie 3

15) Que se passe-t-il si le Primary tombe en panne et qu'il n'y a pas de majorité ?

Si le primary tombe et qu'aucun sous-ensemble de nœuds ne possède la majorité des votes, aucune élection ne peut aboutir. Donc aucun nouveau Primary n'est élu et les écritures sont désormais impossibles.

16) Comment MongoDB choisit-il un nouveau Primary ? Quels critères utilise-t-il ?

Le choix se fait via un protocole d'élection distribué. Critères/principes principaux :

1. Priorité (priority)

- Si plusieurs candidats sont très similaires, la priorité configurée (priority dans la config membre) oriente le choix : plus la priority est élevée, plus un nœud est favorisé pour devenir primary. (priorité = 0 → impossible d'être élu)

2. Actualité des données (freshness / optime)

- Les électeurs préfèrent des candidats qui ont la dernière op (optime) la plus récente (le plus à jour). Le candidat doit être suffisamment à jour pour éviter la perte de données majoritaires.

17) Qu'est-ce qu'une élection dans MongoDB ?

Une élection est le processus distribué par lequel les membres d'un replica set désignent un nouveau primary lorsque l'ancien primary n'est plus disponible.

18) Que signifie « auto-dégradation » du Replica Set ? Dans quel cas cela survient-il ?

C'est lorsqu'un Primary se change lui-même en Secondary par souci de sécurité (il perd la majorité).

19) Pourquoi est-il conseillé d'avoir un nombre impair de nœuds dans un Replica Set ?

Tout simplement pour favoriser l'obtention d'une majorité (on évite un 50 / 50).

20) Quelles conséquences a une partition réseau sur le fonctionnement du cluster ?

Une partition réseau (split) divise les nœuds en deux (ou plusieurs) sous-ensembles incapables de communiquer où seul le côté majoritaire continuera d'avoir un primary. Cela peut aussi provoquer plus de latence.

## Partie 4

21. Vous avez 3 nœuds : 27017 (Primary), 27018 (Secondary), 27019 (Arbitre). Que se passe-t-il si le Primary devient injoignable ?

27018 et 27019 votent pour 27018 qui se voit élu à la majorité des votes (en effet, l'arbitre ne vote pas pour lui-même).

22. Vous avez configuré un Secondary avec un slaveDelay de 120 secondes. Quelle est son utilité ? Quels usages en production ?

Protection contre erreurs humaines : si une suppression/écrasement accidentel(le) est répliqué, on peut récupérer la version antérieure sur le delayed replica (fenêtre de 120s).

23. Un client exige une lecture toujours à jour, même en cas de bascule. Quelles options de readConcern et writeConcern recommanderiez-vous ?

Pour garantir la lecture la plus à jour possible (et une durabilité correcte) :

- Écrire avec :

```
db.collection.insertOne(doc, { writeConcern: { w: "majority", wtimeout: 5000 } })
```

- Lire avec :

```
db.getCollection("collection").find().readConcern("majority")
```

- Pourquoi ce choix :

- writeConcern: "majority" : assure que l'écriture a été acceptée par la majorité avant d'être considérée « confirmée ».
- readConcern: "majority" + lecture depuis le primary garantit que la lecture reflète des écritures qui ont atteint la majorité.

24. Dans une application critique, vous voulez garantir que l'écriture est confirmée par au moins deux nœuds. Quelle option writeConcern utiliser ?

```
db.collection.insertOne(doc, { writeConcern: { w: 2 } })
```

25. Un étudiant a lu depuis un Secondary et récupéré une donnée obsolète. Expliquez pourquoi et comment éviter cela.

C'est le problème Classique avec la lecture via un Secondary, les données ne sont pas forcément à jour, il suffit de lire via le Primary pour avoir celle à jour.

26. Montrez la commande pour vérifier quel nœud est actuellement Primary dans votre Replica Set.

```
rs.status()
```

Regarder dans la sortie la section members : le membre avec "stateStr".

27 — Forcer une bascule manuelle du Primary sans interruption majeure

On fait rs.stepDown() sur le Primary.

28 — Procédure pour ajouter un nouveau nœud secondaire dans un Replica Set en fonctionnement

Étapes pratiques

```
mkdir -p /data/db_new
```

```
mongod --replSet "monReplica7" --port 27021 --dbpath /data/db_new --bind_ip <ip_de_la_machine>
rs.add("nouveau_host:27021")
```

29 — Commande pour retirer un nœud défectueux d'un Replica Set

```
rs.remove("host:port")
```

30 — Configurer un nœud secondaire pour qu'il soit *caché* (hidden) — et pourquoi

```
cfg = rs.conf()
```

```
cfg.members[id].hidden = true
```

```
rs.reconfig(cfg)
```

- On veut qu'il ne serve pas les lectures clients et ne soit pas élu primary.
- Sécurité / conformité : garder une copie interne non exposée.

31 — Modifier la priorité d'un nœud pour en faire le Primary préféré

Exemple : rendre le membre d'index i le candidat préféré

1. Récupérer et modifier la config :

```
cfg = rs.conf()
```

```
cfg.members[i].priority = 30 //Plus c'est haut plus c'est prioritaire.
```

```
rs.reconfig(cfg)
```

32 — Vérifier le délai de réplication d'un Secondary par rapport au Primary

On utilise rs.printSecondaryReplicationInfo() ou rs.status() et on regarde l'optime.

### 33 — rs.freeze() : que fait-elle et quand l'utiliser ?

Cela empêche le nœud en question de devenir Primary pendant un certain temps, cela peut être utile par exemple pendant une maintenance pour éviter de tout changer.

### 34 — Redémarrer un Replica Set sans perdre la configuration

On relance MongoDB

### 35 — Surveiller en temps réel la réPLICATION (logs et shell)

Surveillance via les logs (mongod logs)

On utilise tail -f /var/log/mongodb/mongod.log ou rs.printSlaveReplicationInfo()

## Questions complémentaires :

### 37 — Qu'est-ce qu'un Arbitre (Arbiter) et pourquoi ne stocke-t-il pas de données ?

C'est un nœud qui ne peut pas être élu aux votes lors d'élections (mais fournit un vote) et ne conserve pas de copie des données (pas d'oplog appliqué, pas de données persistées utiles).

### 38 — Vérifier la latence de réPLICATION entre Primary et Secondaries

On utilise rs.printSecondaryReplicationInfo() en comparant l'optime.

### 39. Quelle commande MongoDB permet d'afficher le retard de réPLICATION des membres secondaires ?

rs.printSlaveReplicationInfo()

### 40. Quelle est la différence entre la réPLICATION asynchrone et synchrone ? Quel type utilise MongoDB ?

- RéPLICATION synchrone : l'écriture est considérée comme réussie seulement après avoir été appliquée sur le(s) nœud(s) répliqués requis (ex. primary attend confirmation d'un secondary). Cela garantit que les données sont immédiatement présentes sur les réplicas choisis, mais augmente la latence d'écriture.
- RéPLICATION asynchrone : le primary accepte l'écriture, renvoie une confirmation au client, puis les secondaries récupèrent et appliquent ces opérations ensuite (avec un certain délai). Avantage : latence d'écriture plus faible ; inconvénient : secondaries peuvent être en retard.

41. Peut-on modifier la configuration d'un Replica Set sans redémarrer les serveurs ?

Oui, on utilise rs.reconfig(cfg), voir les exemples au-dessus.

42. Que se passe-t-il si un nœud Secondary est en retard de plusieurs minutes ?

Il risque de ne pas fournir les données correctes et il risque de ne jamais pouvoir devenir Primary (Pas de promotion :( ).

43. Comment MongoDB gère-t-il les conflits de données lors de la réPLICATION ?

MongoDB utilise un modèle de réPLICATION ASYNCHRONE basé sur l'oplog (journal d'opérations).

Dans ce modèle, les conflits peuvent se produire uniquement dans un cas particulier : lorsqu'un nouveau Primary est élu et que l'ancien Primary revient avec des écritures qui n'ont pas encore été réPLIQUÉES. Dans ce cas extrême, MongoDB utilise un mécanisme automatique appelé rollback.

44. Est-il possible d'avoir plusieurs Primaries simultanément dans un Replica Set ? Pourquoi ?

Cela est impossible car MongoDB est conçu pour empêcher "multi-primary" → afin d'éviter incohérences, forks d'historique et corruption des données.

45. Pourquoi est-il déconseillé d'utiliser un Secondary pour des opérations d'écriture même avec readPreference: secondary ?

De toute façon on ne peut pas écrire avec un Secondary, et on évite de faire une lecture avec car on risque de ne pas avoir les données à jour.

46. Quelles sont les conséquences d'un réseau instable sur un Replica Set ?

On va essentiellement avoir un réseau moins stable et donc beaucoup plus d'élections.