Gloire BAYOUNDOULA
INFOA3

Matière : **Bases de données NoSQL**
Partie : Réplication **MongoDB et Cassandra**
Enseignant : **Monsieur Samir Youcef**

Rendu TP2 - Réplication et tolérance aux pannes avec MongoDB et Cassandra

L'objectif de ce TP est de mettre en place une grappe de serveurs de MongoDB.

- **Définition de la notion de réplication quand retrouve dans la plupart des systèmes NoSQL**

Tous les nœuds dans une grappe de serveurs, gestion de données, sont connectés et échangent des messages. Cela signifie que le système repose sur une communication constante entre les différents nœuds, que ce soit pour répliquer les données, transmettre des confirmations ou vérifier l'état de chacun.
Cette interconnexion permet au système de rester cohérent et réactif même dans un environnement distribué.

Panne d'un esclave :
Quand un esclave tombe en panne, le maître (comme dans l'architecture maître esclave de MongoDB) va s'en apercevoir. Car les échanges entre nœuds sont réguliers et une absence de réponse ou un délai anormal est vite détecté. Le maître peut marquer ce nœud comme inactif ou réagir soit en essayant plus tard soit en affectant la charge à un autre nœud actif. Ce mécanisme fait partie des stratégies de tolérance aux pannes.

Panne du maître :
Si le maître tombe en panne. Dans ce cas, le système ne peut plus fonctionner normalement car c'est le maître qui centralise les écritures ou la coordination.

Pour assurer la continuité des services, un processus d'élection automatique est alors déclenché. Il s'agit d'un algorithme distribué où les nœuds restants vont négocier entre eux pour élire un nouveau maître.
Ce processus peut s'appuyer sur des algorithmes connus comme Paxos qui permet d'atteindre un consensus fiable même en cas de panne partielle du réseau ou de certains nœuds.
Par conséquent, c'est ce type de mécanisme qui permet à un système distribué d'être résilient et de s'auto organiser sans intervention humaine .

Exemple d'élection du nouveau noeud maître

Imaginons maintenant un cluster découpé en deux groupes qui ne peuvent plus communiquer entre eux car le maître est tombé en panne.
Dans ce scénario, il existe un risque critique : chaque groupe pourrait croire que l'autre est tombé en panne ou n'est plus accessible et tenté de désigner un nouveau maître. On se retrouverait alors avec 2 maîtres simultanés ce qui est inacceptable pour la cohérence du système.
Pour éviter cela, on autorise uniquement la grappe qui continent la majorité des nœuds à fonctionner. C'est la stratégie adoptée par MongoDB

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

Architecture maître- esclave par exemple MongoDB

MongoDB a une architecture maître-esclave ou primary secondary.
Toutes les requêtes en lecture ou en écriture sont envoyées au serveur principal (primary) qui s'occupe de propager aux autres serveurs (secondary) afin d'assurer la cohérence des données.
Dans MongoDB, les écritures se font toujours sur le nœud principal (primary) car la réconciliation des données et la gestion des conflits sont des processus complexes. En centralisant les écritures, MongoDB garantit la cohérence des données.
Les écritures, par défaut aussi, se font sur le primary ce qui permet de maintenir une cohérence forte.
Autrement dit, tant qu'on lit et écrit sur le nœud principal, on est sûr de toujours avoir accès à la dernière version des données. Cependant, pour le passage à l'échelle et pour répartir la charge, on peut configurer MongoDB afin d'autoriser les lectures sur les réplicas secondaires.

Réponses aux questions des parties du TP

Pour réaliser le tp, il faut suivre les vidéos suivantes qui montrent les manipulations à refaire :
Vidéo 1 :
ReplicationMongoDB1

Vidéo 2 :
ReplicationMongoDB2

Ensuite, il faut installer MongoDB comme suit:
après avoir cliqué sur le lien, Download MongoDB Community Server | MongoDB

vous devriez choisir comme option :
Version: la plus récente
OS : windows
Package: MSI

Voici le lien de la documentation MongoDB Compass : https://www.mongodb.com/docs/manual

Manipulations de la vidéo 1

Pour ce TP, je travaille en environnement windows et en utlisant MongoDB Compass comme client mongo.

Voici les différentes étapes afin de mettre en place une architecture maître-esclave avec mongoDB comme présenté dans la vidéo 1:
Pour une architecture et un replicatset de 3 esclaves et un maître :

- Créer les répertoires pour chaque esclave
  mkdir C:\Users\cloud\disque1
  mkdir C:\Users\cloud\disque2
  mkdir C:\Users\cloud\disque3

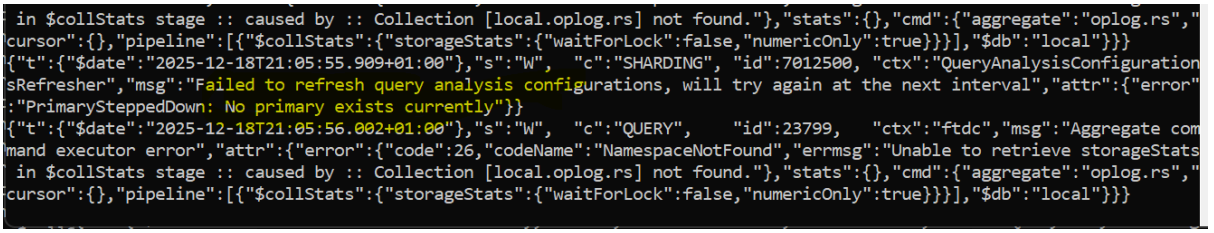
- Lancer les 3 nœuds, il faut ouvrir 3 terminaux ou fenêtres PowerShell séparées en changeant les numéros des ports pour chaque noeuds :
  & "C:\Program Files\MongoDB\Server\8.2\bin\mongod.exe" --replSet monreplicaset --port 27020 --dbpath C:\Users\cloud\disque1

  & "C:\Program Files\MongoDB\Server\8.2\bin\mongod.exe" --replSet monreplicaset --port 27021 --dbpath C:\Users\cloud\disque2

  & "C:\Program Files\MongoDB\Server\8.2\bin\mongod.exe" --replSet monreplicaset --port 27022 --dbpath C:\Users\cloud\disque3

NB: Il est important de mettre le même nom du replica set après l'option –replSet. Car ce nom (ici monreplicatset) va permettre de regrouper plusieurs serveurs dans un même graphe de replica set.

Une fois que les trois serveurs sont démarrés, on remarque le replica set n'est pas encore initialisé. Ce qui est normal car l'option –replSet indique qu'un serveur est prêt à rejoindre un replica set.



Pour initialiser un replica set, on utilisera rs.initiate() dans un serveur. A voir dans la suite du TP.

- Connecter le client (mongosh) au serveur 1
  Ouvrir une 4e fenêtre(terminal ou PowerShell)

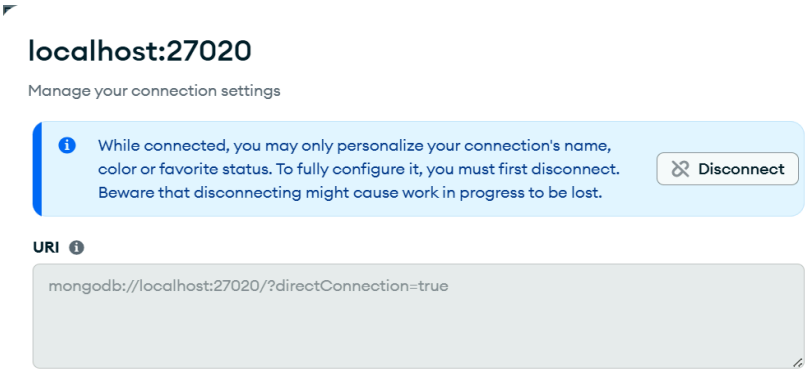
Si vous utilisez un environnement linux, exécutez la commande suivante :

./mongo -port 27020

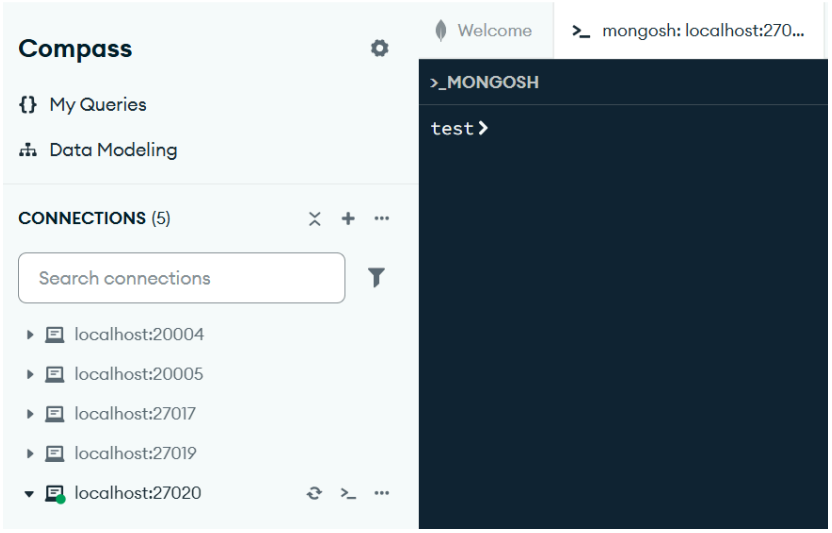
Si vous utilisez un environnement Windows, exécutez la commande suivante:

& "C:\Program Files\MongoDB\Server\8.2\bin\mongosh.exe" --port 27020

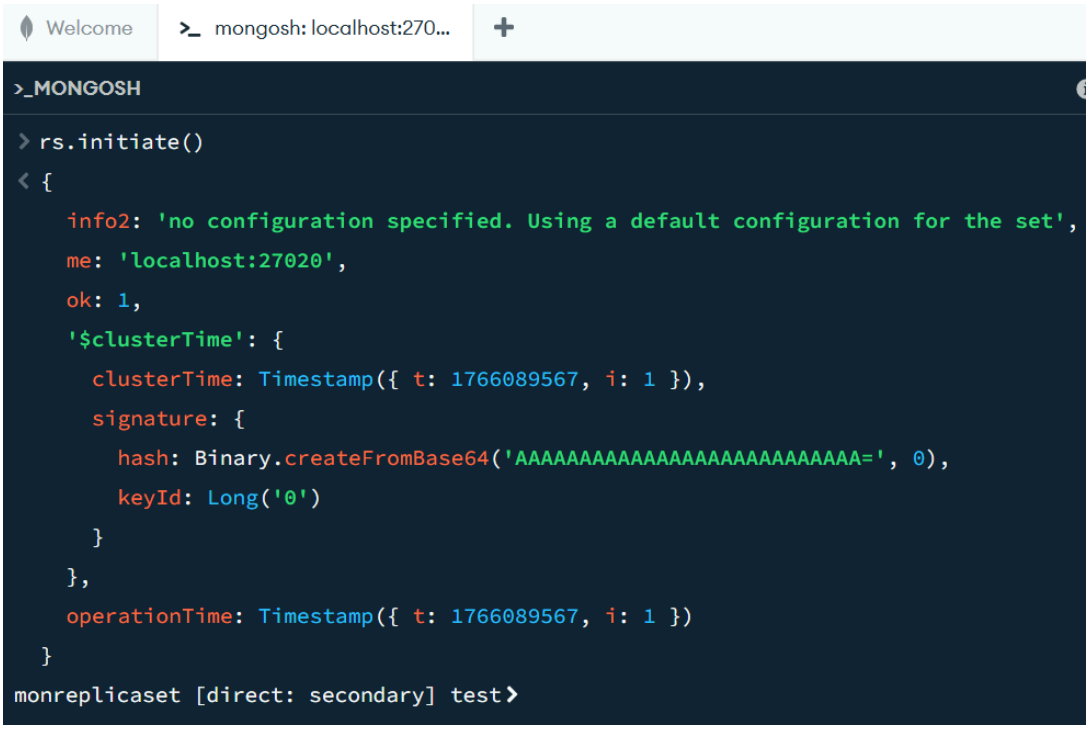
Dans mon cas, j'utilise le client MongoDB Compass, donc je crée une nouvelle connection comme suit dans le client MongoDB Compass:



Une fois, la connexion établie, on observe l'invite de commande du client Mongo comme suit :



La connexion au serveur effectuée, on peut initialiser le replica set (dans mongo db compass) avec la commande : rs.initiate()



La grappe est alors mise en place, on peut maintenant ajouter les deux autres serveurs à la même grappe.
Pour cela, on récupère le nom de la machine. Dans l'image ci-dessus, elle se trouve dans le champ "me" : localhost:27020
On va ajouter le noeud 2 à la grappe avec la commande rs.add('localhost:27021')

On remarque alors que dans le terminal 2, il y'a un échange de message :

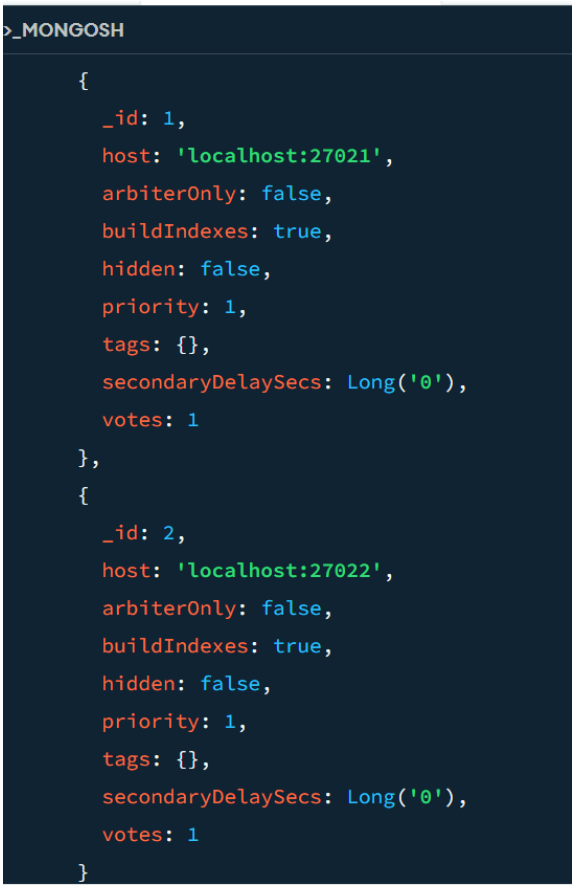
Ensuite, on rajoute le serveur 3: rs.add('localhost:27022')
On remarque que c'est le noeud 27020 qui est le primary (dernière ligne sur l'image):

```
> rs.add('localhost:27022')

< {
    ok: 1,
    '$clusterTime': {
      clusterTime: Timestamp({ t: 1766090028, i: 1 }),
      signature: {
        hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
        keyId: Long('0')
      }
    },
    operationTime: Timestamp({ t: 1766090028, i: 1 })
  }
monreplicaset [direct: primary] test>
```

Pour afficher la configuration du replica set, la commande est : rs.config()
C'est une commande utile pour vérifier par exemple que les nœuds sont bien dans la grappe. On retrouve comme information :
le nom du replica set dans _id, le numéro de la version de la configuration qui est incrémenté à chaque modification : ajout, suppression d'un membre.
Members est un tableau contenant la configuration de chaque nœud.
Host contient de l'adresse de chaque noeud avec le port
Priority, le poids du nœud dans l'élection du primary. Plus la priorité est haute, plus il a des chances d'être élu primary. Si priority est égale à 0, le nœud ne pourra jamais être élu maître.


Welcome
mongosh: localhost:270...
>_MONGOSH
> rs.config()
< {
_id: 'monreplicaset',
version: 5,
term: 1,
members: [
{
_id: 0,
host: 'localhost:27020',
arbiterOnly: false,
buildIndexes: true,
hidden: false,
priority: 1,
tags: {},
secondaryDelaySecs: Long('0'),
votes: 1
},


>_MONGOSH
{
_id: 1,
host: 'localhost:27021',
arbiterOnly: false,
buildIndexes: true,
hidden: false,
priority: 1,
tags: {},
secondaryDelaySecs: Long('0'),
votes: 1
},
{
_id: 2,
host: 'localhost:27022',
arbiterOnly: false,
buildIndexes: true,
hidden: false,
priority: 1,
tags: {},
secondaryDelaySecs: Long('0'),
votes: 1
}

La commande suivante permet de voir le statut de chaque nœud primary ou secondary
rs.status().
On observe aussi entre autres:

-Le health:1 signifie que le nœud est en ligne. Si health est égal à 0, le noeud est hors ligne ou tombé en panne

-Le uptime est le temps de fonctionnement. On remarque que les 3 nœuds ont commencé à fonctionner du premier au troisème.

Voici le résultat de rs.status() en détail :

Pour le noeud 1

```
>_MONGOSH
    members: [
      {
        _id: 0,
        name: 'localhost:27020',
        health: 1,
        state: 1,
        stateStr: 'PRIMARY',
        uptime: 2990,
        optime: [Object],
```

Pour le noeud 2

```
>_MONGOSH
    {
      _id: 1,
      name: 'localhost:27021',
      health: 1,
      state: 2,
      stateStr: 'SECONDARY',
      uptime: 1329,
      optime: [Object],
```

Pour le noeud 3

La commande rs.isMaster()
rs.isMaster()
permet de savoir si le nœud connecté est le primary ou pas.
Comme dans l'image, les champs primary et me ont la même valeur, on en déduit que le nœud 27020 est le primary.
On peut voir aussi, la liste des membres dans le champs hosts et le nom du replica set dans le champ setName

Par rapport au partitionnement réseau, on doit rajouter un nœud arbitre.
Il faut ainsi, définir un nouveau répertoire de l'arbitre

mkdir C:\Users\cloud\arbitre

Ensuite, démarrer un autre noeud dans un nouveau terminal qui va écouter dans le port 27023

& "C:\Program Files\MongoDB\Server\8.2\bin\mongod.exe" --replSet monreplicaset --port 27023 --dbpath C:\Users\cloud\arbitre

Pour le rajouter, dans le client mongo, on tape :
rs.addArb('localhost:27023')

```
> rs.addArb('localhost:27023')
< {
    ok: 1,
    '$clusterTime': {
      clusterTime: Timestamp({ t: 1766092829, i: 2 }),
      signature: {
        hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
        keyId: Long('0')
      }
    },
    operationTime: Timestamp({ t: 1766092829, i: 2 })
  }
monreplicaset [direct: primary] test>
```

Manipulations de la vidéo 2

On repart de la configuration précédente avec un maître et deux esclaves.
Dans le client Mongo, toujours connecté au port 27020, on crée une collection qui s'appelle demo1 comme suit:

```
> use demo1
< switched to db demo1
> db.createCollection("personnes")
< { ok: 1 }
```

Ensuite, on insère 4 éléments
Sur Mongo Compass, la fonction insert est dépréciée donc j'ai utilisé insertOne() à la place. Mais si vous utilisez un terminal bash, la fonction insert() fonctionne.

```
> db.personnes.insert({"prenom":"gloire"})
< DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany,
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('6944735f595c58ad7856a636')
    }
  }
> db.personnes.insertOne({"prenom":"gloire"})
< {
    acknowledged: true,
    insertedId: ObjectId('69447377595c58ad7856a637')
  }
> db.personnes.insertOne({"prenom":"yanis"})
< {
    acknowledged: true,
    insertedId: ObjectId('694473a2595c58ad7856a638')
  }
> db.personnes.insertOne({"prenom":"olivier"})
```

Pour lire les éléments, on utilise la fonction find:

```
> db.personnes.find()
< {
    _id: ObjectId('6944735f595c58ad7856a636'),
    prenom: 'gloire'
  }
  {
    _id: ObjectId('69447377595c58ad7856a637'),
    prenom: 'gloire'
  }
  {
    _id: ObjectId('694473a2595c58ad7856a638'),
    prenom: 'yanis'
  }
  {
    _id: ObjectId('694473ac595c58ad7856a639'),
    prenom: 'olivier'
  }
monreplicaset [direct: primary] demo1>
```

Avec les fonctions find et insert (ou insertOne), on arrive à lire sur le primary.
Avec MongoDB, les paramètres par défaut permettent d'assurer une cohérence forte où l'écriture et la lecture se font par défaut sur le port primary et on ne peut jamais écrire sur les secondary.
Par contre, on peut forcer et lire directement à partir des secondary avec le risque de récupérer des données incohérentes car MongoDB procède à une réplication asynchrone.
Lorsqu'un client envoie une requête d'insertion, le serveur écrit dans son journal et envoie un acquittement. A partir du moment où l'acquittement est parti, il commence à initialiser la réplication.
Entretemps, une application se connecte à un secondary et récupère la donnée. Elle pourrait récupérer une donnée qui n'est pas à jour.

A présent, nous allons nous connecter le client au port 27021:

```
>_MONGOSH
> use demo1
< switched to db demo1
> show collections
< personnes
> db.personnes.find()
< {
    _id: ObjectId('6944735f595c58ad7856a636'),
    prenom: 'gloire'
  }
  {
    _id: ObjectId('69447377595c58ad7856a637'),
    prenom: 'gloire'
  }
  {
    _id: ObjectId('694473a2595c58ad7856a638'),
    prenom: 'yanis'
  }
  {
    _id: ObjectId('694473ac595c58ad7856a639'),
    prenom: 'olivier'
  }
```

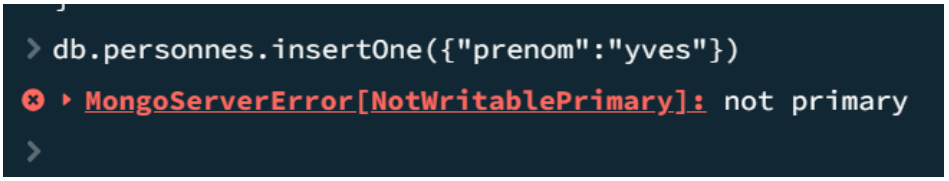
On remarque qu'on accède à demo1, à la collection personnes et on peut lire les personnes.,

Notons que :

- Quand on a inséré, le port utilisé est le 27020. Mais après la connexion sur le 27021, on observe bien les données. Cela signifie que les données ont été bien répliquées. Cela est normal car le 27021, même si ce n'est pas le primary, appartient au même replica set que le 27020.
- On ne peut pas lire sur un secondary sauf si on force la valeur du slay à true.

Mais l'écriture reste impossible sur un secondary.

Pour prouver cela, essayons d'insérer un élément via le secondary comme suit :

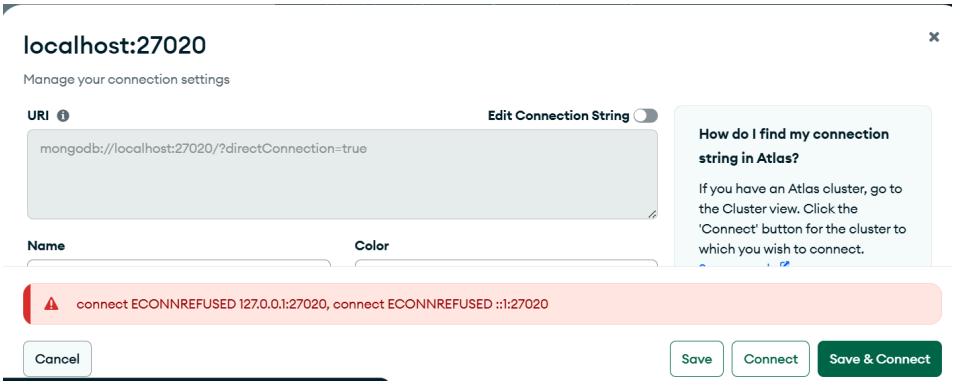


La réponse "not primary" montre qu'on ne peut pas écrire car il s'agit d'un secondary.

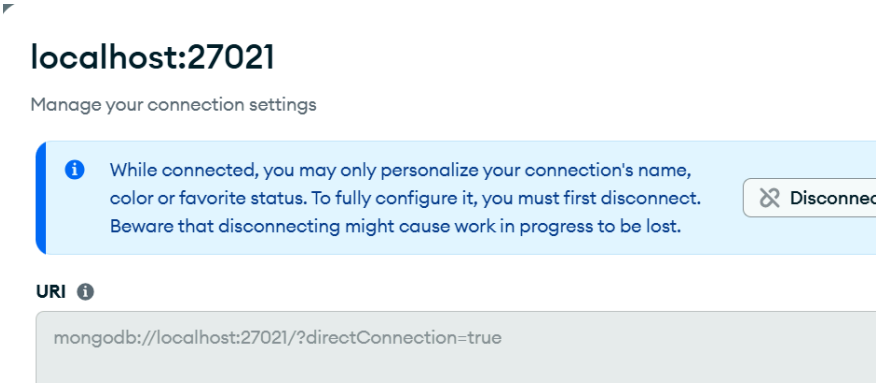
- Simulation d'une panne:

Pour se faire, on arrête le serveur 27020 avec un CTRL+C

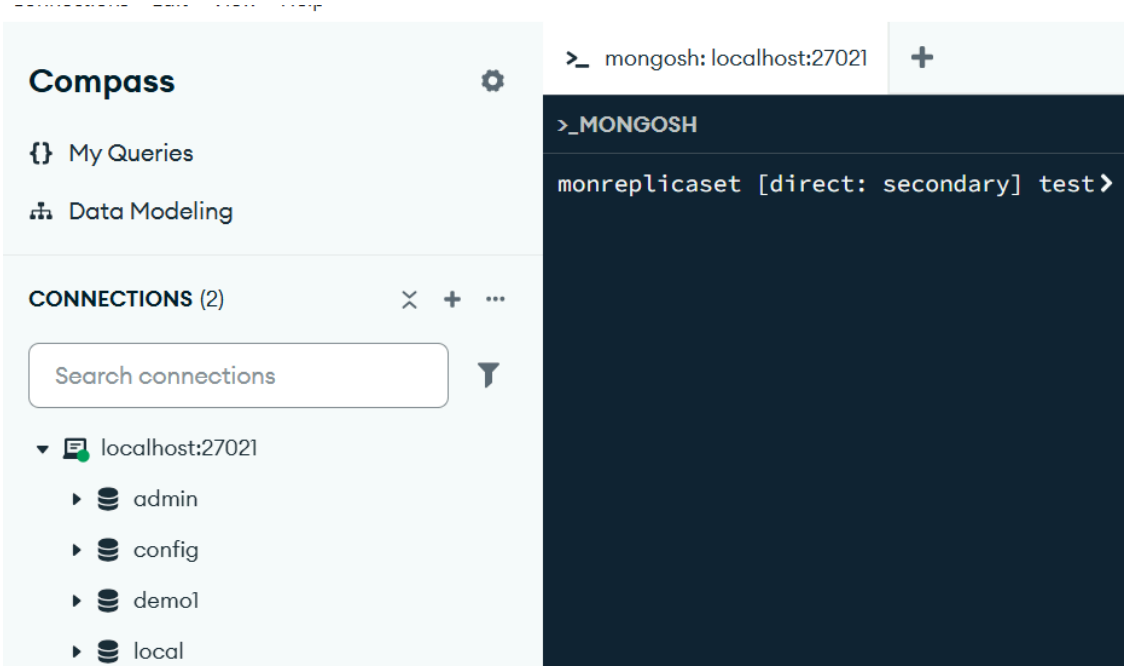
On essaie de connecter le client mongo au serveur 27020
La connexion a échoué sur le port 27020 car le serveur 1 n'est plus en cours d'exécution.



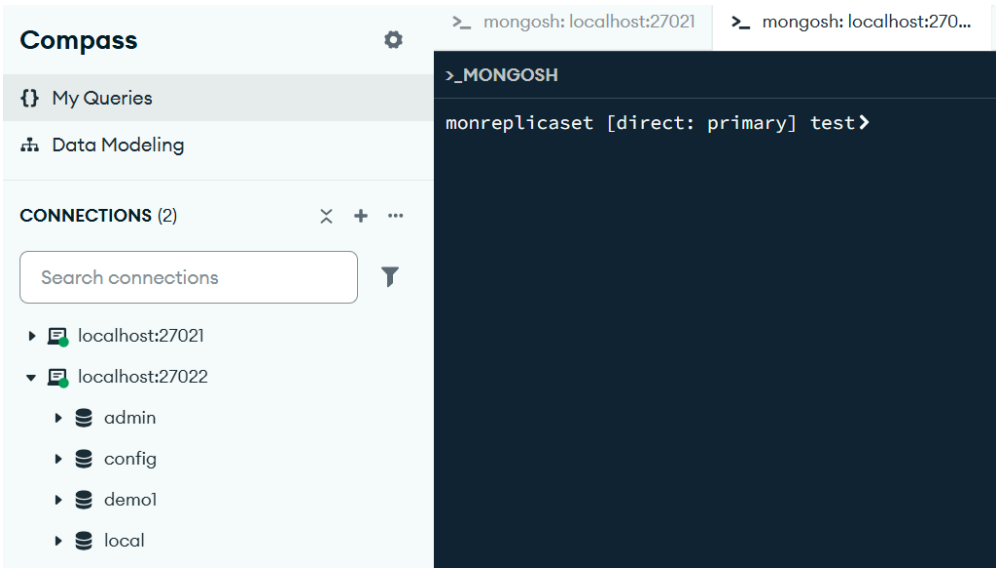
Quand on se reconnecte au serveur 27021, la connexion est établie:



Maintenant, vérifions lequel des deux noeuds secondary a été élu :
Noeud 2 - port 27021
La connexion est rétablie mais il est toujours secondary



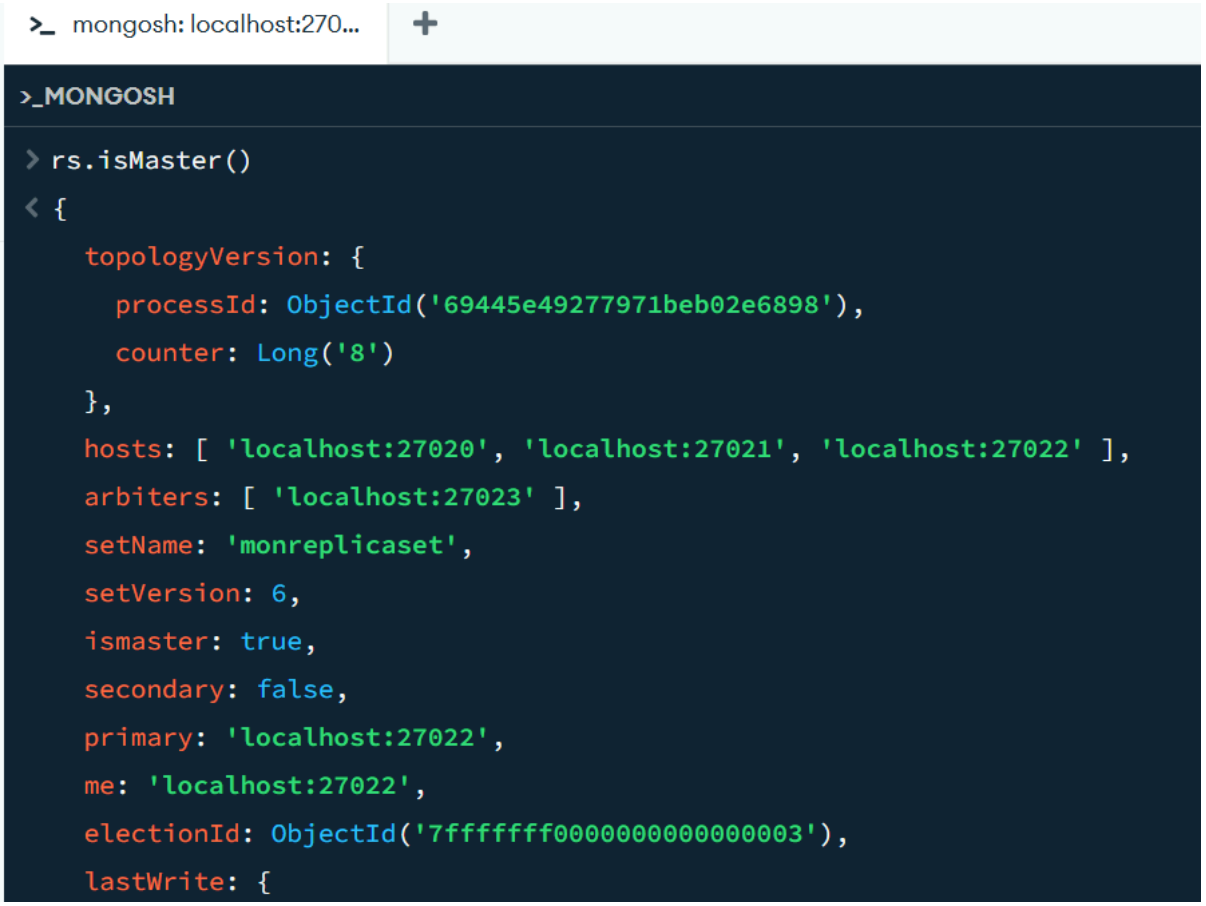
Noeud 3 - port 27022
La connexion est rétablie et il est devenu primary



Sur le serveur 2, on peut lire les données:

Enfin, en utilisant la commande rs.isMaster() apprise plus-haut, on peut confirmer que le Noeud sur le port 27022 est devenu le nouveau Primary :

le champ isMaster: true et les champs primary et me sont égaux.



Réponses aux questions sur la réplication et tolérance aux pannes avec MongoDB

Partie 1 : Compréhension de base

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

Un Replica Set est un ensemble de serveurs MongoDB contenant la même copie d'un jeu de données. Cela permet d'avoir : la tolérance aux pannes (haute disponibilité), la réplication automatique des données ainsi que l'élection automatique d'un nouveau Primary en cas de défaillance.

Il est constitué de :

- 1 Primary ou serveur principal (qui reçoit toutes les requêtes en écriture et en lecture)

- 1 ou plusieurs Secondaries ou serveur secondaire (en charge de la réplication et de la lecture optionnelle)

- un Arbiter (vote mais ne stocke pas de données).

2. Rôle du Primary dans un Replica Set

Le Primary est le seul nœud qui accepte les écritures.
Il traite les opérations d'écriture, fournit des lectures cohérentes, envoie son oplog aux Secondaries pour la réplication.

3. Quel est le rôle essentiel des Secondaries ?

Les Secondaries répliquent les données en lisant l'oplog du Primary et peuvent répondre à des lectures (si readPreference le permet). Aussi, ils peuvent devenir Primary lors d'une élection.

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

MongoDB n'autorise pas les écritures sur un Secondary car la réplication est asynchrone. Cela crée un risque de conflits.
C'est le Primary qui uniquement doit écrire pour assurer la cohérence et l'ordre global, et éviter les divergences de données entre nœuds.

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

La forte cohérence signifie que toute lecture provenant du Primary renvoie la version la plus récente des données, les écritures sont immédiatement visibles sur ce nœud.

6. Différence entre readPreference: "primary" et "secondary"

- "primary" : la lecture se fait strictement sur Primary (cohérence forte).

- "secondary" :si la lecture s'effectue sur un Secondary, il y'a un risque d'avoir des données potentiellement obsolètes.

7. Dans quel cas lire sur un Secondary malgré les risques ?

On lit sur un Secondary malgré les risques lorsqu'il faut répartir la charge de lecture, faire des requêtes analytiques, lire des données anciennes (avec slaveDelay), faire des backups.

Partie 2 : Commandes & configuration

8. La commande pour initialiser un Replica Set

rs.initiate()

9. La commande pour ajouter un nœud après initialisation

rs.add("hostname:port")

10. La commande pour afficher l'état actuel du Replica Set

rs.status()

11. La commande pour identifier le rôle d'un nœud

Dans le résultat de la rs.status(), regarder :

- "stateStr" : "PRIMARY"
- "stateStr" : "SECONDARY"
- "stateStr" : "ARBITER"

Ou :

db.isMaster()

12. La commande pour forcer le basculement du Primary

rs.stepDown()

13. La commande pour désigner un nœud comme Arbitre ? Pourquoi ?

rs.addArb("host:port")

On désigne un nœud comme Arbitre car il ne stocke pas de données, il sert uniquement à donner un vote de plus pour éviter les égalités lors des élections.

Enfin, il est utile pour atteindre une majorité impaire à faible coût.

14. La commande pour configurer un Secondary avec un slaveDelay:

```
cfg = rs.conf()
cfg.members[1].slaveDelay = 120
rs.reconfig(cfg)
```

Partie 3 : Résilience et tolérance aux pannes

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

Si cela arrive, aucun nouveau Primary ne sera élu. Le Replica Set passe en mode lecture seule. Enfin, toutes les écritures échouent (NotWritablePrimary).

16. Comment MongoDB choisit un nouveau Primary ?

MongoDB choisit un nouveau Primary en se basant sur ces critères :

1. Eligibilité du nœud (priority > 0)
2. Vérification si le nœud est le plus à jour (oplog)
3. Vérification de la latence et de la connectivité
4. Vérification de la priorité configurée

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

C'est le processus automatique lors duquel les nœuds votent pour désigner un nouveau Primary en cas de panne ou d'indisponibilité du précédent.

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

Un nœud se dégrade (cela signifie se rétrograde en Secondary) lorsqu'il perd la majorité des votes. Ainsi, il ne peut plus être Primary.

Cela survient lors de partitions réseau.

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 afin d'éviter les égalités lors des élections et les blocages liés au manque de majorité.

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

Les conséquences d’une partition réseau sur le fonctionnement du cluster sont :

Selon le côté :

- le côté avec majorité élit un Primary,
- pour le côté sans majorité : la lecture est possible et l’écriture est impossible.

Partie 4 : Scénarios pratiques

21. 3 nœuds : Primary, Secondary, Arbitre ; Primary devient injoignable

Le Secondary + Arbitre = 2 votes = majorité → un nouveau Primary est élu.

Le cluster reste entièrement fonctionnel.

22. L’utilité d’un Secondary avec slaveDelay = 120

Il possède volontairement un retard de 2 minutes.

Cela est utile pour :

- la récupération après erreur humaine,
- la restauration d’un document supprimé,
- l’audit ou l’analyse d’évolution.

23. Un client exige une lecture toujours à jour, même en cas de bascule

Je recommanderai comme option :

- readConcern : "majority" qui assure que la donnée soit lue est validée par la majorité

- writeConcern : { w: "majority" } qui assure que la donnée reste cohérente même après une élection.

24. Afin de garantir que l'écriture est confirmée par au moins 2 nœuds, vous devez utiliser w de writeConcern :

{ writeConcern: { w: 2 } }

25. Un étudiant lit depuis un Secondary et obtient une donnée obsolète, pourquoi ?

Cela se produit parce que la réplication est asynchrone, le Secondary peut être en retard.

Pour éviter cela, il faut spécifier :

- readPreference: "primary"
- ou readConcern: "majority"

Ou bien il faut configurer un Secondary pour être plus rapide.

26. La commande pour savoir quel nœud est Primary

rs.status()

Ou :

db.isMaster()

27. Pour forcer une bascule manuelle sans interruption majeure, il faut exécuter sur le Primary la commande :

rs.stepDown(60)

28. La procédure pour ajouter un nouveau Secondary dans un Replica Set en fonctionnement consiste à exécuter la commande :

rs.add("newhost:port")

29. La commande pour retirer un nœud défectueux

rs.remove("host:port")

30. La commande pour configurer un nœud Secondary caché

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

On ferait cela pour que le nœud Secondary ne soit jamais élu, ne soit pas utilisé par les clients, idéal pour analytique ou sauvegarde.

31. Pour modifier la priorité d'un nœud pour qu'il devienne préféré comme Primary, il faut exécuter les commandes suivantes dans cet ordre :

```
cfg = rs.conf()
cfg.members[i].priority = 2
rs.reconfig(cfg)
```

Plus la priorité est haute, plus le nœud sera élu.

32. Pour vérifier le délai de réplication d'un Secondary par rapport au Primary :

Avec les commandes : rs.printReplicationInfo() ou rs.printSlaveReplicationInfo()
on voit le retard en secondes.

33. La commande rs.freeze() empêche le nœud de devenir Primary pour une durée donnée.

Par exemple, en précisant le temps en secondes comme paramètre
rs.freeze(seconds)
Cela est utile lors de la maintenance.

34. Pour redémarrer un Replica Set sans perdre la configuration, il faut arrêter MongoDB et redémarrer normalement. Ainsi, la configuration est automatiquement rechargée car elle stockée dans la base locale.

35. Pour surveiller la réplication en temps réel :

Il faut soit utiliser les logs MongoDB (mongod.log) ou bien exécuter les commandes Shell suivantes :

rs.status()

rs.printSlaveReplicationInfo()

db.printReplicationInfo()

Questions complémentaires :

37. Qu'est-ce qu'un Arbitre ? Pourquoi ne stocke-t-il pas de données ?

Un Arbitre est un nœud qui sert uniquement à voter lors des élections. Il n'a pas d'oplog, ni de stockage donc la perte de données est impossible. Enfin, il consomme très peu de ressources.

38. La commande pour vérifier la latence de réplication :

rs.printSlaveReplicationInfo()

39. La commande pour afficher le retard de réplication est la même commande que pour la question précédente :

rs.printSlaveReplicationInfo()

40. La différence entre la réplication asynchrone et synchrone

Pour la réplication synchrone : les nœuds appliquent immédiatement toutes les opérations. Cela est lent.

Pour la réplication asynchrone : les nœuds appliquent la mise à jour plus tard.

MongoDB utilise une réplication asynchrone.

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

Oui.

En utilisant la commande : rs.reconfig(cfg), on n'a pas besoin d'arrêter les nœuds.

42. Si un Secondary a plusieurs minutes de retard :

- Il ne peut pas être élu Primary
- Les lectures peuvent retourner des données obsolètes
- Le retard doit être rattrapé via son oplog.

43. Pour gérer des conflits de données lors de la réplication MongoDB :

- applique les opérations dans l'ordre de l'oplog
- le gagnant est le nœud devenant Primary
- les opérations du Primary sont source de vérité. Il ne peut pas avoir de multi-master.

44. Plusieurs Primarys en même temps ?

Non.
Cela est possible grâce au système de votes et à la majorité. On ne peut avoir qu'un seul Primary possible.

45. Pourquoi est-il déconseillé d'utiliser un Secondary pour écrire ?

Cela est déconseillé parce que MongoDB rejette toute écriture autre que sur le Primary. Même utiliser un Secondary en readPreference ne change rien, l'écriture est impossible..

46. Les conséquences d'un réseau instable sur un Replica Set :

- élections fréquentes
- pertes de majorité car le cluster en lecture seule
- retards de réplication
- dégradations automatiques
- risques de données obsolètes côté clients.