Seyfeddine Jouini 3ING

TP 2 : Réplication avec MongoDB

Vidéo 1 : https://www.youtube.com/watch?v=ytCv2EX37Q8
Grappe de serveur = un ensemble de serveur.
Dans MongoDB, l'écriture et lecture se font toujours sur le noeud maître pour :
La réconciliation des données et la gestion des conflits des processus complexes.

La réplication de MONGODB est asynchrone.

Pour la montée en charge, MONGODB utilise le sharding/Partitionnement.

Pour l'élection, MongoDB se base sur la majorité.
Si l'un des groupes n'a pas de majorité, on va avoir un arbitre pour avoir une majorité pour l'élection en cas de panne.

On va créer des répertoires pour mettre dedans les données.
On va avoir 3 serveur et un client :
et pour chaque serveur on va créer un dossier disque**i** avec i 1->3:

```
 Répertoire de C:\Users\jouin\Desktop\TP NOSQL\partitionnement

04/12/2025  14:17    <DIR>          .
04/12/2025  14:15    <DIR>          ..
04/12/2025  14:28    <DIR>          disque1
04/12/2025  14:28    <DIR>          disque2
04/12/2025  14:28    <DIR>          disque3
               0 fichier(s)                0 octets
               5 Rép(s)  10 489 544 704 octets libres
```

On démarre les serveurs chacun sur un port de 27018 à 27020 :
mongod --replSet monreplicaset --port 27018 --dbpath disque1

Ici, on remarque que les replica ne sont pas initialisés, donc il faut les initialiser à partir d'un serveur avec la commande :
on se connecte : **mongosh –port 27018**
puis : **rs.initiate()**
on se connecte à partir du client au premier serveur sur le port 27018 si on met le port par défaut ça va être 27017.

En étant connecté à 27018, on va rajouter les autres serveurs : **rs.add("localhost:270..")**

```
test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for t
  me: 'localhost:27018',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764855826, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764855826, i: 1 })
}
monreplicaset [direct: secondary] test> rs.add("localhost:27019")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764855998, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764855998, i: 1 })
}
monreplicaset [direct: primary] test> rs.add("localhost:27020")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764856004, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764856004, i: 1 })
}
monreplicaset [direct: primary] test>
```

On ajoutant à partir du serveur 27018, il devient primary (maître).

Pour voir la structure du replicaset :
rs.config()

```
  },
  operationTime: Timestamp({ t: 1764856004, i: 1 })
}
monreplicaset [direct: primary] test> rs.config()
{
  _id: 'monreplicaset',
  version: 4,
  term: 1,
  members: [
    {
      _id: 0,
      host: 'localhost:27018',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'localhost:27019',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 2,
      host: 'localhost:27020',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
```

Afficher l'état complet du replica set :
Ceci se fait avec la commande **rs.status() :**

```
monreplicaset [direct: primary] test> rs.status()
{
  set: 'monreplicaset',
  date: ISODate('2025-12-06T19:04:14.078Z'),
  myState: 1,
  term: Long('3'),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long('2000'),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1765047853, i: 1 }), t: Long('3') },
    lastCommittedWallTime: ISODate('2025-12-06T19:04:13.271Z'),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1765047853, i: 1 }), t: Long('3') },
    appliedOpTime: { ts: Timestamp({ t: 1765047853, i: 1 }), t: Long('3') },
    durableOpTime: { ts: Timestamp({ t: 1765047853, i: 1 }), t: Long('3') },
    writtenOpTime: { ts: Timestamp({ t: 1765047853, i: 1 }), t: Long('3') },
    lastAppliedWallTime: ISODate('2025-12-06T19:04:13.271Z'),
    lastDurableWallTime: ISODate('2025-12-06T19:04:13.271Z'),
    lastWrittenWallTime: ISODate('2025-12-06T19:04:13.271Z')
  },
  lastStableRecoveryTimestamp: Timestamp({ t: 1765047793, i: 1 }),
  electionCandidateMetrics: {
    lastElectionReason: 'electionTimeout',
    lastElectionDate: ISODate('2025-12-06T19:02:53.199Z'),
    electionTerm: Long('3'),
    lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 0, i: 0 }), t: Long('-1') },
    lastSeenWrittenOpTimeAtElection: { ts: Timestamp({ t: 1764864933, i: 1 }), t: Long('1') },
    lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1764864933, i: 1 }), t: Long('1') },
    numVotesNeeded: 2,
    priorityAtElection: 1,
    electionTimeoutMillis: Long('10000'),
    numCatchUpOps: Long('0'),
    newTermStartDate: ISODate('2025-12-06T19:02:53.221Z'),
    wMajorityWriteAvailabilityDate: ISODate('2025-12-06T19:02:53.320Z')
  },
```

Afin de connaître si le noeud sur lequel on est connecté est primary (master) ou pas : on exécute la commande suivante : **rs.imMaster()**

```
monreplicaset [direct: primary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId('69347dbc52a3da4a5d29414b'),
    counter: Long('7')
  },
  hosts: [ 'localhost:27018', 'localhost:27019', 'localhost:27020' ],
  setName: 'monreplicaset',
  setVersion: 3,
  ismaster: true,
  secondary: false,
  primary: 'localhost:27018',
  me: 'localhost:27018',
  electionId: ObjectId('7fffffff0000000000000003'),
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1765048003, i: 1 }), t: Long('3') },
    lastWriteDate: ISODate('2025-12-06T19:06:43.000Z'),
    majorityOpTime: { ts: Timestamp({ t: 1765048003, i: 1 }), t: Long('3') },
    majorityWriteDate: ISODate('2025-12-06T19:06:43.000Z')
  },
  maxBsonObjectSize: 16777216,
  maxMessageSizeBytes: 48000000,
  maxWriteBatchSize: 100000,
  localTime: ISODate('2025-12-06T19:06:52.328Z'),
  logicalSessionTimeoutMinutes: 30,
```

Comme le montre la capture ci-dessous : la réponse à la requête nous dit que le noeud est un master.

Comme on le sait, MONGODB a une forte cohérence => on ne peut écrire que sur le master. Cependant, on a la possibilité d'écrire aussi sur un slave.

Création d'un arbitre pour avoir une majorité lors d'une panne.
Ceci se fait avec : **rs.addArb("localhost:27021")**

Vidéo 2 : https://www.youtube.com/watch?v=QhYyi1frKMc
Passant maintenant à la manipulation de données dans le replica set :
on commence par créer une base “Demo1”, une collection “Personne” et on y insère quelques documents.

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

Insertion :

```
monreplicaset [direct: primary] demo1> db.Personne.insert({"nom" : "seyf"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693481be146d55fb99b5f899') }
}
monreplicaset [direct: primary] demo1> db.Personne.insert({"nom" : "nozha"})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693481c3146d55fb99b5f89a') }
}
monreplicaset [direct: primary] demo1> db.Personne.insert({"nom" : "nesrine"})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693481c7146d55fb99b5f89b') }
}
monreplicaset [direct: primary] demo1>
```

La lecture et l’écriture se fait que sur le maître :

```
monreplicaset [direct: primary] demo1> db.Personne.find()
[
  { _id: ObjectId('693481be146d55fb99b5f899'), nom: 'seyf' },
  { _id: ObjectId('693481c3146d55fb99b5f89a'), nom: 'nozha' },
  { _id: ObjectId('693481c7146d55fb99b5f89b'), nom: 'nesrine' }
]
monreplicaset [direct: primary] demo1>
```

On va essayer de lire à partir d’un slave :
si on se connecte sur un secondary exemple sur le 27019 avec : **mongosh –port 27019**
on ne peut pas lire que si on le force avec : **rs.primaryOK().**
Cependant, pour la lecture, on ne peut écrire que sur un maître (primary) comme le montre la figure ci-dessous :

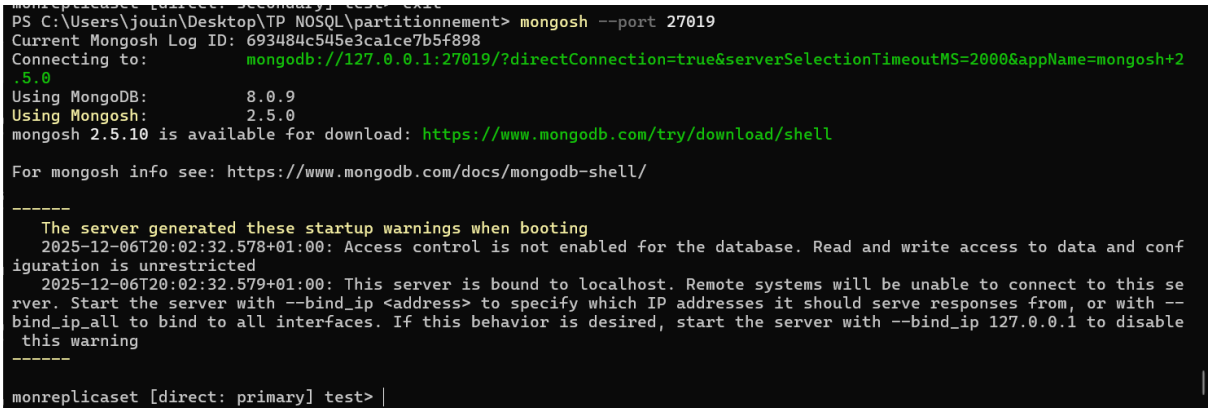
```
monreplicaset [direct: secondary] demo1> db.Personne.insert({"nom":"se"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
Uncaught:
MongoBulkWriteError[NotWritablePrimary]: not primary
Result: BulkWriteResult {
  insertedCount: 0,
  matchedCount: 0,
  modifiedCount: 0,
  deletedCount: 0,
  upsertedCount: 0,
  upsertedIds: {},
  insertedIds: { '0': ObjectId('69348375cc28e2ecd4b5f899') }
}
Write Errors: []
monreplicaset [direct: secondary] demo1>
```

Élection :

Si j'arrête le serveur maître : le 27018, il va y avoir une élection :

Si j’essaye de me connecter au 27018, on ne le peut pas :

Quand je me connecte à 27019, je remarque qu’il a été élu comme maître :



Quand je me connecte à 27020, je remarque qu’il n’a pas été élu comme maître :

De ce fait, je peux écrire et lire à partir de 27019.