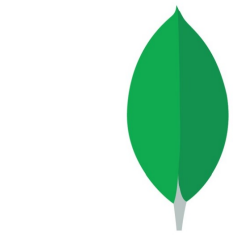
SUP GALILÉE
UNIVERSITÉ SORBONNE PARIS NORD

TP 2 : Réplication et tolérance aux pannes avec MongoDB

Auteure:
CatherineS.

Professeur encadrant :
S. YOUCEF



mongoDB®

2025/2026

Partie 1 : Manipulations MongoDB

Vidéo 1 :

L'objectif est de mettre en place une grappe de serveur. Une grappe est un rassemblement de nœud.

Rappels :

- Tous les nœuds dans une grappe de serveurs de gestion des données sont connectés et échangent des messages. Le système repose sur une communication constante entre les nœuds que ce soit pour : pour répliquer des données, transmettre des confirmations, vérifier l'état de chacun. Cette interconnexion permet au système de rester cohérent et réactif même dans un environnement distribué.
- Si un esclave tombe en panne, le maître (on est dans une architecture maître-esclave), va s'en apercevoir car les échanges entre nœuds sont réguliers, en l'absence de réponse durant un délai anormal est vite détectées. Le maitre marque le nœud comme inactif : en réaffectant la charge à un nœud actif (cette stratégie fait partie de la tolérance aux pannes)
- Si le maître tombe en panne, le système ne peut plus fonctionner normalement, car c'est qui centralise les écritures ou la coordination, pour assurer la continuité du service, un processus d'élection automatique est déclenché, c'est un algorithme distribué. Les nœuds restant négocient entre eux pour élire un nouveau maître. Ce processus s'appuie sur des algorithmes connus comme pacsos ou raft, qui permettent d'atteindre un consensus fiable même en cas de panne partielle du réseau. Ce type de mécanisme permet au système d'être résilient et s'auto organiser même sans interventions humaines.

Prenons un cluster coupé en 2 groupes et ne peux plus communiquer entre les 2 groupes. Ce scénario cause un risque critique : chaque groupe peut penser que l'autre est tombé en panne et tente de désigner un nouveau maître, on se retrouve donc avec 2 maîtres simultané, ce qui n'est pas possible pour la cohérence du système.

Pour éviter cela : on autorise seulement la grappe avec la majorité absolue à continuer à fonctionner (MongoDB utilise cette stratégie).

Le groupe majoritaire élit un maître et continue de traiter les requêtes tandis que l'autre groupe reste inactif jusqu'à rétablissement du réseau.

Le maître s'appelle un Primary et les esclaves des Secondary

MongoDB repose, par défaut, sur une architecture maître-esclave. Toutes les requêtes, en lecture ou écriture, sont envoyé au serveur principale (Primary) et il propage les changements aux autres serveurs (Secondary). Afin d'assurer la cohérence des données (sur MongoDB), les écritures se font sur le nœud principale (Primary) pour une raison :

la réconciliation des données et la gestion des conflits sont des processus complexes, en centralisant les écritures, mongoDB garantit la cohérence des données.

Par défaut, les lectures aussi se font sur le primary, ce qui permet de maintenir une cohérence forte, tant qu'on lit et écrit sur le nœud principale, on est sûr de toujours avoir accès à la dernière version des données.

Pour passer à l'échelle et configurer la charge, on peut configurer MongoDB pour autoriser les lectures sur les réplicas secondaires, mais cela présente un risque car la réplication n'est pas instantanée, on pourrait donc lire une donnée qui n'est pas à jour. Il faut faire attention à ce compromis entre performance et cohérence.

La réplication avec mongoDB est asynchrone, des que l'écriture est effectué par le primary, il envoie un acquittement pour l'application cliente puis il commence à initier des réplicas sur les secondary.

La réplication n'est pas la stratégie utilisé par mongoDB pour la montée en charge.

Pour cela mongoDB propose le sharding, ou partionnement des données (à voir plus tard dans un autre TP).

La réplication à un rôle clé pour la tolérance aux pannes, lors d'une défaillance du primary. MongoDB lance une éléction automatique pour élire un nouveau maître parmi les réplicas. S'il n'y a pas de majorité absolue, le système rentre en mode dégradé, pour éviter ce cas de figure, on ajoute un nœud spécial appelé arbitre qui ne possède pas de données, pour atteindre une majorité en cas d'élection.

MongoDB privilégie une cohérence forte, les écritures se font sur le primary et une fois la donnée enregistrée et le journal de log mis à jour, le serveur envoie un accusé de réception, on a l'acquittement client. C'est seulement maintenant que la donnée est repliqué vers les secondary.

Expérimentons ce mécanisme : on peut tout faire sur un seul ordi.

Comment instancier un replica set mongoDB ?

Chaque instance de serveur :

- ⇨ Doit être configuré avec un certains nombre de paramètres
- ⇨ Le nom du réplica set

- ⇨ Le port d'écoute pour chaque serveur

On définit une grappe constituée de 3 serveurs et répertoire de données pour chaque instance. MongoDB essaye d'écrire à la racine de notre machine /data, ici on va définir des répertoires disque1, disque2, disque3

On crée 3 serveurs et un client pour communiquer avec ces serveurs.

Mettre en places les replicas :

On ouvre 4 fenêtres de terminal et on met cette commande pour se placer dans le conteneur :

`docker exec -it MongoMongo bash`

Serveur 1 : `mkdir disque1`

Serveur 2 : `mkdir disque2/`

Serveur 3 : `mkdir disque3/`

Puis on créer les 3 serveurs :

Serveur 1 : `mongod --replSet monreplicaset --port 27018 --dbpath disque1/`

Serveur 2 : `mongod --replSet monreplicaset --port 27019 --dbpath disque2/`

Serveur 3 : `mongod --replSet monreplicaset --port 27020 --dbpath disque3/`

On se connecte au premier serveur :

Sur le client : `mongosh –port 27018`

```
root@c7200649911f:/# mongosh --port 27018
Current Mongosh Log ID: 69319732edc08845139dc29c
Connecting to:          mongodb://127.0.0.1:27018/?directConnection=true&serverSelectionTi
meoutMS=2000&appName=mongosh+2.5.9
Using MongoDB:          8.2.2
Using Mongosh:          2.5.9
```

Le client est bien connecté au premier nœud.

Il faut maintenant initialiser le replica set :

`Client#rs.initiate()`

Cette commande :

- ⇨ Créer une configuration du replica set avec le nom monreplicaset, il y a un seul membre, le nœud sur lequel on est connecté.
- ⇨ Enregistre cette configuration dans la base locale
- ⇨ Passe le nœud Serveur 1 en PRIMARY (il est tout seul donc automatiquement élu)

⇨ Démarre la réplication (donc prêt à recevoir les autres membres qu'on va ajouter)

On obtient ça :

```
monreplicaset [direct: secondary] test> 
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: 'localhost:27018',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764867395, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764867395, i: 1 })
}
```

Tout d'abord on récupère la valeur de la clé me car ca peut être localhost ou l'ip de notre machine, dans notre cas c'est localhost.

On ajoute maintenant les 2 autres serveurs à la même grappe :

`Client# rs.add("localhost:27019")` serveur 2

`Client# rs.add("localhost:27020")` serveur 3

```
monreplicaset [direct: secondary] test> rs.add("localhost:27019")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764868837, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764868837, i: 1 })
}
monreplicaset [direct: primary] test> rs.add("localhost:27020")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764868844, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764868844, i: 1 })
}
```

Quelques commandes qui permettent de surveiller l'état de notre grappe de serveur

Pour consulter la configuration actuelle du replica set :

`Client#rs.config()`

Champ	Description
_id	Nom du Replica Set (défini avec --replSet)
version	Numéro de version de la configuration. Incrémente à chaque modification
protocolVersion	Version interne du protocole de réplication

Utile pour voir si nos nœuds sont bien dans la grappe, ou modifier des paramètres, comme designer un nœud secondaire sans droit d'élection (en mettant true pour arbiterOnly), changer l'ordre de priorité etc...

Ensuite dans le tableau members :

Champ	À quoi ça sert ?
_id	Identifiant interne du nœud
host	Adresse + port du serveur (IP:PORT / hostname:PORT)
priority	Définit la chance pour ce nœud de devenir **primary** (plus haut = priorité)
arbiterOnly	= true → membre arbitre, ne stocke pas de données
hidden	Nœud non visible, souvent utilisé pour backup ou analytics
votes	Nombre de votes dans une élection

Pour consulter en temps réel l'état des noeud :

```
Client#rs.status()
```

Champ	Signification
stateStr	Rôle du nœud : PRIMARY, SECONDARY, ARBITER...
health	1 = en ligne · 0 = hors ligne
uptime	Temps de fonctionnement depuis le dernier démarrage
optimeDate	Date de la dernière écriture réplication appliquée
name	Adresse du nœud (équivalent du host de rs.config)

Pour voir quel noeuf est PRIMARY dans le replica set:

Client#rs.isMaster()

```
monreplicaset [direct: primary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId('6931c29c6a98dc0e6bd6acdf'),
    counter: Long('10')
  },
  hosts: [ 'localhost:27018', 'localhost:27019', 'localhost:27020' ],
  setName: 'monreplicaset',
  setVersion: 5,
  ismaster: true,
  secondary: false,
  primary: 'localhost:27018',
  me: 'localhost:27018',
  electionId: ObjectId('7fffffff0000000000000001'),
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1764870144, i: 1 }), t: Long('1') },
    lastWriteDate: ISODate('2025-12-04T17:42:24.000Z'),
    majorityOpTime: { ts: Timestamp({ t: 1764870144, i: 1 }), t: Long('1') },
    majorityWriteDate: ISODate('2025-12-04T17:42:24.000Z')
  },
  maxBsonObjectSize: 16777216,
  maxMessageSizeBytes: 48000000,
  maxWriteBatchSize: 100000,
  localTime: ISODate('2025-12-04T17:42:29.760Z'),
  logicalSessionTimeoutMinutes: 30,
  connectionId: 2,
  minWireVersion: 0,
  maxWireVersion: 27,
  readOnly: false,
  ok: 1,
  '$clusterTime': {
monreplicaset [direct: primary] test>
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
```

Champ	Signification
ismaster: true	Le nœud sur lequel tu es est **PRIMARY**
ismaster: false + secondary: true	Tu es sur un **SECONDARY**
primary: "host:port"	Montre quel nœud est l'actuel **PRIMARY**
hosts: []	Liste des membres du Replica Set
setName	Nom du Replica Set
arbiterOnly	Indique si ce serveur est un arbitre

Par défaut, MongoDB privilégie une cohérence forte, donc toutes les lectures et écritures passent par le primary. Il n'est pas possible d'écrire sur un nœud secondary, mais on peut forcer cela :

- ⇨ rs.slaveOk() ou readPref()
- ⇨ Dans un programme (Java, Python, NodeJS…) → utilisation du **ReadPreference**.

Les modes de lecture possibles :

Mode	Comportement
primary (défaut)	Lecture uniquement sur le PRIMARY
primaryPreferred	PRIMARY prioritaire, mais SECONDARY si indisponible
secondary	Lecture uniquement sur les SECONDARY
secondaryPreferred	SECONDARY priorisé, PRIMARY si aucun disponible
nearest	Lecture sur le nœud le plus proche/rapide

Vidéo 2 :

L'objectif est de voir ce que la réplication implique dans la gestion des données

On se positionne d'abord sur notre base de données demo1

Use demo1

```
monreplicaset [direct: primary] demo1> 
switched to db demo1
```

Créer une collection :

db.createCollection("personnes")

```
monreplicaset [direct: primary] demo1> db.createCollection("personnes")
{ ok: 1 }to db demo1
```

Ajouter des éléments à notre collection

db.personnes.insertOne({"nom":"Youcef"})

```
monreplicaset [direct: primary] demo1> db.personnes.insert({"nom":"Youcef"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('6932ba0cfecfaa86e59dc29d') }
}
monreplicaset [direct: primary] demo1> db.personnes.insertOne({"nom":"Youcef"})
{
  acknowledged: true,
  insertedId: ObjectId('6932ba1bfecfaa86e59dc29e')
}
```

La requête insert n'est plus utilisé, on utilise donc insertOne pour insérer un élément à notre collection ou insertMany pour insérer plusieurs.

On ajoute 2 autres personnes :

```
monreplicaset [direct: primary] demo1> db.personnes.insertOne({"nom":"Bassette"})
{
  acknowledged: true,
  insertedId: ObjectId('6932baccfecfaa86e59dc29f')
}
monreplicaset [direct: primary] demo1> db.personnes.insertOne({"nom":"Taylor"})
{
  acknowledged: true,
  insertedId: ObjectId('6932bae4fecfaa86e59dc2a0')
}
```

Pour consulter les éléments de la collection :

db.personnes.find()

```
monreplicaset [direct: primary] demo1> db.personnes.find()
[
  { _id: ObjectId('6932ba0cfecfaa86e59dc29d'), nom: 'Youcef' },
  { _id: ObjectId('6932ba1bfecfaa86e59dc29e'), nom: 'Youcef' },
  { _id: ObjectId('6932baccfecfaa86e59dc29f'), nom: 'Bassette' },
  { _id: ObjectId('6932bae4fecfaa86e59dc2a0'), nom: 'Taylor' }
]
```

Les paramètres par défaut de MongoDB permettent d'assurer une cohérence forte. L'écriture et la lecture se font par défaut sur le Primary, donc le port 27018. On ne peut pas écrire sur les secondaries, cependant on peut forcer la lecture sur les secondaries, en prenant le risque d'avoir des données pas à jour.

Rappel : MongoDB procède à une réplication asynchrone, lorsque le client envoie une requête d'insertion, le serveur l'écrit dans son journal et envoie un acquittement, c'est après l'envoi de l'acquittement que la réplication est initiée. Si entre temps un client se connecte à un secondary et récupère la donnée, elle ne sera pas à jour.

Maintenant connectons nous à un autre port :

mongosh –port 27018

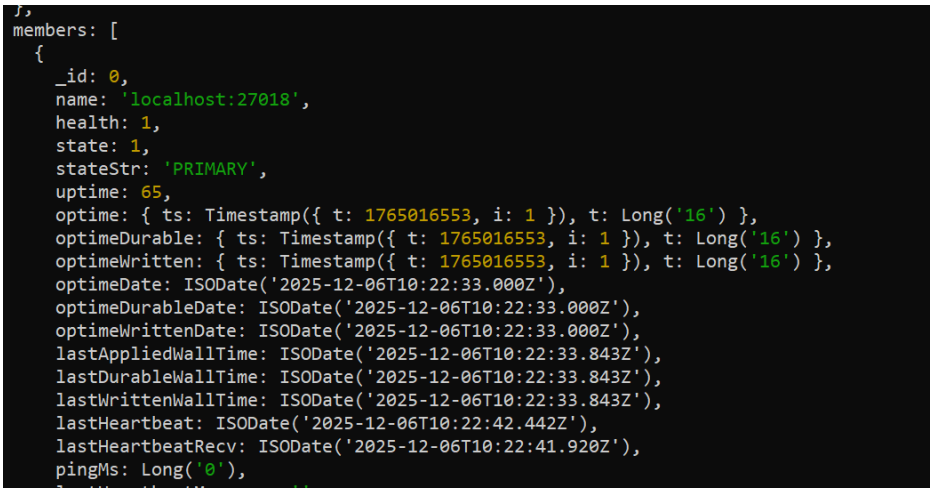
```
monreplicaset [direct: secondary] test>
```

Nous sommes bien sur un secondary

J'essaye d'accéder aux données et j'y ai accès alors que je ne devrais pas avoir accès aux données en lecture en tant que secondary, je ne comprends donc pas pourquoi j'ai accès en lecture aux données depuis le secondary. J'ai regardé les ReadPreference et c'est bien primary. Je n'ai pas compris mais je passe aux autres questions et je pourrais ainsi demander à l'encadrant. C'est probablement car je suis en mode admin ?

```
monreplicaset [direct: secondary] demo1> db.getMongo().getReadPref()
ReadPreference {
  mode: 'primary',
  tags: undefined,
  hedge: undefined,
  maxStalenessSeconds: undefined,
  minWireVersion: undefined
}
monreplicaset [direct: secondary] demo1> db.personnes.find()
[
  { _id: ObjectId('6932ba0cfecfaa86e59dc29d'), nom: 'Youcef' },
  { _id: ObjectId('6932ba1bfecfaa86e59dc29e'), nom: 'Youcef' },
  { _id: ObjectId('6932baccfecfaa86e59dc29f'), nom: 'Bassette' },
  { _id: ObjectId('6932bae4fecfaa86e59dc2a0'), nom: 'Taylor' }
]
monreplicaset [direct: secondary] demo1>
```

Pourtant on voit bien que c'est le 1er serveur qui est primary :



```
members: [
  {
    _id: 0,
    name: 'localhost:27018',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 65,
    optime: { ts: Timestamp({ t: 1765016553, i: 1 }), t: Long('16') },
    optimeDurable: { ts: Timestamp({ t: 1765016553, i: 1 }), t: Long('16') },
    optimeWritten: { ts: Timestamp({ t: 1765016553, i: 1 }), t: Long('16') },
    optimeDate: ISODate('2025-12-06T10:22:33.000Z'),
    optimeDurableDate: ISODate('2025-12-06T10:22:33.000Z'),
    optimeWrittenDate: ISODate('2025-12-06T10:22:33.000Z'),
    lastAppliedWallTime: ISODate('2025-12-06T10:22:33.843Z'),
    lastDurableWallTime: ISODate('2025-12-06T10:22:33.843Z'),
    lastWrittenWallTime: ISODate('2025-12-06T10:22:33.843Z'),
    lastHeartbeat: ISODate('2025-12-06T10:22:42.442Z'),
    lastHeartbeatRecv: ISODate('2025-12-06T10:22:41.920Z'),
    pingMs: Long('0'),
```

J'ai demandé une explication à une IA qui m'a donnée comme réponse :

« Dans notre TP, nous avons constaté que, même en étant connectés directement sur un nœud SECONDARY ([direct: secondary]), il est possible d'effectuer des lectures (find).
Ceci est lié au fait que le shell mongosh se comporte comme un outil d'administration : en connexion directe sur un nœud, il autorise les lectures sur un secondary (flag secondaryOk), même si le readPreference est primary.
En revanche, les écritures restent interdites (NotWritablePrimary).
Dans une application "normale" qui se connecte au replica set via une URL avec replicaSet=..., le readPreference contrôle réellement si les lectures peuvent être envoyées vers un secondary ou non. »

On passe à la suite

Autoriser les secondaries à lire :

Ancienne requête : `rs.secondaryOk()` (obsolète maintenant)

Nouvelle requête :

Pour forcer la lecture sur un secondary :

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

Pour faire lire sur le primary en priorité mais secondary si primary non dispo

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

En general pour changer les préferences de lecture:
`db.getMongo().setReadPref("primary" | "secondary" | "primaryPreferred" | "secondaryPreferred" | "nearest")`

A retenir :

- ⇨ Lors de l'insertion nous étions sur le port 27018, actuellement nous sommes sur le port 27019, mais nous arrivons à bien voir les données, donc la réplication a été effectué correctement
- ⇨ On ne peut pas lire sur un secondary sauf si on force la lecture, mais l'écriture reste toujours impossible.

```
monreplicaset [direct: secondary] demo1> db.personnes.insertOne({"nom":"Urso"})
MongoServerError[NotWritablePrimary]: not primary
```

Passons maintenant à la simulation d'une panne pour voir comment le système va réagir :

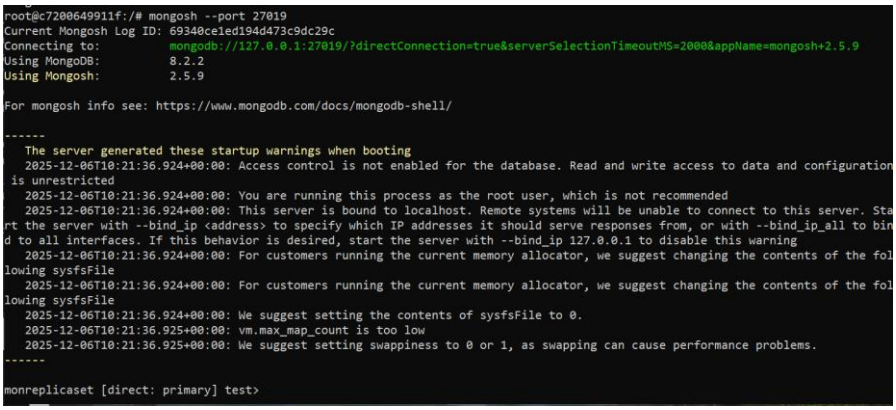
Nous allons arrêter le serveur 1 : 27018, on fait juste CTRL + C

Lorsque nous éteignons le serveur 1, nous pouvons remarquer que les 2 autres nœuds s'agitent afin d'élire un nouveau maître.

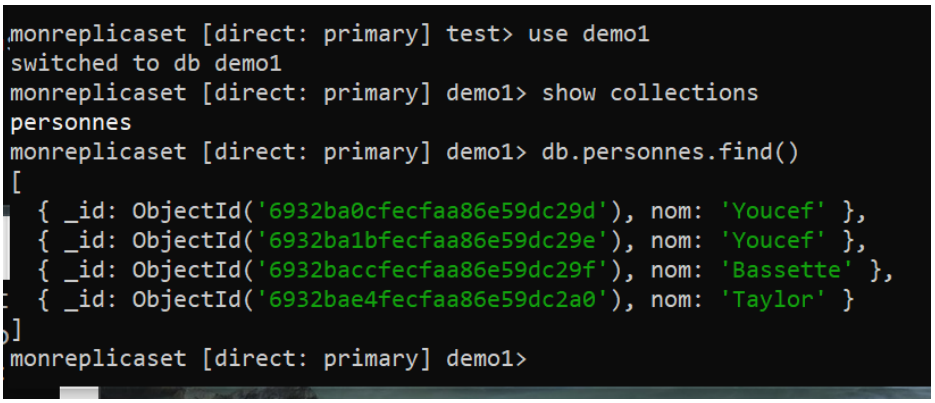
Si on essaye de se reconnecter au serveur qu'on vient d'éteindre on reçoit un message d'erreur : Ce qui est normal

```
root@c7200649911f:/# mongosh --port 27018
Current Mongosh Log ID: 69340c906bd386e4039dc29c
Connecting to:          mongodb://127.0.0.1:27018/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongo
sh+2.5.9
MongoNetworkError: connect ECONNREFUSED 127.0.0.1:27018
```

On se connecte au serveur 2 : 27019 et on peut voir qu'il a été élu primary.



Nous avons bien accès aux personnes :



Définir un arbitre :

Dans certains cas (perte d'un nœud → risque de blocage des élections), on ajoute un **ARBITRE**.

L'arbitre :

- Ne stocke PAS les données.
- Participe aux votes d'élection du PRIMARY.
- Permet d'éviter un blocage si un noeud tombe.

Pour créer un arbitre :

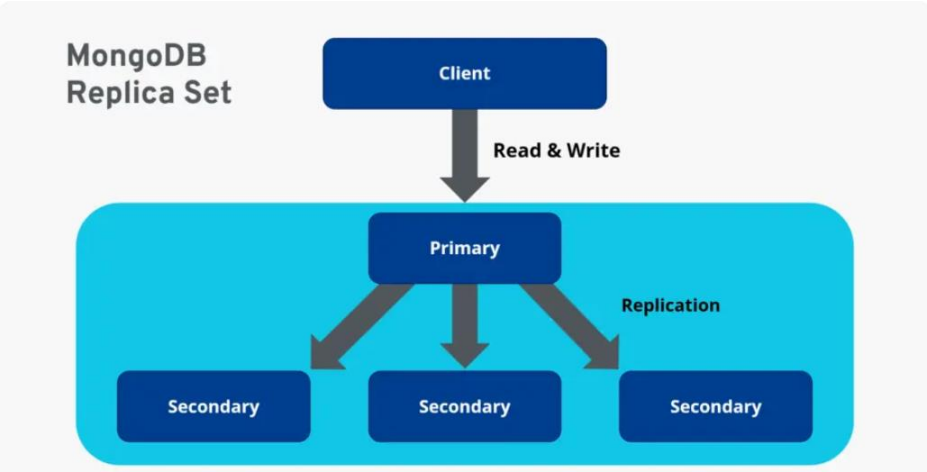
1. Créer un nouveau dossier pour lui.
2. Démarrer un mongod sur un nouveau port (ex: 27021).
3. L'ajouter au replica set avec :

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

L'arbitre écrit très peu de données, mais a tout de même besoin d'un dossier.

Partie 2 : Réplication et tolérances aux pannes avec MongoDB

1. Compréhension de base



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

Un replica set dans MongoD, c'est un groupe de serveurs/noeuds qui contiennent la même base de données copiées automatiquement pour garantir l'accès aux données même en cas de panne.

Il y a 2 types de nœuds, Primary et Secondaries

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

Le primary est le nœud maître, c'est le point de départ pour la création des Replica Sets MongoDB et il possède les droits de lecture/écriture

3. Quel est le rôle essentiel des Secondaries ?

Les Secondaries, qui sont les nœuds secondaires, esclaves, sont des copies exactes du nœud maître. Ils servent uniquement à conserver les données copiées et n'ont pas de droits de lecture/écriture généralement. En cas de panne du nœud primaire, un nœud secondaire va prendre le relais après élection.

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

Si on prend l'exemple d'un site, si on vend 2 produits alors qu'il n'en reste qu'un, cela posera un problème, en effet les secondaries ne sont pas synchronisés en temps réel. MongoDB n'autorise pas les écritures sur un secondary pour assurer la cohérence forte des données, toutes les écritures passent par le Primary.

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

La cohérence forte oblige tout les lectures et écritures à passer par le primary et ainsi la donnée lue est toujours la plus récente. Dès qu'une écriture est validée sur le primary, elle devient la référence puis elle est répliquée vers les secondaries, on est donc sur d'avoir une donnée à jour et cohérente.

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

Les readPreferences sont des paramètres qui définissent ou les lectures vont être effectués dans un replica set (primary ou secondary). Ca permet donc de choisir le nœud depuis lequel un client va lire les données.

La différence entre les ReadPreference primary et secondary, est principalement le fait que le primary permet une cohérence forte, avec des données toujours à jour.

Tandis que secondary en readPreference permet une lecture uniquement sur le secondary, il peut donc avoir des données pas à jour car la réplication n'est pas en temps réel.

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

La On peut lire sur un secondary lorsqu'on veut augmenter les performances ou répartir la charge de lecture, notamment quand la performance est prioritaire sur l'actualité des données (on peut donc se permettre une latence).

Exemple :

- ⇨ La Gros volume de données, on peut décharger le primary.
- ⇨ Statistiques, export de données ou autres : les données ne sont pas à la seconde près.

2. Commandes et configuration

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

`mongod --replSet monreplicaset --port 27018 --dbpath disque1/`

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

Tout d'abord, on initialise avec `rs.initiate()` puis on ajoute le nœud avec

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

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 ?

`rs.status()` ou `db.isMaster()` ,on regarde stateSr qui indiquera le rôle.

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("localhost:27021")
```

Un arbitre ne stocke pas de données mais participe au vote donc il aide à maintenir une majorité lors des élections, c'est très utile pour éviter un blocage si un nœud tombe.

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

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

```
cfg.members[1].slaveDelay = 120
```

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

Le nœud répliquera avec 120 secondes de retard => pratique pour restaurer des données supprimées accidentellement.

3. Résilience et tolérances aux pannes

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

S'il n'y a pas de majorité, aucun primary peut etre élu, le replica set devient en lecture seule et les écritures sont bloquées.

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

Critère d'élection : la priorité du nœud, le vote majoritaire, la fraicheur des données(le nœud le + à jour et le + prioritaire gagne)

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

C'est un processus automatique qui détermine quel secondary devient primary quand le primary tombe en panne, chaque nœud vote et celui qui obtient la majorité devient primary

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

L'auto-dégradation du Replica Set c'est le primary qui abandonne volontairement son rôle et se rétrograde en secondary. Il fait ca quand il perd contact avec les autres membres(il ne voit plus la majorité des votes) , donc plutôt que de continuer à accepter des écritures et risques des données incohérentes, il se désactive automatiquement.

Situations :

Scénario	Conséquence
Le PRIMARY ne voit plus la majorité des nœuds	Il se dégrade en SECONDARY
Partition réseau (coupure entre serveurs)	Il ne reçoit plus les heartbeats
Perte de communication ou machine isolée	Il s'auto-retire du rôle de PRIMARY

C'est pour éviter d'avoir deux primary en même temps. (split-brain)

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

Pour éviter d'avoir une égalité dans les votes, ca permet de faciliter l'élection et assurer une majorité.

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

Une partition réseau, c'est lorsque le réseau se coupe en plusieurs "morceaux", empêchant certains serveurs du Replica Set de voir les autres.

Les serveurs sont toujours allumés, mais isolés.

- ⇨ Si le primary est isolé, il stepDown et devient secondary
- ⇨ Si un nœud perd contact avec la majorité il devient secondary
- ⇨ Une partition peut bloquer les écritures s'il n'y a pas de majorité

4. Scénarios pratiques

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

Le secondary et l'arbitre forment la majorité, ils vont donc élire un nouveau primary qui sera le nœud 27018 et le cluster va continuer d'accepter les écritures

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 ?

Le secondary sera une copie du primary mais avec du retard (ici 120 secondes), ce décalage peut être utilise dans le cadre d'une récupération, protection ou analyse.

=> Si un dev supprime par erreur une collection dans un primary, il y a un décalage donc il peut récupérer les données dans le secondary

=> Si une attaque modifie/chiffre des données du Primary, il y a un délai avec le secondary donc on peut isoler l'incident, stopper la propagation et restaurer un état sain

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

Bascule = failover = changement automatique de Primary

- readConcern: "majority" → lecture sur données confirmées par la majorité
- writeConcern: { w: "majority" } → les écritures sont validées par plusieurs nœuds

Ainsi les lectures restent à jour même si le Primary change.

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 ?

On veut être sur que l'écriture est enregistrée sur au moins deux serveurs pas uniquement sur le primary afin de ne pas perdre les données meme si le primary tombe après l'écriture.

{ writeConcern: { w: 2 } } ou writeConcern: "majority"

MongoDB n'acceptera l'écriture que lorsqu'au moins 2 nœuds l'ont validée, primary ecrit la donnée, secondary confirme et après ca mongoDB répond OK au client.

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

Un secondary ne lit pas les données en temps réel car le contenu du primary est répliqué avec un peu de retard car la réplication est asynchrone.

Pour éviter ce problème : `db.getMongo().setReadPref("primary")`

Comme ca les données seront lu strictement sur le primary

On peut aussi utiliser `readConcern: "majority"` qui permet de recevoir une réponse que si au moins 2 serveurs ont la même info

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

`rs.status()` ou `db.isMaster()` ,on regarde stateSr qui indiquera le rôle.

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

`rs.stepDown()`: le primary abandonne son rôle et un secondary prend le relais

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

1) preparer le secondary : créer un dossier puis le lancer en mode réplicatif

```
mongod --replSet monreplicaset --port nb_port --dbpath nb_nom/
```

--replSet doit être le meme que le replica set en fonctionnement

2) se connecter au primary avec mogosh –port nb_primary

3) ajouter le nœud : `rs.add("localhost:27019")`

4) on peut vérifier la bonne intégration avec rs.status()

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

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

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

Un nœud secondaire caché est un nœud qui réplique les données comme un secondary normal, cependant il ne peut pas devenir primary, et il n'est pas proposé automatiquement aux clients pour les lectures.

Pourquoi ? on peut cacher un secondary pour l'utiliser sans impacter la prod, on peut faire des sauvegardes, Analyses, BI, test etc il est exploitable pour les besoins internes mais non utilisé par les clients

Pour configurer nœud secondaire caché :

1) Récupérer la configuration actuelle : `cfg = rs.conf()`
2) Trouver le membre qu'on veut cacher dans la liste :

cfg.members

[{ _id: 0, host: "localhost:27018", priority: 1 },

{ _id: 1, host: "localhost:27019", priority: 1 },

{ _id: 2, host: "localhost:27020", priority: 1 }]

3) Le cacher en rendant hidden, et mettre la priorité à 0 pour qu'il ne soit pas primary

```
cfg.members[2].hidden = true
cfg.members[2].priority = 0
```

4) Appliquer la nouvelle configuration : `rs.reconfig(cfg)`

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

Il faut juste changer la priorité et mettre une priorité plus élevé dans la configuration

```
cfg = rs.conf()

cfg.members[i].priority = 2   // plus haut que les autres

rs.reconfig(cfg)
```

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

On regarde le champ otpime après avoir fait la requête : `rs.status()`

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

La commande rs.freeze() permet de bloquer temporairement un secondary, pour l'empêcher de devenir un primary pendant un certain temps.

Elle est utilisé lors d'opérations de mintenances, de test ou pour éviter qu'un nœud lent soit élu primary, c'est un moyen temporaire de contrôler l'élection sans modifier la configuration permanente du replica set.

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

On arrête les nœuds puis on relance avec la même option

mongod --replSet monreplicaset ...

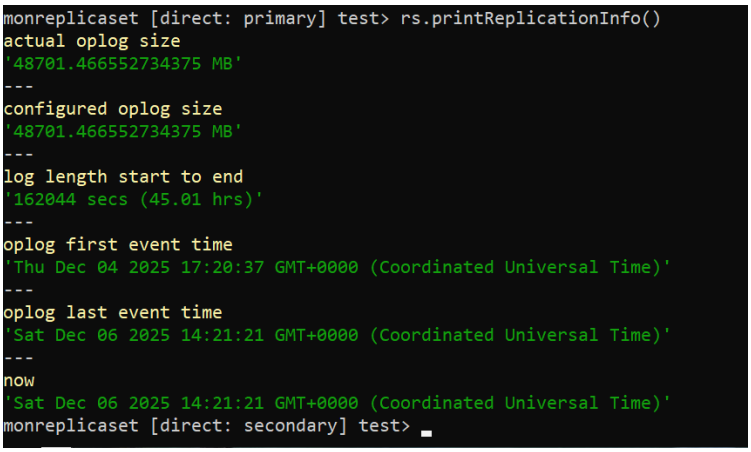
35. Expliquez comment surveiller en temps réel la réplication via les logs MongoDB ou commandes shell.

On surveille la réplication pour vérifier que les secondary suivent le primary, qu'il n'y a pas de retard, pas de perte de données. 2 manières

- Via les logs : `docker logs -f MongoMongo`

  Car notre container s'appelle MongoMongo

- Via commandes shell : `rs.printReplicationInfo()` ou `rs.printSecondaryReplicationInfo()`



Les choses à surveiller :

Le Secondary est-il à jour ?	si non → lecture obsolète possible
Quelle est la latence (lag) ?	Trop haut → risque failover lent
Est-ce qu'il change de source (syncing from) ?	Indice de saturation réseau
L'oplog est-il suffisant ?	Sinon → resync complet → long
Y a-t-il des erreurs heartbeat ?	Risque de bascule Primary

5. Questions complémentaires

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

Un Arbitre est un membre spécial du Replica Set qui participe uniquement aux votes lors des élections, mais ne contient aucune donnée.

Il sert à maintenir une majorité pour l'élection sans nécessiter une machine supplémentaire avec stockage.

- ⇨ Pas de réplication, pas de disque, très léger.
- ⇨ Utilisé lorsque l'on veut 3 votes mais seulement 2 serveurs de données.

38. Comment vérifier la latence de réplication entre le Primary et les Secondaries ?

On compare l'optime du Primary et Secondary avec `rs.status()`

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

`rs.printSecondaryReplicationInfo()` : indique pour chaque Secondary le temps de retard par rapport au Primary

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

Synchrone Tous les nœuds doivent appliquer l'écriture avant validation

Asynchrone Le Primary valide l'écriture puis la réplique ensuite aux Secondary

MongoDB utilise une réplication asynchrone

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

Oui, on peut, avec la commande : rs.reconfig(cfg), pour modifier les priorités, secondary caché, slaveDelay ou autre

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

Il va renvoyer des données obsolètes aux client, il ne pourra pas devenir Primary

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

MongoDB évite les conflits grâce à l'oplog : les SECONDARY rejouent toutes les opérations du PRIMARY dans le même ordre.
Si un Secondary diverge ou prend du retard, il se resynchronise automatiquement ; si l'écart est trop grand, il est entièrement réaligné sur le Primary.
Le Primary est toujours la bonne version, il n'existe jamais deux versions concurrentes d'une donnée.

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

Non, c'est impossible car mongoDB garantit qu'il y a exactement 1 primary, plusieurs primary peuvent provoquer un split brain qui provoquent une incohérence des données

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

Parce que les Secondary ne sont pas conçus pour écrire, seulement pour répliquer.

- ⇨ Risque de divergence
- ⇨ Données non synchronisées
- ⇨ Conflits impossibles à résoudre facilement

!!!!!!!!!! Toutes les écritures doivent passer par le Primary. !!!!!!!!!

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

- bascules fréquentes de Primary (failover répétés)

- pertes de communication entre nœuds → partition réseau
- retards de réplication → données obsolètes
- risque de blocage des écritures en absence de majorité

⇨ replica set moins cohérent et moins fiable

Rappel : partition réseau = Quand les serveurs sont encore allumés mais qu'ils **ne peuvent plus communiquer entre eux** à cause d'un problème réseau.