TP2 :
Partie 1 :

Introduction : Principe de réplication :

Dans une grappe de serveurs, tous les nœuds sont connectés et échangent des messages. Cela signifie que le système repose sur une communication constante entre les différents nœuds, pour la réplication, vérification de l'état des nœuds etc garantissant que le système reste dans un état cohérent et réactif dans un environnement distribué.
On fait référence à une topologie maître esclave. Si un esclave tombe en panne, le maître s' en aperçoit suite à une absence de réponses. La panne reste donc locale d'où sa tolérance aux pannes. Si le maître tombe en panne, le système ne fonctionne plus normalement. On a recours à un algorithme d'élection ou les nœuds restants désignent un nouveau maître. Ce mécanisme permet au système d'être résilient et de s'auto organiser sans intervention humaine. Dans ce cas là, pour éviter toute incohérence, on autorise que la grappe qui contient la majorité des nœuds a continuer à fonctionner et donc ce groupe a élu un maître et continue son fonctionnement tandis que l'autre groupe attend la réparation du réseau.
Toutes les requêtes (lecture ou écriture) sont adressées au maître garantissant la cohérence. La réplication sur MongoDB est asynchrone. Pour la montée en charge ce n'est pas assuré par la réplication.
Pour éviter qu'aucun sous-ensemble de la grappe ne dispose de la majorité absolue, on ajoute un nœud arbitre qui ne sera jamais maître, qui ne sauvegarde rien mais qui participe au vote pour garantir la majorité absolue.
Si je lis depuis le maître je lis des données mise à jour sinon si je lis au niveau des esclaves je risque d'avoir des données obsolètes.

Comment simuler un replica set avec MongoDB ?

On définit une grappe de 3 serveurs et un répertoire de données dédiée pour chaque instance.

Creation du reseau pour le cluster :

```
PS C:\Users\Fei3> docker network create mongoCluster
a9388cb3775e8f2276f6837f2c362e39ce46f40b319800542b7e17fe1b4a9372
```

Lancement des 3 instances mango :

```
PS C:\Users\Fei3> docker run -d --rm -p 27017:27017 --name mongo1 --network mongoCluster mongo:5 mongod --replSet myReplicaSet --bind_ip localhost,mongo1
Unable to find image 'mongo:5' locally
5: Pulling from library/mongo
d9802f032d67: Pull complete
3d7e6a7004dc: Pull complete
0456147d1430: Pull complete
b48e1c442bc1: Pull complete
aeddebb0b1fc: Pull complete
a7a1a92aeac5: Pull complete
b076be86645a: Pull complete
16e97e335103: Pull complete
Digest: sha256:54bcd8da3ea5eec561b68c605046c55c6b304387dc4c2bf5b3a5f5064fbb7495
Status: Downloaded newer image for mongo:5
f43f5b230e24ae2dec81556d9ed588fe0e0bc17b907af8e64f0f441f479e8094
PS C:\Users\Fei3> docker run -d --rm -p 27018:27017 --name mongo2 --network mongoCluster mongo:5 mongod --replSet myReplicaSet --bind_ip localhost,mongo2
a1c6cb6738cf1a19622b5c34c160268a76215e904619f05a9c60e9d8d7340cb6
PS C:\Users\Fei3> docker run -d --rm -p 27019:27017 --name mongo3 --network mongoCluster mongo:5 mongod --replSet myReplicaSet --bind_ip localhost,mongo3
02fd6c7a357f69a6c5e176bd3739a64a1f636d227142e736ad7f30e2a820cf32
PS C:\Users\Fei3> docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                                                 NAMES
02fd6c7a357f   mongo:5   "docker-entrypoint.s…"   9 seconds ago    Up 9 seconds    0.0.0.0:27019->27017/tcp, [::]:27019->27017/tcp   mongo3
a1c6cb6738cf   mongo:5   "docker-entrypoint.s…"   20 seconds ago   Up 19 seconds   0.0.0.0:27018->27017/tcp, [::]:27018->27017/tcp   mongo2
f43f5b230e24   mongo:5   "docker-entrypoint.s…"   45 seconds ago   Up 43 seconds   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp   mongo1
```

Je me connecte a mon primary :

```
PS C:\Users\Fei3> docker exec -it mongo1 mongosh
Current Mongosh Log ID: 693191ed3cd186832c544ca6
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.8
Using MongoDB:          5.0.31
Using Mongosh:          2.3.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-12-04T13:37:35.545+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/pr
odnotes-filesystem
   2025-12-04T13:37:35.794+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------

myReplicaSet [direct: primary] test>
```

J’initialise mon replica set :

```
myReplicaSet [direct: primary] test> rs.initiate({ _id: \"rs0\", members: [ {_id:0, host: \"<host_ip>:27017\"}, {_id:1, host: \"<host_ip>:27018\"}, {_id:2,
host: \"<host_ip>:27019\"}]})
```

J’affiche l'état de mes conteneurs (j’ai du reprendre le tp après) :

```
rs0 [direct: primary] test> rs.status().members.map(m => ({ name: m.name, stateStr: m.stateStr }))
[
  { name: 'mongo1:27017', stateStr: 'SECONDARY' },
  { name: 'mongo2:27017', stateStr: 'SECONDARY' },
  { name: 'mongo3:27017', stateStr: 'PRIMARY' }
]
```

☐	●	mongo3	d1a7214a8ba0	mongo	27019:27017	2.66%	39 minutes ago
☐	●	mongo2	2a828e5ef990	mongo	27018:27017	1.94%	39 minutes ago
☐	●	mongo1	b4155e6b472d	mongo	27017:27017	2.15%	41 minutes ago

Comme j’utilise Docker, les répertoires sont déjà gérés automatiquement dans chaque conteneur.
J’affiche la configuration actuelle de mon replica set :

```
rs0 [primary] test> rs.config()
{
  _id: 'rs0',
  version: 91274,
  members: [
    {
      _id: 0,
      host: 'mongo1:27017',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'mongo2:27017',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
```

```
    {
      _id: 2,
      host: 'mongo3:27017',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ],
  protocolVersion: Long('1'),
  writeConcernMajorityJournalDefault: true,
  settings: {
    chainingAllowed: true,
    heartbeatIntervalMillis: 2000,
    heartbeatTimeoutSecs: 10,
    electionTimeoutMillis: 10000,
    catchUpTimeoutMillis: -1,
    catchUpTakeoverDelayMillis: 30000,
    getLastErrorModes: {},
    getLastErrorDefaults: { w: 1, wtimeout: 0 },
    replicaSetId: ObjectId('69319a29e9b1de6b63e090ff')
  }
}
```


Ici tous mes nœuds ont la même priorité (le même poids) si on doit voter un nouveau primary.
J’affiche le statut actuel de mon replica set :

```
rs0 [primary] test> rs.status()
{
  set: 'rs0',
  date: ISODate('2025-12-06T14:47:43.737Z'),
  myState: 1,
  term: Long('2'),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long('2000'),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    lastCommittedWallTime: ISODate('2025-12-06T14:47:41.316Z'),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    appliedOpTime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    durableOpTime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    writtenOpTime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    lastAppliedWallTime: ISODate('2025-12-06T14:47:41.316Z'),
    lastDurableWallTime: ISODate('2025-12-06T14:47:41.316Z'),
    lastWrittenWallTime: ISODate('2025-12-06T14:47:41.316Z')
  },
  lastStableRecoveryTimestamp: Timestamp({ t: 1765032404, i: 1 }),
  electionCandidateMetrics: {
    lastElectionReason: 'electionTimeout',
    lastElectionDate: ISODate('2025-12-06T13:53:44.552Z'),
    electionTerm: Long('2'),
    lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 0, i: 0 }), t: Long('-1') },
    lastSeenWrittenOpTimeAtElection: { ts: Timestamp({ t: 1764859040, i: 1 }), t: Long('1') },
    lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1764859040, i: 1 }), t: Long('1') },
    numVotesNeeded: 2,
    priorityAtElection: 1,
    electionTimeoutMillis: Long('10000'),
    numCatchUpOps: Long('0'),
    newTermStartDate: ISODate('2025-12-06T13:53:44.560Z'),
    wMajorityWriteAvailabilityDate: ISODate('2025-12-06T13:53:45.075Z')
  },
  members: [
    {
      _id: 0,
      name: 'mongo1:27017',
      health: 1,
      state: 2,
      stateStr: 'SECONDARY',
      uptime: 3249,
      optime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
      optimeDurable: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
      optimeWritten: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
      optimeDate: ISODate('2025-12-06T14:47:41.000Z'),
      optimeDurableDate: ISODate('2025-12-06T14:47:41.000Z'),
      optimeWrittenDate: ISODate('2025-12-06T14:47:41.000Z'),
      lastAppliedWallTime: ISODate('2025-12-06T14:47:41.316Z'),
      lastDurableWallTime: ISODate('2025-12-06T14:47:41.316Z'),
      lastWrittenWallTime: ISODate('2025-12-06T14:47:41.316Z'),
      lastHeartbeat: ISODate('2025-12-06T14:47:42.888Z'),
```

```
  lastHeartbeatRecv: ISODate('2025-12-06T14:47:43.610Z'),
  pingMs: Long('0'),
  lastHeartbeatMessage: '',
  syncSourceHost: 'mongo3:27017',
  syncSourceId: 2,
  infoMessage: '',
  configVersion: 91274,
  configTerm: -1
},
{
  _id: 1,
  name: 'mongo2:27017',
  health: 1,
  state: 2,
  stateStr: 'SECONDARY',
  uptime: 3249,
  optime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
  optimeDurable: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
  optimeWritten: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
  optimeDate: ISODate('2025-12-06T14:47:41.000Z'),
  optimeDurableDate: ISODate('2025-12-06T14:47:41.000Z'),
  optimeWrittenDate: ISODate('2025-12-06T14:47:41.000Z'),
  lastAppliedWallTime: ISODate('2025-12-06T14:47:41.316Z'),
  lastDurableWallTime: ISODate('2025-12-06T14:47:41.316Z'),
  lastWrittenWallTime: ISODate('2025-12-06T14:47:41.316Z'),
  lastHeartbeat: ISODate('2025-12-06T14:47:42.889Z'),
  lastHeartbeatRecv: ISODate('2025-12-06T14:47:43.610Z'),
  pingMs: Long('0'),
  lastHeartbeatMessage: '',
```

```
  syncSourceHost: 'mongo3:27017',
  syncSourceId: 2,
  infoMessage: '',
  configVersion: 91274,
  configTerm: -1
},
{
  _id: 2,
  name: 'mongo3:27017',
  health: 1,
  state: 1,
  stateStr: 'PRIMARY',
  uptime: 3603,
  optime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
  optimeDate: ISODate('2025-12-06T14:47:41.000Z'),
  optimeWritten: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
  optimeWrittenDate: ISODate('2025-12-06T14:47:41.000Z'),
  lastAppliedWallTime: ISODate('2025-12-06T14:47:41.316Z'),
  lastDurableWallTime: ISODate('2025-12-06T14:47:41.316Z'),
  lastWrittenWallTime: ISODate('2025-12-06T14:47:41.316Z'),
  syncSourceHost: '',
  syncSourceId: -1,
  infoMessage: '',
  electionTime: Timestamp({ t: 1765029224, i: 1 }),
  electionDate: ISODate('2025-12-06T13:53:44.000Z'),
  configVersion: 91274,
  configTerm: -1,
```

```
    uptime: 3603,
    optime: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    optimeDate: ISODate('2025-12-06T14:47:41.000Z'),
    optimeWritten: { ts: Timestamp({ t: 1765032461, i: 3 }), t: Long('2') },
    optimeWrittenDate: ISODate('2025-12-06T14:47:41.000Z'),
    lastAppliedWallTime: ISODate('2025-12-06T14:47:41.316Z'),
    lastDurableWallTime: ISODate('2025-12-06T14:47:41.316Z'),
    lastWrittenWallTime: ISODate('2025-12-06T14:47:41.316Z'),
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    electionTime: Timestamp({ t: 1765029224, i: 1 }),
    electionDate: ISODate('2025-12-06T13:53:44.000Z'),
    configVersion: 91274,
    configTerm: -1,
    self: true,
    lastHeartbeatMessage: ''
  }
 ],
 ok: 1,
 '$clusterTime': {
  clusterTime: Timestamp({ t: 1765032461, i: 3 }),
  signature: {
    hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
    keyId: Long('0')
  }
 },
 operationTime: Timestamp({ t: 1765032461, i: 3 })
}
```

Pour savoir si le noeud sur lequel je suis est primary ou secondary :

```
rs0 [primary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId('693433fcaaf2ef21d0373cbb'),
    counter: Long('7')
  },
  hosts: [ 'mongo1:27017', 'mongo2:27017', 'mongo3:27017' ],
  setName: 'rs0',
  setVersion: 91274,
  ismaster: true,
  secondary: false,
  primary: 'mongo3:27017',
  me: 'mongo3:27017',
  electionId: ObjectId('7fffffff0000000000000002'),
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1765032414, i: 1 }), t: Long('2') },
    lastWriteDate: ISODate('2025-12-06T14:46:54.000Z'),
    majorityOpTime: { ts: Timestamp({ t: 1765032414, i: 1 }), t: Long('2') },
    majorityWriteDate: ISODate('2025-12-06T14:46:54.000Z')
  },
  maxBsonObjectSize: 16777216,
  maxMessageSizeBytes: 48000000,
  maxWriteBatchSize: 100000,
  localTime: ISODate('2025-12-06T14:46:55.016Z'),
  logicalSessionTimeoutMinutes: 30,
  connectionId: 52,
  minWireVersion: 0,
  maxWireVersion: 27,
```

```
  minWireVersion: 0,
  maxWireVersion: 27,
  readOnly: false,
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1765032414, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1765032414, i: 1 }),
  isWritablePrimary: true
}
```

Dans sa configuration de base, MongoDB privilégie une cohérence forte et donc toute lecture ou écriture passe par le primary donc ce n'est pas possible d'écrire sur un nœud secondaire. Mais je peux forcer une lecture à partir d'un secondary.
Je crée un arbitre et je vérifie :

```
PS C:\Users\Fei3> docker run -d --name arbiter --network mongo-cluster mongo --replSet rs0 --bind_ip_all
98b8489edeaf0c68b9f33f35fe538d988d11e139710d7eb053e67c4dd27c83ff
PS C:\Users\Fei3> docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS             PORTS
               NAMES
98b8489edeaf   mongo     "docker-entrypoint.s…"   8 seconds ago   Up 8 seconds       27017/tcp
               arbiter
d1a7214a8ba0   mongo     "docker-entrypoint.s…"   2 days ago      Up About an hour   0.0.0.0:27019->27017/tcp, [::]:2
9->27017/tcp   mongo3
2a828e5ef990   mongo     "docker-entrypoint.s…"   2 days ago      Up About an hour   0.0.0.0:27018->27017/tcp, [::]:2
8->27017/tcp   mongo2
b4155e6b472d   mongo     "docker-entrypoint.s…"   2 days ago      Up About an hour   0.0.0.0:27017->27017/tcp, [::]:2
7->27017/tcp   mongo1
```

J'ajoute mon arbitre a mon replica set :

```
rs0 [direct: primary] test> rs.addArb("arbiter:27017")
```

```
rs0 [direct: primary] test> rs.status().members.map(m => ({ name: m.name, state: m.stateStr }))
[
  { name: 'mongo1:27017', state: 'SECONDARY' },
  { name: 'mongo2:27017', state: 'SECONDARY' },
  { name: 'mongo3:27017', state: 'PRIMARY' },
  { name: 'arbiter:27017', state: 'ARBITER' }
]
```

Je crée ma collection :

```
rs0 [primary] test> use demotest
switched to db demotest
rs0 [primary] demotest> db.createCollection("personnes")
{ ok: 1 }
```

J'insère des éléments dans ma collection :

```
rs0 [primary] demotest> db.personnes.insert({"nom":"Fatimazahra"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('693449f3bf8143239f9dc29d') }
}
```

```
rs0 [primary] demotest> db.personnes.insert({"nom":"Malo"})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('69344a17bf8143239f9dc29e') }
}
```

```
rs0 [primary] demotest> db.personnes.insert({"nom":"Amine"})
{
  acknowledged: true,
  insertedIds: { '0': ObjectId('69344a2fbf8143239f9dc29f') }
}
```

Pour afficher mes éléments :

```
rs0 [primary] demotest> db.personnes.find()
[
  { _id: ObjectId('693449f3bf8143239f9dc29d'), nom: 'Fatimazahra' },
  { _id: ObjectId('69344a17bf8143239f9dc29e'), nom: 'Malo' },
  { _id: ObjectId('69344a2fbf8143239f9dc29f'), nom: 'Amine' }
]
```

MongoDB procède à une réplication asynchrone donc si je force la lecture à partir d'un secondary, je peux récupérer des données obsolètes.
Je me connecte donc à mon secondary et je switch sur ma démo et j'affiche ma collection :

```
rs0 [direct: secondary] demotest> show collections
personnes
```

```
rs0 [direct: secondary] demotest> db.personnes.find()
[
  { _id: ObjectId('693449f3bf8143239f9dc29d'), nom: 'Fatimazahra' },
  { _id: ObjectId('69344a17bf8143239f9dc29e'), nom: 'Malo' },
  { _id: ObjectId('69344a2fbf8143239f9dc29f'), nom: 'Amine' }
]
```

Quand on a inserer, on a utilisé le master mais on peut quand même les voir depuis le secondary ce qui veut dire que les données ont été bien répliquées car ce secondary appartient au même replica set que le primary. Je ne peux pas lire sur un secondary sauf si je passe le slaveok a true. Mais l'écriture sur un secondary reste impossible.

```
rs0 [direct: secondary] demotest> db.personnes.insert({"nom":"Arthur"})
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('69344f23aedd375e199dc29d') }
}
Write Errors: []
```

On simule une panne maintenant pour voir comment le système va réagir. J'arrête le primary :

```
PS C:\Users\Fei3> docker stop mongo3
mongo3
```

Quand j'essaye de me reconnecter sur mongo3, mon primary, j'obtiens l'erreur suivante :

```
PS C:\Users\Fei3> docker exec -it mongo3 mongosh --port 27017
Error response from daemon: container d1a7214a8ba09f15324e4c29b50a982d7a7dc219ffca1ed14e1c52fd1f1eb118 is not running
```

En me connectant a mongo2, j'observe que c'est celui qui a été élu primary :

```
PS C:\Users\Fei3> docker exec -it mongo2 mongosh --port 27017
Current Mongosh Log ID: 693450d2f68ac8f0c19dc29c
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+
.5.9
Using MongoDB:          8.2.2
Using Mongosh:          2.5.9

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-12-06T13:47:05.410+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine.
ee http://dochub.mongodb.org/core/prodnotes-filesystem
   2025-12-06T13:47:05.846+00:00: Access control is not enabled for the database. Read and write access to data and con
iguration is unrestricted
   2025-12-06T13:47:05.846+00:00: For customers running the current memory allocator, we suggest changing the contents
f the following sysfsFile
   2025-12-06T13:47:05.846+00:00: For customers running the current memory allocator, we suggest changing the contents
f the following sysfsFile
   2025-12-06T13:47:05.846+00:00: We suggest setting the contents of sysfsFile to 0.
   2025-12-06T13:47:05.846+00:00: We suggest setting swappiness to 0 or 1, as swapping can cause performance problems.
------

rs0 [direct: primary] test>
```

```
rs0 [direct: primary] test> use demotest
switched to db demotest
rs0 [direct: primary] demotest> show collections
personnes
rs0 [direct: primary] demotest> db.personnes.find()
[
  { _id: ObjectId('693449f3bf8143239f9dc29d'), nom: 'Fatimazahra' },
  { _id: ObjectId('69344a17bf8143239f9dc29e'), nom: 'Malo' },
  { _id: ObjectId('69344a2fbf8143239f9dc29f'), nom: 'Amine' }
]
```

Partie 2:

Compréhension de base :

1/ Un replica set est une grappe de serveurs sur lesquels j'effectue la réplication des données. Elle est constituée d'un nœud primary et le reste sont des secondary. Le but, c'est d'assurer une tolérance aux pannes.

2/ Le primary permet de centraliser la lecture (par défaut) et l'écriture.

3/ Les secondary permettent la réplication des écritures reçue par le primary.

4/ MongoDB n'autorise pas l'écriture sur un secondary parce qu'il privilégie une cohérence forte qu'il assure en n'autorisant que le maître à écrire, le seul point d'entrée pour l'écriture pour éviter les conflits et incohérences.

5/ La cohérence forte en MongoDB est assurée par le fait de lire directement dans le primary qui est la source la plus fiable et mise à jour.

6/ Quand je définis mon read préférence, je suis en train de dire à MongoDB que si mon client veut lire une donnée, MongoDB doit aller la chercher la ou j' ai précisé dans mon read préférence.

7/ Dans le cas des applications fortement consultées, on peut permettre la lecture sur un secondary pour diminuer la charge sur le primary même si cela peut mener à la consultation de données en retard.

Commandes et configuration :
8/ rs.initiate()

9/ rs.add("mongo2:27017")

10/ rs.status()

11/ rs.status() et on regarde stateStr

12/ rs.stepDown()

13/ rs.addArb("arbiter:27017")
Un arbitre participe aux élections mais ne stocke pas de données.
Il est utilisé pour obtenir un nombre impair de votes, améliorer les élections, et éviter les situations où un Replica Set ne peut pas élire de PRIMARY.

14/ cfg = rs.conf()
cfg.members[1].secondaryDelaySecs = 30
rs.reconfig(cfg)

Résilience et tolérance aux pannes :
15/ Dans ce cas, le replica reste en mode lecture seul comme il ne peut pas élire un primary.

16/ Le nœud doit être dans la majorité. Les nœuds avec une priorité plus élevée sont favorisés. Le niveau d'avancement de la réplication, on favorise le nœud le moins en retard. Le nœud doit être joignable et en bonne santé.

17/ Une élection est le processus automatique par lequel les nœuds d'un Replica Set choisissent un nouveau PRIMARY lorsque celui-ci devient indisponible.
Les SECONDARY se concertent, vérifient leur priorité et leur état de réplication, et élisent le nœud le plus à jour pour garantir la continuité du service.

18/ L'auto-dégradation signifie qu'un primary se rétrograde lui-même en secondary.
Cela arrive lorsque le primary perd la majorité des nœuds. Pour éviter des écritures incohérentes, il abandonne immédiatement son rôle.

19/ Parce qu'un replica set a besoin d'une majorité absolue pour élire un PRIMARY.
Et donc, un nombre impaire assure que les votes ne seront pas égaux.

20/ Une partition réseau peut diviser les nœuds en groupes qui ne peuvent plus communiquer ce qui dégrade les performances.

Scénarios pratiques :
21/ Le secondary et l'arbitre forment une majorité et élisent le secondary comme nouveau primary.

22/ Cela signifie que ce secondary réplique les données avec 120 secondes de retard volontaire. Il permet de protéger contre les suppressions accidentelles.

23/ readConcern : "linearizable"
garantit une lecture strictement à jour, même lors d'une élection.

writeConcern : "majority"
garantit que les écritures sont validées par une majorité de nœuds avant confirmation.

24/ writeConcern: { w: 2 }

25/ Parce que la réplication MongoDB est asynchrone.

Le secondary était en retard par rapport au primary. Pour éviter cela, il peut lire directement depuis le primary.

26/ db.isMaster()

27/ rs.stepDown()

28/ rs.add("nouveauNoeud:27017")

29/ rs.remove("nomDuNoeud:27017")

30/ cfg = rs.conf()

cfg.members[1].hidden = true

cfg.members[1].priority = 0

rs.reconfig(cfg)

Pour créer un secondary dédié aux backups, aux analyses lourdes, aux reportings sans impacter les performances du cluster.

31/ cfg = rs.conf()

cfg.members[1].priority = 2

rs.reconfig(cfg)

32/ rs.status().members.map(m => ({

  name: m.name,

  optime: m.optimeDate

}))

33/ rs.freeze(n) empêche un secondary d’être élu primary pendant n secondes.

Utile quand ce nœud n’est pas à jour, on veut forcer un autre nœud à devenir primary, maintenance en cours.

34/ La configuration est stockée dans la base de données locale.
Il suffit de relancer les mongod avec la même option :

--replSet rs0

Aucune configuration n'est nécessaire.

35/ Via les logs : tail -f /var/log/mongodb/mongod.log

Via les commandes shell : rs.printReplicationInfo()

rs.printSlaveReplicationInfo()

Questions complémentaires :

37/ Un Arbitre est un membre spécial du Replica Set qui ne stocke aucune donnée et ne participe pas à la réplication.

Son seul rôle est de voter lors des élections, afin de garantir un nombre impair de votants et d'assurer l’élection d’un primary.

38/ En comparant le temps des derniers oplog appliqués par chaque nœud :

rs.status().members.map(m => ({ name: m.name, optime: m.optimeDate }))

La différence de temps entre le Primary et chaque Secondary représente la latence de réplication.

39/ rs.printSlaveReplicationInfo()

40/ Réplication synchrone

> L’écriture est considérée comme validée seulement après que tous les nœuds (ou un quorum) aient appliqué les modifications.

Réplication asynchrone

> Le Primary écrit localement puis envoie l’opération aux Secondaries sans attendre qu’ils l’appliquent.

Quel type utilise MongoDB ?

> MongoDB utilise une réplication asynchrone.

41/ Oui.
La configuration peut être modifiée dynamiquement.

42/ Il devient out-of-date et peut avoir un retard trop important pour rattraper le Primary à partir de l'oplog.

MongoDB peut alors exiger une resynchronisation complète, beaucoup plus coûteuse.

Ce retard peut entraîner :
• lectures obsolètes (readPreference = secondary)
• impossibilité pour ce nœud d'être élu Primary

43/ MongoDB n'autorise les écritures que sur le primary, ce qui élimine quasiment tous les conflits.

En cas de divergences après une partition réseau, MongoDB applique un rollback des opérations du nœud perdant, pour revenir à l'état du primary.

44/ Non.
MongoDB garantit qu'un seul Primary peut exister à la fois afin d'éviter des écritures concurrentes et des incohérences de données.

Le mécanisme de quorum et d'élection empêche explicitement l'existence de plusieurs Primaries.

45/ Parce que les secondary ne peuvent pas écrire.
Ils rejettent toute requête d'écriture.

Même en lecture préférée secondaire, les écritures doivent obligatoirement aller au primary.

Cela garantit la cohérence et empêche les conflits de données.

46/ Un réseau instable peut entraîner :

1. Impossibilité d'élire un Primary
2. Auto-dégradation du Primary, il devient Secondary
3. Retard de réplication élevé

Le cluster devient moins fiable, moins performant, et parfois read-only.