TP2 : Réplication et tolérance aux pannes avec MongoDB

L’objectif de ce TP est de comprendre le principe de la réplication dans les systèmes distribués et de l’illustrer concrètement à travers la mise en place d’un Replica Set MongoDB.

Partie 1 :

1. Réplication dans les systèmes distribués

Dans un système distribué, les données sont répliquées sur plusieurs nœuds afin d’assurer :

- la tolérance aux pannes ;
- la disponibilité du service ;
- la cohérence des donnée.

Tous les nœuds d’une grappe de serveurs sont interconnectés et échangent régulièrement des messages. Ces échanges servent notamment à répliquer les données, vérifier l’état de chaque nœud et détecter rapidement les pannes. Cette communication constante permet au système de rester cohérent et réactif.

Dans une architecture maître-esclave, si un nœud secondaire tombe en panne le maître détecte l’absence de réponse, le nœud est marqué comme inactif puis le système continue de fonctionner sans interruption. Cela fait partie des mécanismes de tolérance aux pannes.

Si le nœud maître tombe en panne une élection automatique est déclenchée et les nœuds restants votent pour élire un nouveau maître. Cette élection repose sur des algorithmes de consensus distribués. Ce mécanisme permet au système de s’auto-organiser sans intervention humaine.

En cas de partition réseau, un risque apparaît :
plusieurs groupes de nœuds pourraient élire chacun un maître, ce qui casserait la cohérence.
Pour éviter cela, MongoDB applique une règle simple : seul le groupe possédant la majorité des nœuds peut continuer à fonctionner.

2. Réplication dans MongoDB

MongoDB implémente la réplication via une architecture Primary / Secondary, appelée Replica Set.

- Un seul nœud est Primary :
  - il reçoit toutes les écritures ;
  - par défaut, il gère aussi les lectures.
- Les autres nœuds sont Secondary :
  - ils répliquent les données du Primary ;
  - ils peuvent servir des lectures si on l’autorise explicitement.

La réplication MongoDB est asynchrone : l’écriture est validée sur le Primary, le client reçoit un accusé de réception et ensuite seulement, les données sont propagées vers les Secondaries.

3. Mise en place d’un Replica Set MongoDB

Le Replica Set est simulé sur une seule machine à l’aide de :

- 3 instances MongoDB
- 3 ports différents

- 3 répertoires de données distincts

Noeud	Port	Répertoire
Noeud 1	27018	disque1
Noeud 2	27019	disque2
Noeud 3	27020	disque3

Le nom du Replica Set est identique pour tous les nœuds.

On commence par créer le réseau Docker :

```
ktl@Quingjie:~$ docker network create mongo-rs
316b997d9925ce00eda7b321f3af6d17cd129fe892ff04c0ea27aca15c5338fd
```

On démarre les nœuds :

```
ktl@Quingjie:~$ docker run -d \
  --name mongo1 \
  --network mongo-rs \
  -p 27018:27017 \
  mongo:7 \
  mongod --replSet rs0 --bind_ip_all
234668a58d11ddc307a5989679b876df953ef16a12418adcf2489d31fdf0e444
```

```
ktl@Quingjie:~$ docker run -d \
  --name mongo2 \
  --network mongo-rs \
  -p 27019:27017 \
  mongo:7 \
  mongod --replSet rs0 --bind_ip_all
33ab8e3168f5b4937d4d8b4d081b093a6678e971cb118dc75acf24f8e0f0d2a2
ktl@Quingjie:~$ docker run -d \
  --name mongo3 \
  --network mongo-rs \
  -p 27020:27017 \
  mongo:7 \
  mongod --replSet rs0 --bind_ip_all
79fe08f0640ecd7ab8db1a428c8bd9a366739e859a0ee78202ee98954da56f3c
```

Ensuite on vérifie :

```
ktl@Quingjie:~$ docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED         STATUS         PORTS
                                                         NAMES
79fe08f0640e   mongo:7                  "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:2702
0->27017/tcp, [::]:27020->27017/tcp                      mongo3
33ab8e3168f5   mongo:7                  "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:2701
9->27017/tcp, [::]:27019->27017/tcp                      mongo2
234668a58d11   mongo:7                  "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes   0.0.0.0:2701
8->27017/tcp, [::]:27018->27017/tcp                      mongo1
```

Les trois conteneurs MongoDB sont bien actifs.

On peut alors se connecter au premier conteneur (le futur primary) :

```
ktl@Quingjie:~$ docker exec -it mongo1 mongosh
Current Mongosh Log ID: 6942a75579cdb489279dc29c
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTim
eoutMS=2000&appName=mongosh+2.5.9
Using MongoDB:          7.0.26
Using Mongosh:          2.5.9
```

Ensuite il faut initialiser le Replica Set :

```
test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: '234668a58d11:27017',
  ok: 1
}
```

Ajouter mongo2 et 3 :

```
rs0 [direct: primary] test> rs.add("mongo2:27017")
...
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1765976511, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1765976511, i: 1 })
}
rs0 [direct: primary] test> rs.add("mongo3:27017")
...
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1765976519, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1765976519, i: 1 })
}
```

Puis on vérifie :

```
    _id: 2,
    name: 'mongo3:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    uptime: 49,
    optime: { ts: Timestamp({ t: 1765976560, i: 1 }), t: Long('1') },
    optimeDurable: { ts: Timestamp({ t: 1765976560, i: 1 }), t: Long('1') },
    optimeDate: ISODate('2025-12-17T13:02:40.000Z'),
    optimeDurableDate: ISODate('2025-12-17T13:02:40.000Z'),
    lastAppliedWallTime: ISODate('2025-12-17T13:02:40.415Z'),
    lastDurableWallTime: ISODate('2025-12-17T13:02:40.415Z'),
    lastHeartbeat: ISODate('2025-12-17T13:02:47.265Z'),
    lastHeartbeatRecv: ISODate('2025-12-17T13:02:47.765Z'),
    pingMs: Long('0'),
    lastHeartbeatMessage: '',
    syncSourceHost: 'mongo2:27017',
    syncSourceId: 1,
    infoMessage: '',
    configVersion: 5,
    configTerm: 1
  }
 ],
 ok: 1,
 '$clusterTime': {
  clusterTime: Timestamp({ t: 1765976560, i: 1 }),
  signature: {
    hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
    keyId: Long('0')
  }
 },
 operationTime: Timestamp({ t: 1765976560, i: 1 })
}
    lastHeartbeatMessage: '',
    syncSourceHost: '234668a58d11:27017',
    syncSourceId: 0,
    infoMessage: '',
    configVersion: 5,
    configTerm: 1
  },
  {
```

4. Création d’une base et d’une collection (sur le Primary)

On commence par créer une base :

```
rs0 [direct: primary] test> use demo
switched to db demo
rs0 [direct: primary] demo>
```

Ensuite on crée une collection :

```
rs0 [direct: primary] demo> db.createCollection("personnes")
...
{ ok: 1 }
rs0 [direct: primary] demo>
```

5. Insérer des données (sur le Primary)

```
rs0 [direct: primary] demo> db.personnes.insertMany([
...   { nom: "Alice", age: 22 },
...   { nom: "Bob", age: 30 },
...   { nom: "Charlie", age: 27 }
... ])
...
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6942b3c879cdb489279dc29d'),
    '1': ObjectId('6942b3c879cdb489279dc29e'),
    '2': ObjectId('6942b3c879cdb489279dc29f')
  }
}
```

MongoDB impose que toutes les écritures soient effectuées sur le nœud PRIMARY afin de garantir la cohérence des données. En interne, MongoDB écrit sur le Primary, puis dans le journal (oplog), fait un accusé de reception au client et enfin fait la réplication asynchrone vers les Secondaries.

6. Vérifier la réplication (sur un Secondary)

Pour vérifier la réplication sur un Secondary, on ouvre un nouveau terminal dans lequel on écrit :

```
ktl@Quingjie:~$ docker exec -it mongo2 mongosh
Current Mongosh Log ID: 6942bd98bdf410ee289dc29c
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=t
electionTimeoutMS=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/


To help improve our products, anonymous usage data is collected and s
oDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.
```

Puis on sélectionne la base et on essaye de lire :

```
rs0 [direct: secondary] test> use demo
...
switched to db demo
rs0 [direct: secondary] demo> db.personnes.find()
...
[
  {
    _id: ObjectId('6942b3c879cdb489279dc29f'),
    nom: 'Charlie',
    age: 27
  },
  { _id: ObjectId('6942b3c879cdb489279dc29d'), nom: 'Alice', age: 22 },
  { _id: ObjectId('6942b3c879cdb489279dc29e'), nom: 'Bob', age: 30 }
]
rs0 [direct: secondary] demo>
```

On n'a pas besoin d'autoriser la lecture que le secondary puisqu'ici c'est une version récente de MongoDB, donc la lecture est autorisée par défaut. Sans ça il aurait fallut faire : *rs.secondaryOk()*.

7. Tester l'interdiction d'écriture (sur un Secondary)

Pour cela on va tenter à partir de mongo2 d'écrire quelque chose :

```
rs0 [direct: secondary] demo> db.personnes.insertOne({ nom: "David", age: 40 })
...
MongoServerError[NotWritablePrimary]: not primary
rs0 [direct: secondary] demo>
```

Comme on le voit, il y a une erreur. MongoDB interdit toute écriture sur les nœuds secondaires afin de garantir une cohérence forte et éviter les conflits.

8. Simuler une panne du Primary

Pour cela on va ouvrir un nouveau terminal et on va faire :

```
ktl@Quingjie:~$ docker stop mongo1
mongo1
ktl@Quingjie:~$
```

Alors normalement, le Primary disparaît, les Secondaries détectent son absence, une élection automatique démarre et un nouveau Primary est élu.

5 à 10 secondes plutard sur mongo2, on a :

```
rs0 [direct: secondary] demo> rs.status()
...
{
  set: 'rs0',
  date: ISODate('2025-12-17T17:29:17.496Z'),
  myState: 1,
  term: Long('4'),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long('2000'),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1765992554, i: 1 }), t: Long('4') },
    lastCommittedWallTime: ISODate('2025-12-17T17:29:14.314Z'),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1765992554, i: 1 }), t: Long('4') },
    appliedOpTime: { ts: Timestamp({ t: 1765992554, i: 1 }), t: Long('4') },
    durableOpTime: { ts: Timestamp({ t: 1765992554, i: 1 }), t: Long('4') },
    lastAppliedWallTime: ISODate('2025-12-17T17:29:14.314Z'),
    lastDurableWallTime: ISODate('2025-12-17T17:29:14.314Z')
  },
  lastStableRecoveryTimestamp: Timestamp({ t: 1765992514, i: 1 }),
  electionCandidateMetrics: {
    lastElectionReason: 'stepUpRequestSkipDryRun',
    lastElectionDate: ISODate('2025-12-17T17:27:04.276Z'),
    electionTerm: Long('4'),
    lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 1765992421, i: 1 }), t: Long('3') }
,
    lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1765992421, i: 1 }), t: Long('3') },
    numVotesNeeded: 2,
    priorityAtElection: 1,
    electionTimeoutMillis: Long('10000'),
    priorPrimaryMemberId: 0,
    numCatchUpOps: Long('0'),
    newTermStartDate: ISODate('2025-12-17T17:27:04.300Z'),
    wMajorityWriteAvailabilityDate: ISODate('2025-12-17T17:27:04.329Z')
  },
  electionParticipantMetrics: {
    votedForCandidate: true,
    electionTerm: Long('3'),
    lastVoteDate: ISODate('2025-12-17T17:23:31.860Z'),
    electionCandidateMemberId: 0,
    voteReason: '',
    lastAppliedOpTimeAtElection: { ts: Timestamp({ t: 1765992201, i: 1 }), t: Long('2') }
```

```
    lastAppliedOpTimeAtElection: { ts: Timestamp({ t: 1765992201, i: 1 }), t: Long('2') },
    maxAppliedOpTimeInSet: { ts: Timestamp({ t: 1765992201, i: 1 }), t: Long('2') },
    priorityAtElection: 1
  },
  members: [
    {
      _id: 0,
      name: '234668a58d11:27017',
      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('2025-12-17T17:27:14.304Z'),
      lastDurableWallTime: ISODate('2025-12-17T17:27:14.304Z'),
      lastHeartbeat: ISODate('2025-12-17T17:29:15.809Z'),
      lastHeartbeatRecv: ISODate('2025-12-17T17:27:12.817Z'),
      pingMs: Long('0'),
      lastHeartbeatMessage: 'Error connecting to 234668a58d11:27017 :: caused by :: Could n
ot find address for 234668a58d11:27017: SocketException: onInvoke :: caused by :: Host not
found (non-authoritative), try again later',
      syncSourceHost: '',
      syncSourceId: -1,
      infoMessage: '',
      configVersion: 5,
      configTerm: 4
    },
    {
      _id: 1,
      name: 'mongo2:27017',
      health: 1,
      state: 1,
      stateStr: 'PRIMARY',
      uptime: 16944,
      optime: { ts: Timestamp({ t: 1765992554, i: 1 }), t: Long('4') },
      optimeDate: ISODate('2025-12-17T17:29:14.000Z'),
      lastAppliedWallTime: ISODate('2025-12-17T17:29:14.314Z'),
      lastDurableWallTime: ISODate('2025-12-17T17:29:14.314Z'),
      syncSourceHost: '',
      syncSourceId: -1,
      infoMessage: '',
      electionTime: Timestamp({ t: 1765992424, i: 1 }),
      electionDate: ISODate('2025-12-17T17:27:04.000Z')
```

On voit que mongo2 est le nouveau Primary.

9. Vérifier que le cluster continue de fonctionner

```
rs0 [direct: primary] demo> db.personnes.find()
..

 {
   _id: ObjectId('6942b3c879cdb489279dc29f'),
   nom: 'Charlie',
   age: 27
 },
 { _id: ObjectId('6942b3c879cdb489279dc29d'), nom: 'Alice', age: 22 },
 { _id: ObjectId('6942b3c879cdb489279dc29e'), nom: 'Bob', age: 30 }

rs0 [direct: primary] demo>
```

Les données sont toujours là malgré la panne de l'ancien Primary.

```
rs0 [direct: primary] demo> db.personnes.insertOne({ nom: "David", age: 35 })
...
{
  acknowledged: true,
  insertedId: ObjectId('6942ead4bdf410ee289dc29e')
}
rs0 [direct: primary] demo>
```

Avec mongo2 on a pu ajouter une donnée.

Partie 2 : Questions

Partie 1 :

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

Un Replica Set dans MongoDB est un groupe de serveurs MongoDB qui maintiennent tous les mêmes données. C'est important puisque :

- dans le cas ou un serveur tombe en panne, les autres peuvent continuer de fonctionner
- les données sont sauvegardées à plusieurs endroits
- le système continue même si un serveur crash

La réplication n'est pas la stratégie utilisée par MongoDb pour la monté en charge mais elle a un rôle clé pour la tolérance aux pannes et la haute disnonibilité des données.

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

Le primary dans un Replica Set, c'est le serveur maitre, c'est le seul qui accepte les écritures (insertions, modifications, suppressions). C'est un peu le chef du groupe.
Plus précisément, il reçoit toutes les opérations d'écriture puis il les enregistre dans un journal oplog (operations log). Après avoir fait les replicas, le primary envoie les données aux secondaries. Par défaut, c'est lui qui gère les lectures.

3. Quel est le rôle essentiel des Secondaries ?

Les secondaries dans un Replica Set sont les esclaves. Ils répliquent les données du Primary et servent de backup en cas de panne. Ils copient automatiquement toutes les modifications du Primary. Si le Primary tombe, un Secondary est élu pour devenir le nouveau Primary. Ils peuvent être en lecture seule pour les clients, mais c'est risqué car les données ne sont pas forcément actualisées/ les dernières données existantes au moment des requêtes des clients.

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

MongoDB n’autorise pas les écritures sur un secondary pour garantir la cohérence des données et éviter les conflits. Sur MongoDB, la réplication est asynchrone.

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

MongoDB privilégie une cohérence forte. La cohérence forte garantit qu’on lit toujours la version la plus récente des données, celle qui vient d’être écrite. Les écritures se font sur le primary et une fois la donnée enregistrée et le journal de log à jour, le serveur envoie un accusé de réception ou d’acquittement au client. Ensuite seulement la donnée est répliqué vers les autres nœud.
Sans cohérence forte, on pourrait lire sur un secondary qui n’a pas reçu la dernière mise à jour (décalage de quelques milisecondes).

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

Aspect	readPreference : « primary »	readPreference : «secondary»
Lecture sur	Le serveur primary	Les serveurs secondaries
Cohérence	Forte (données à jour)	Faible (peut être en retard)
Performance	Peut surcharger le Primary	Répartit la charge de lecture
Défaut	C'est le comportement par défaut	Doit être configuré explicitement

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

Si on lit depuis le primary, on est certain d’avoir la version la plus récente de la donnée. Mais si on lit depuis le secondary (nœud secondaire), on risque d’obtenir une version obsolète. Donc lire sur un secondary, c’est faire un compromis : on sacrifie la cohérence immédiate pour obtenir d’autres bénéfices.

Les raisons principales sont :

- La répartition de la charge (Load Distribution) : Comme le primary gère toutes les écritures, en laissant toutes les écritures cela peut le surchargé. Ainsi en distribuant les lecture sur les secondaries, on équilibre la charge de travail. Cela permet une meilleure performance globale du système, que le Primary reste réactif pour les écritures et une capacité de lecture qui scale horizontalement (on peut ajouter plus de Secondaries).
- La réduction de la latence réseau : Si les serveurs sont situés à différents endroits géographiques, certains clients peuvent être plus proche d’un secondary que du primary et comme le temps de transmission des données est diretement lié à la distance physique, ça peut être plus intéressant de lire sur un secondary pour avoir un temps de réponse plus rapide pour l’utilisateur et optimiser les ressources réseau et ainsi avoir une meilleure expérience utilisateur.
- L’isolation des charges lourdes : Comme certaines opérations sont très couteuses, elles peuvent monopoliser les ressources, donc éxécuter ces opérations sur un secondary protège le primary. Ainsi, les opérations critiques en écriture ne sont pas impactées et il y a une séparation des préoccupations (production/analyse) ce qui permet une stabilité du système principal.
- Tolérance à la cohérence éventuelle : Toutes les données n’ont pas le même niveau de criticité : certaines informations peuvent être **l**égèrement obsolètes sans conséquence alors la cohérence éventuelle (eventual consistency) suffit pour ces cas. Cela permet une meilleur utilisation des ressources, une architecture plus flexible et des performances optimisées selon les besoins réels

Partie 2 :

Soit rs le Replica Set et db un nœud.

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

La commande qui permet d'initialiser un Replica Set après un démarrage est *rs.initiate()*.

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

On peut ajouter un nœud à un Replica Set après son initialisation avec la commande : *rs.add("hostname:port").*

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

La commande qui permet d'afficher l'état actuel du Replica Set est *rs.status()*.

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

On peut savoir son rôle en faisant les commandes *db.isMaster()* ou *db.secondary()* qui renverront true ou flse selon le nœud.

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

La commande qui permet de forcer le basculement du Primary est *rs.stepDown()*.

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

On peut désigner un nœud comme arbitre en faisant :

1. Récupération de la configuration : *cfg = rs.conf()*
2. Modification du membre ciblé pour le convertir en arbitre : *cfg.members[<index>].arbiterOnly = true* (Un arbitre ne peut pas être un nœud de données, donc ce nœud ne doit plus avoir de priorités ou options liées à un nœud classique.)
3. Application de la nouvelle configuration : *rs.reconfig(cfg)*

Un arbitre ne stocke pas de données, il participe uniquement au vote lors des élections pour élire le Primary. C'est utile pour maintenir un nombre impair de votes afin de garantir un quorum, sans ajouter un vrai nœud de données.

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

La commande pour configurer un nœud secondaire avec un délai de réplication est *cfg.members[<index>].slaveDelay = le_delai*.
Plus précisément on fait :
cfg = rs.conf()
cfg.members[<index>].slaveDelay = le_delai
cfg.members[<index>].priority = 0
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 en panne et qu'il n'y a pas de majorité, le cluster devient indisponible en écriture et aucun nouveau Primary ne peut être élu.

Comme MongoDB utilise un mécanisme de vote à la majorité (plus de 50 % des votants), sans majorité c'est impossible de prendre une décision collective fiable alors le système entre en

mode lecture seule (selon la configuration), appelé le mode dégradé, on fait alors de l'auto-dégradation. Cela permet d 'éviter le split-brain (deux Primary simultanés), et donc ça garantie une seule version de la vérité. En conséquence, les écritures sont refusées, les secondaires restent en mode lecture seule et le système attend que la majorité soit rétablie

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

Pour choisir un nouveau Primary, MongoDB fait une élection. Pour cela ils utilisent des critères :

- Critères d'éligibilité (éliminatoires) : pour déterminer les nœuds qui sont éligibles
    1. Priorité > 0 : les nœuds avec priorité 0 ne peuvent jamais devenir Primary
    2. État "healthy" : le nœud doit être opérationnel et joignable
    3. Données à jour : ne doit pas avoir un retard de réplication trop important
- Critères de sélection (ordre de préférence) : pour sélectionner le noeud
    1. Priorité la plus élevée : chaque nœud a une priorité (0-1000, défaut = 1)
    2. Fraîcheur des données : l'oplog le plus récent (moins de replication lag)
    3. ID du nœud : en cas d'égalité parfaite, critère arbitraire pour départager

Chaque nœud vote, et celui qui a la majorité devient le Primary.

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

Une élection est le processus de vote par lequel les membres du Replica Set désignent un nouveau Primary.

Il se déclenche en cas de panne du Primary actuel, de perte de communication avec le Primary, d'ajout/de retrait de membres, de la modification de configuration (ex: changement de priorité) ou suite à une commande manuelle (*rs.stepDown()*).

Le déroulement de l'élection :

1. Détection : les membres constatent l'absence du Primary (heartbeat manquant)
2. Candidature : un ou plusieurs Secondaries éligibles se proposent
3. Vote : chaque membre vote pour un candidat
4. Majorité : le candidat qui obtient >50% des votes devient Primary
5. Synchronisation : les autres membres reconnaissent le nouveau Primary

Celle ci dure généralement quelques secondes (5-10s en conditions normales).

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

L'auto-dégradation est le processus où le cluster réduit automatiquement ses capacités pour préserver l'intégrité des données. Alors, le Primary se rétrograde en Secondary (stepdown), le cluster passe en mode lecture seule et il est impossible d'élire un nouveau Primary.

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

Il est conseillé d'avoir un nombre impair de nœuds dans un Replica Set pour éviter les situations d'égalité lors des votes et optimiser les ressources.

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

Une partition réseau isole les nœuds du cluster en plusieurs groupes incapables de communiquer entre eux. Seul le groupe qui possède la majorité peut conserver ou élire un Primary et continuer à accepter lectures et écritures. Le groupe minoritaire, lui, ne peut pas élire de Primary : s'il en contenait un, il se rétrograde automatiquement. Ce groupe passe alors en lecture seule ou devient indisponible. MongoDB agit ainsi pour éviter tout split-brain, garantissant la cohérence des

données au prix d’une éventuelle perte de disponibilité. Une fois la partition résolue, les nœuds se resynchronisent automatiquement et le cluster redevient pleinement fonctionnel.

Partie 3:

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

Si le Primary devient injoignable, une élection se déclenche et le Secondary 27018 devient le nouveau Primary.

1. Détection
    - 27018 et 27019 ne reçoivent plus de heartbeat de 27017
    - Après ~10 secondes sans réponse, ils considèrent 27017 comme mort
2. Élection
    - 27018 (Secondary) se propose comme candidat
    - 27018 vote pour lui-même (1 vote)
    - 27019 (Arbitre) vote pour 27018 (2 votes)
    - Majorité atteinte : 2/3 votes
3. Promotion
    - 27018 devient le nouveau Primary
    - Il commence à accepter les écritures

Et si 27017 revient plus tard, alors il se joint comme Secondary, il se resynchronise avec le nouveau Primary (27018) et il reprend son rôle de membre normal.

22. Vous avez configuré un Secondary avec un slaveDelay de 120 secondes. Quelle est son utilité ? Quels usages peut-on en faire dans la vraie vie ?

Un Secondary retardé (delayed) maintient une copie historique des données avec un décalage temporel intentionnel. Le secondary applique les opérations avec 120 secondes de retard, il a toujours l’état du Système tel qu’il était dans les 2 min avant l’instant T et il ne participe pas aux élections. Il peut être utile pour protéger les données contre les erreurs humaines (ex : suppression) ou contre les corruptions logiques (ex : mise à jour défectueuse).

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

Pour garantir qu’un client lise toujours la version la plus récente, même après un failover, il faut utiliser :

- writeConcern: "majority" → l’écriture n’est confirmée que lorsqu’elle est répliquée sur la majorité
- readConcern: "majority" → la lecture ne renvoie que des données validées par la majorité

Cette combinaison assure que, même si le Primary tombe juste après une écriture, le nouveau Primary disposera forcément de cette donnée. Cela élimine le risque de lire une valeur non validée ou annulée après failover. Le coût est une latence légèrement plus élevée.

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

Il faut utiliser :

- writeConcern: { w: 2 }

Cela impose que l'écriture soit confirmée par le Primary et au moins un Secondary avant de répondre au client. Ce niveau protège contre la perte de données si le Primary tombe immédiatement après l'écriture. On peut aussi ajouter un timeout ou la journalisation si nécessaire.

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

Un Secondary peut renvoyer une donnée obsolète à cause du replication lag : la réplication est asynchrone, donc un Secondary peut avoir un léger retard et retourner une valeur encore non mise à jour. Pour éviter cela :

- Lire sur le Primary → cohérence garantie
- Ou utiliser readConcern: "majority" → lire uniquement des données validées
- Ou utiliser un Secondary avec limite de retard (maxStalenessSeconds)
- Ou appliquer un schéma "read-after-write" : lire sur le même nœud que celui ayant écrit

En résumé : les lectures Secondaries améliorent la performance, mais pas la fraîcheur des données.

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

La commande pour vérifier que le noeud db est Primary est db.isMaster() (renvoie les informations sur le Primary). Mais on peut aussi utiliser rs.status (renvoie la liste de tous les membres avec leur état respectif) et cela permet de regarder tous les noeuds.

27. Expliquez comment forcer une bascule manuelle du Primary sans interruption majeure.

On peut forcer une bascule manuelle du Primary sans interruption majeur en utilisant la commande db.stepDown() sur le primary actuel. Alors, il termine les opérations en cours, il refuse les nouvelles écritures, il se rétrograde en secondary, il y a une élection automatique d'un nouveau primary puis la reprise des opérations (cela crée une interruption d'environ 10 à 30secs). Ça peut être utile pour la maintenance planifiée du Primary , pour la migration de charge, pour les tests de haute disponibilité ou pour la mise à jour progressive (rolling upgrade)

28. Décrivez la procédure pour ajouter un nouveau nœud secondaire dans un Replica Set en fonctionnement.

1. Démarrer MongoDB en mode Replica Set : *mongod --replSet nom_du_replicaset --port 27020 --dbpath /data/db*
2. Se connecter au primary existant : *mongo --host primary_host:27017*
3. Ajouter le nouveau membre : *rs.add("nouveau_host:27020")*

Après l'ajout, automatiquement le nouveau noeud rejoint le Replica Set, il y a alors une synchronisation et le nouveau noed copie toutes les données depuis un membre existant puis il passe en réplication continue (oplog). Alors il devient opérationnel comme Secondary.

29. Quelle commande permet de retirer un nœud défectueux d'un Replica Set ?

La commande pour retirer un nœud defectueux est *rs.remove("host_defectueux:27019")*. Pour cela on se connecte au Primary de Replica Set, on exécute la commande de retrait, alors le nœud est immédiatement exclu de la configuration alors les autres membres cessent de

communiquer avec lui et il y a un recalcul automatique de la majorité. Avant la suppression il faut s'assurer que le retrait ne fait pas perdre la majorité des nœuds (si on a 3 nœuds on peut en supprimer 1 mais pas 2).

30. Comment configurer un nœud secondaire pour qu'il soit caché (non visible aux clients) ? Pourquoi ferait-on cela ?

En faisant :

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

Le nœud caché est invisible pour les clients, il ne peut pas devenir Primary, il continue de répliquer normalement et il participe aux votes (sauf si il est configuré autrement). Cela peut être utile pour avoir un serveur dédié aux analytics (exécute des requêtes lourdes sans impacter les clients , isolé du trafic de production), pour faire des backups (prend des snapshots sans perturber le service, peut être arrêté/redémarré sans notification aux clients), pour faire du reporting (génère des rapports périodiques, ne doit pas recevoir de trafic applicatif) ou pour faire du test ou du développement (environnement de test avec données réelles, isolé de la production). L'avantage principal est la séparation des charges opérationnelles et analytiques.

31. Montrez comment modifier la priorité d'un nœud afin qu'il devienne le Primary préféré.

En faisant :

cfg = rs.conf()
cfg.members[0].priority = 10 // Priorité élevée (défaut = 1, max = 1000)
rs.reconfig(cfg)

Alors, pendant la prochaine élection, le nœud sera favorisé. Si il y a plusieurs candidats, ce sera celui avec la priorité la plus haute qui l'emportera. Le Primary actuel peut être forcé à se rétrograder si un nœud prioritaire apparaît. Cela est utile si on a un serveur plus puissant qui devrait gérer les écritures ou si on a un emplacement géographique optimal. Exemple de valeur de priorité : priority:0 (ne peut jamais devenir Primary), priority:1 (1 est la valeur par défaut), priority :10 (le nœud est préféré par rapport à des numéro de priorité inférieure).

32. Expliquez comment vérifier le délai de réplication d'un Secondary par rapport au Primary.

Le délai de réplication (replication lag) correspond au retard avec lequel un Secondary applique les opérations du Primary. On peut le vérifier par rapport au primary en faisant :
rs.printReplicationInfo() // donne les infos sur l'oplog du primary
rs.printSecondaryReplicationInfo() // pour chaque secondaries affiche la date de sadernière opération et son retard exact en secondes

L'interprétation est simple :

- < 1 seconde : synchronisation excellente
- 1 à 10 secondes : normal
- > 30 secondes : signe de problème (réseau, charge, ressources insuffisantes)
  Un lag qui augmente régulièrement indique que le Secondary n'arrive plus à suivre et nécessite une investigation.

33. Que fait la commande rs.freeze() et dans quel scénario est-elle utile ?

La commande *rs.freeze(secondes)* empêche temporairement un nœud de devenir Primary. C'est utile dans le cas d'une maintenance planifiée (On veut éviter qu'un nœud devienne Primary

pendant une fenêtre de maintenance et ça permet de travailler sur le nœud sans risque de bascule), d'une synchronisation en cours (Un Secondary vient d'être ajouté et n'est pas encore à jour, on le freeze jusqu'à ce qu'il soit synchronisé et ça évite qu'il soit élu alors qu'il est en retard, de tests contrôlés (On veut tester le comportement avec un ensemble spécifique de candidats, ça freeze certains nœuds pour contrôler qui peut être élu) ou de problèmes de performance temporaires (Un nœud connaît des problèmes (disque lent, surcharge) alors on le freeze pour éviter qu'il devienne Primary et dégrade les performances globales).

Après la durée spécifiée, le nœud redevient éligible automatiquement.

34. Comment redémarrer un Replica Set sans perdre la configuration ?

Pour redémarrer un Replica Set sans perdre la configuration, on fait un redémarrage progressif (rolling restart) membre par membre. On redémarre les Secondaries d'abord puis on redémarre le Primary, on redemarre le Primary en dernier pour minimiser le nombre de bascule (une seule élection nécessaire, quand le Primary s'arrete, sinon on doit faire 2 élections)

35. Expliquez comment surveiller en temps réel la réplication via les logs MongoDB ou commandes shell.

La réplication peut être surveillée en temps réel grâce à plusieurs outils complémentaires fournis par MongoDB et par le système. La méthode la plus directe consiste à utiliser les commandes du shell pour observer l'état du Replica Set. Avec *rs.status()*, on obtient une vue complète du rôle, de la santé et du niveau de synchronisation de chaque nœud. Il est possible d'automatiser l'affichage en boucle pour disposer d'une mise à jour régulière, ou d'utiliser *rs.printSecondaryReplicationInfo()* pour suivre uniquement le retard de réplication des Secondaries.

En parallèle, l'analyse des logs MongoDB permet de détecter immédiatement les problèmes. En surveillant le fichier *mongod.log* en continu, on peut filtrer les événements importants : élections, erreurs de heartbeat, retards de synchronisation ou anomalies d'oplog. Cela permet de réagir rapidement en cas de ralentissement ou de bascule involontaire.

MongoDB fournit également des métriques internes via *db.serverStatus().repl* ou *replSetGetStatus*, qui donnent des informations détaillées sur l'état de la réplication, la taille de l'oplog et les éventuels incidents.

Enfin, pour une supervision plus robuste, des outils comme MongoDB Ops Manager, Atlas, Prometheus ou Grafana peuvent générer des tableaux de bord et des alertes. Les indicateurs clés à suivre sont notamment le replication lag, la durée de l'oplog, les erreurs de heartbeat et la fréquence des élections.

Ces différentes approches combinées permettent un suivi fiable et réactif du fonctionnement du Replica Set.

Partie 4 : Questions complémentaires

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

Un Arbitre est un membre du Replica Set qui participe uniquement aux élections sans répliquer les données. Il ne stocke aucune donnée, il participe aux votes lors des élections, il consomme très peu de ressources (RAM, disque, CPU), il ne peut jamais devenir Primary et il ne sert pas les lectures. Il

ne stocke pas de données car sçn rôle est uniquement électoral, pas opérationnel. Il sert de "voix décisive" dans les votes, c'est tout, il permet d'obtenir un nombre impair de votants sans le coût d'un Secondary complet.

38. Comment vérifier la latence de réplication entre le Primary et les Secondaries ?
Pour vérifier la latence de réplication entre le Primary et les Secondaries, on peut utiliser *rs.printSecondaryReplicationInfo()*, qui affiche directement le retard en secondes pour chaque Secondary. On peut aussi analyser *rs.status()* en comparant le champ *optimeDate* du Primary et de chaque Secondary pour calculer le lag. Un script peut automatiser cette vérification en lisant régulièrement ces valeurs. Enfin, *db.serverStatus().metrics.repl* fournit aussi des informations sur la réplication. En général, un lag < 1s est excellent, 1–10s normal, et > 60s indique un problème à investiguer.

39. Quelle commande MongoDB permet d'afficher le retard de réplication des membres secondaires ?
La commande MongoDB permettant d'afficher le retard de réplication des membres secondaires est *rs.printSecondaryReplicationInfo()*. Elle renvoie l'identité du Secondary, le dernier timeStamp synchronisé et le décalage exact en secondes/heures par rapport au Primary.

40. Quelle est la différence entre la réplication asynchrone et synchrone ? Quel type utilise MongoDB ?
La réplication synchrone impose que chaque écriture soit confirmée par tous les nœuds avant de renvoyer la réponse au client. Elle garantit une cohérence stricte et aucune perte de données, mais augmente fortement la latence et peut réduire la disponibilité si un Secondary répond lentement.

À l'inverse, la réplication asynchrone — celle utilisée par défaut par MongoDB — valide immédiatement l'écriture sur le Primary, puis réplique en arrière-plan vers les Secondaries. Cela offre de meilleures performances et une haute disponibilité, mais introduit un risque de *replication lag* et, en cas de crash brutal du Primary, une possible perte d'écritures récentes.

MongoDB peut toutefois se rapprocher d'un comportement synchrone grâce au *writeConcern*, qui permet d'exiger la confirmation par plusieurs nœuds avant de valider une opération. Cela offre un compromis ajustable entre performance et durabilité.

41. Peut-on modifier la configuration d'un Replica Set sans redémarrer les serveurs ?
On peut modifier la configuration d'un Replica Set sans redémarrer les serveurs. Les opérations comme ajouter ou retirer un membre, modifier une priorité, changer le nombre de votes ou rendre un nœud hidden se font toutes via une mise à jour de la configuration, appliquée dynamiquement avec *rs.reconfig()*. MongoDB propage ensuite automatiquement ces changements aux autres nœuds, ce qui permet d'ajuster la topologie du cluster sans interrompre le service.
En revanche, certaines modifications nécessitent un redémarrage car elles dépendent du fichier *mongod.conf* et non de la configuration du Replica Set. C'est le cas des changements de port, du chemin des données (*dbpath*) ou de paramètres système. En résumé, tout ce qui touche à la configuration logique du Replica Set est modifiable en ligne, tandis que les paramètres internes du processus *mongod* exigent un restart.

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

Lorsqu’un nœud Secondary accumule plusieurs minutes de retard, il ne parvient plus à suivre le rythme du Primary, ce qui provoque plusieurs effets. D’abord, les lectures effectuées sur ce nœud deviennent obsolètes : il renvoie des données anciennes et peut créer une incohérence perçue par les clients. Si le décalage devient trop important, MongoDB peut même le passer en état RECOVERING, le retirant automatiquement des nœuds disponibles pour les lectures le temps qu’il se resynchronise.

Si le retard continue à augmenter, un risque plus critique apparaît : le Secondary peut dépasser la durée de rétention de l’oplog. Comme l’oplog a une taille limitée, les opérations les plus anciennes finissent par être écrasées. Si le nœud a besoin d'opérations déjà supprimées, il ne peut plus rattraper son retard automatiquement et nécessite alors une resynchronisation complète, beaucoup plus coûteuse.

Ce retard peut aussi poser problème lors d’une élection : si un Secondary très en retard devient Primary, il pourrait ramener tout le cluster à un état ancien, entraînant une perte de données récentes. MongoDB inclut toutefois des protections, comme l’exclusion automatique des Secondaries trop décalés lorsque des limites (*maxStalenessSeconds*) sont configurées.

En pratique, un retard modéré est rattrapable, mais un retard dépassant la fenêtre d’oplog nécessite une intervention manuelle. Les causes habituelles sont un réseau lent, un disque saturé, une charge CPU élevée ou des ressources insuffisantes. Des commandes comme *rs.printReplicationInfo(), rs.printSecondaryReplicationInfo()* ou *rs.status()* permettent de diagnostiquer rapidement la situation.

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

MongoDB évite presque totalement les conflits de données grâce à son architecture : seul le Primary accepte les écritures, tandis que les Secondaries ne font que répliquer les opérations dans le même ordre. Ce modèle à source unique élimine les scénarios où deux nœuds modifient simultanément la même donnée.

Les rares conflits apparaissent uniquement lors d’événements exceptionnels, comme une bascule mal synchronisée. Si un Primary tombe avant d’avoir répliqué toutes ses opérations, puis revient après l’élection d’un nouveau Primary, MongoDB peut détecter une divergence : l’ancien Primary possède des écritures que les autres n’ont jamais vues. Dans ce cas, MongoDB effectue un rollback. Les opérations divergentes sont supprimées, sauvegardées dans des fichiers dédiés, puis le nœud se resynchronise avec le Primary en place.

Pour éviter ces situations, il est recommandé d'utiliser *writeConcern: "majority"*, qui garantit que toute écriture confirmée est déjà répliquée sur la majorité des nœuds. Ainsi, même en cas de panne, le nouveau Primary dispose forcément des opérations récentes, empêchant le rollback.

En résumé, MongoDB privilégie la prévention : une seule source d’écriture, une réplication ordonnée, des writeConcern adaptés. Lorsqu’un conflit survient malgré tout, il est résolu automatiquement via rollback, sauf cas extrêmes nécessitant une resynchronisation complète.

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

MongoDB ne peut jamais avoir plusieurs Primarys en même temps. C’est une garantie fondamentale du Replica Set, conçue pour éviter tout risque de split-brain, une situation où deux nœuds accepteraient des écritures différentes, créant des versions contradictoires des données.

Pour éviter cela, MongoDB impose qu’un nœud ne devienne Primary que s’il obtient la majorité des votes. Comme il est mathématiquement impossible d’avoir deux majorités simultanément, deux Primarys ne peuvent jamais coexister. De plus, les nœuds échangent en continu des heartbeats. Si un Primary perd le contact avec la majorité, il se rétrograde automatiquement en Secondary et refuse toute écriture, empêchant ainsi toute divergence.

En cas de partition réseau, seul le côté ayant la majorité élit un Primary ; l’autre côté reste en lecture seule. Grâce à ce mécanisme, MongoDB garantit toujours une unique source de vérité, sans conflit et sans ambiguïté.

45. Pourquoi est-il déconseillé d’utiliser un Secondary pour des opérations d’écriture même en lecture préférée secondaire ?

MongoDB n’autorise jamais les écritures sur un Secondary. Même si on utilise une lecture préférée *secondary*, cette option ne concerne que les lectures, jamais les opérations d’écriture. Les Secondaries sont strictement en lecture seule, car seul le Primary peut modifier les données.

Cette limitation est essentielle pour garantir la cohérence du Replica Set : une unique source d'écriture évite les conflits, les divergences et tout risque de *split-brain*. Si les Secondaries pouvaient recevoir des écritures, ils pourraient évoluer différemment du Primary, ce qui rendrait la réplication impossible à maintenir.

Techniquement, toute tentative d’écriture sur un Secondary renvoie une erreur comme *"not master"*, car ce nœud ne fait que répliquer passivement l’oplog du Primary. Le modèle de MongoDB repose donc sur une séparation stricte : écritures uniquement sur le Primary, lectures configurables ailleurs. Cela garantit l’intégrité et simplifie complètement la cohérence du cluster.

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

Un réseau instable peut avoir des conséquences importantes sur un Replica Set MongoDB. Lorsqu’un nœud perd le contact avec les autres, cela peut déclencher des élections fréquentes, retarder la réplication vers les Secondaries et provoquer des échecs de heartbeats, ce qui rend l’état du cluster incertain. En cas de partition réseau, le Primary isolé se rétrograde et le groupe majoritaire élit un nouveau Primary, entraînant une incohérence temporaire et des écritures refusées si elles ne peuvent atteindre la majorité.

Ces problèmes peuvent aussi provoquer des lectures obsolètes sur les Secondaries, des fluctuations répétées de l’état des nœuds, des logs saturés, des timeouts pour les clients et une surconsommation des ressources du cluster. MongoDB permet d’ajuster certains paramètres (timeouts d’élection et de heartbeat) pour atténuer ces effets, mais la qualité et la stabilité du réseau restent critiques : un réseau fiable est essentiel pour garantir la disponibilité, la cohérence et la performance du cluster.