# Coordination in MongoDB

💡 In this assignment, you will learn about leader election, replication and consistency using MongoDB.

**Table of contents**

**Tips**

💡 Read through the assignment description first before attempting the experiments. Try and complete the scenarios in sequence, so that you can build upon your understanding as the scenarios progress.

⚠️ Read the setup document before attempting the assignment.

## Grading

The assignment comprises three scenarios:

Scenario 1: Leader election (15p)

Scenario 2: Read and write consistency (20p)

Scenario 3: Rollbacks when two link failures occur (15p)

## Get familiar with MongoDB and replication

Read the following material:

- Lecture material and slides from #4 Distributed Systems: Coordination
- Introduction to MongoDB: What is a document database and its key features
- Replication in MongoDB: This assignment uses a three-node replica set

## Learning outcomes

You will learn how:

- Leader elections work in MongoDB replica sets
- Read and write consistency works with network link failures in MongoDB replica sets

⚠️ Ensure that you source `exercise2_script.sh` before running each scenario.

## Scenario 1: Leader election

As described in the MongoDB documentation on replica set elections, leader elections are triggered due a variety of reasons. You will experiment with three separate cases. In each case, analyze the election behaviour and answer the questions

- **Case A**: initiating a replica set
- **Case B**: performing replica set maintenance using `rs.stepDown()`
- **Case C**: the secondary members losing connectivity to the primary

### Case A: Initiating a replica set

Perform the following steps:

```
# 1. Clean up any docker compose environment, and restart
docker compose down -v; docker compose up -d;

# 2. Start the monitoring script for mongo1 on one terminal
e2_rs_monitor mongo1

# 3. Start observing logs on another terminal
docker compose logs | grep -i election

# 4. Initiate the replica set on another terminal. Note the time when you execut
e this comment with date
date; e2_rs_initiate_default;

# 5. Wait and observe what happens in each terminal for at least 30 seconds, not
e down relevant observations from the monitoring script and Docker compose logs

# 6. Cleanup
docker compose down -v;
```

**Questions:**

1. Which node becomes the primary? How long did it take from the time you initiated the replica set (Step 4) to electing a primary? (1p)
2. Run the experiments multiple times, do you get similar results for which node becomes the primary and how long it takes? Explain the reasons for your observations. (1p)
3. Are any other states visible in the monitoring scripts? (1p)
4. Explain the logs you see related to elections. (1p)
5. From the logs, do you observe that any node attempts a dry run? (1p)

### Case B: Replica set maintenance

In a MongoDB replica set, the primary of the replica set can be instructed to become a secondary through the rs.stepDown() method. This may be done for maintenance reasons. Perform the following steps:

```
# 1. Clean up any docker compose environment, and restart
docker compose down -v; docker compose up -d;

# 2. Initiate the replica set on one terminal
```

```
e2_rs_initiate_default

# 3. Start observing logs on another terminal
docker compose logs | grep -i election

# 4. Start the monitoring script on mongo1 on one terminal. Wait until replica s
et is initiatized, i.e., one node becomes PRIMARY and other two are SECONDARIES
e2_rs_monitor mongo1

# 5. Perform stepdown. Note down the time using date command
date; docker exec -it mongo1 mongosh --eval 'rs.stepDown()'

# 6. Wait and observe for at least 120 seconds. Note down relevant observations
from the monitoring script and Docker compose logs.

# 7. Cleanup
docker compose down -v;
```

**Questions:**

1. After executing step 5, which node becomes the primary? Does this change during the observation time of 120 seconds? (1p)
2. How long does it take after the stepdown command is performed (in step 5) to choose a new primary? Report the time taken and explain reasons for whether this is different from case A. (1p)
3. Explain the logs you see related to elections. Does a dry run take place? Provide the most relevant logs. (1p)
4. Repeat the experiment, i.e., steps 1 - 6 a few times. Is it always the same node that becomes the primary. Explain why. (2p)

## Case C: Primary loses connectivity to secondary nodes

Perform the following steps:

```
# 1. Clean up any docker compose environment, and restart
docker compose down -v; docker compose up -d;

# 2. Start the monitoring script on mongo1 and mongo2 on two separate terminals
e2_rs_monitor mongo1
e2_rs_monitor mongo2

# 3. Initiate the replica set (on another terminal). Wait until system reaches a
stable state, i.e., you observe one node becomes a PRIMARY and the other two are
SECONDARIES
e2_rs_initiate_default

# 4. Start observing logs on a separate terminal
docker compose logs | grep -i election

# 5. Bring down the links between mongo1 and other nodes. Note down the time.
date; e2_link_block_m1_to_m2_and_m3

# 6. Wait and observe the logs for at least 120 seconds, note down relevant obse
```

```
rvations

# 7. Bring up the links between mongo1 and other nodes. Note down the time.
date; e2_link_restore_m1

# 8. Wait and observe as needed for at least 120 seconds, note down relevant obs
ervations

# 9. Cleanup
docker compose down -v;
```

**Questions:**

1. In step 6, after disconnecting the primary from other nodes:
   - Which node becomes primary? How long did it take for a new primary to be selected after step 5 was executed? (1p)
   - Which replica set settings affect this behaviour? (Hint: check `e2_rs_initiate_default` in `exercise2_script.sh`) (2p)
2. In step 8, after restoring the link:
   - Which node becomes the primary? How long did it take after restoring connectivity to mongo1 in step 7? (1p)
   - From the logs, describe at what intervals does mongo1 attempt elections. (1p)

## Scenario 2: Read and write consistency

Consistency in distributed databases like MongoDB is crucial for ensuring data reliability and durability across replicas. It determines how data is propagated to secondary nodes, and what consistency guarantees are provided for write and read operations.

MongoDB uses properties called write concerns and read concerns. These properties are set on the MongoDB clients that write and read data from MongoDB. Through the effective use of write concerns and read concerns, it is possible to adjust the level of consistency and availability guarantees as appropriate for an application or use case.

The experiments in this scenario are divided into two cases:

- **Case A:** All nodes are up and working normally
- **Case B:** Link between the primary and one of the secondaries is broken

⚠️ You will write code to read and write data to the replica set. Use Python and PyMongo to complete this assignment. You can use the code provided for scenario 3 (`scenario3.py`) to check how to read and write documents to the replica set.
More information from the official documentation are available here and here

⚠️ Ensure that you source `exercise2_script.sh` before running the scenario.

### Case A: All nodes are up and working normally

The objective of this experiment is to analyze time taken for inserting a document to the database with different write concern values. Write Python code to use a replica set client to perform writes. For each write concern (0, 1, 2, 3, majority), your code should write 100 documents to the database one at a time, and calculate the time taken for each write. Your task is to analyze the differences in time taken for different write concerns.

The steps to be carried out are as follows:

```
# 1. Write code for this scenario

# 2. Clean up any docker compose environment, and restart
docker compose down -v; docker compose up -d;

# 3. Start the monitoring script on mongo1
e2_rs_monitor mongo1

# 4. Initiate the replica set (on another terminal)
e2_rs_initiate_default

# 5. Wait until system reaches the stable state, i.e., one node is the PRIMARY a
nd the other two are SECONDARIES

# 6. Run your code.

# 7. Cleanup
docker compose down -v
```

**Questions:**

1. Plot the time taken for each write operation with different write concerns. Visualize the values using an empirical cumulative distribution function. (5p)
2. Explain the reasons for the differences in time taken (if any) between different write concerns. (3p)
3. What is journaling? Why can't it be enabled with write concern 0? (1p)

### Case B: Link between the primary and one of the secondary nodes is broken

The objective of this experiment is to analyze write concerns and how the data is read from the replica set when one link is broken.

Write Python code to **write one document** (not 100 like in the previous case) with different write concerns (i.e., 0, 1, 2, 3, majority). For each write, read the data from each node with a direct connection and with different read concerns (i.e., local, majority, linearizable) to validate what is written. Also keep track of the time taken for each read.

Writes should be performed using a replica set client.

Reads should be performed using three independent clients, each which make a direct connection to mongo1, mongo2 and mongo3.

This is explained in the following pseudo code.

```
write_concerns = [0, 1, 2, 3, "majority"]
read_concerns = ["local", "majority", "linearizable"]
for wc in write_concerns:
    # TODO: Write one document to the collection
    sleep(0.2) # Ensure that sufficient time passes before reading
    for reader in readers:
        for rc in read_concerns:
            # TODO: Read the data. Measure time taken for each read.
```

💡 You can check the code provided for scenario 3 (`scenario3.py`) to see how data is written and read with different clients, and how to track the time taken for reads and writes.

Once you have written your code, the experiment steps are as follows:

```
# 1. Write code for this scenario

# 2. Clean up any docker compose environment, and restart
docker compose down -v; docker compose up -d;

# 3. Initiate the replica set
e2_rs_initiate_default

# 4. Wait until system reaches the stable state where one node becomes PRIMARY a
nd the other two SECONDARIES
e2_rs_monitor

# 5. Break link between the PRIMARY (mongo1) and one SECONDARY (mongo2)
e2_link_block_m1_to_m2

# 6. Wait until system reaches the new stable state, i.e., you see mongo2 is not
reachable from mongo1
e2_rs_monitor

# 7. Run your code to perform the reads and writes

# 8. Clean up
docker compose down -v
```

**Questions:**

1. Do any write errors occur for the different write concerns? Explain why. (2p)
2. If there is an error, is the data still readable from the database? Explain why. (3p)
3. Do reads with different read concerns take different time? Explain why. (3p)
4. Does the data get replicated to mongo 2 when the link is restored? Explain why or why not, and how this process works. (3p)

## Scenario 3: Rollback

In the scenario, we create a situation where a rollback occurs.

You are provided the necessary Python code to execute this scenario and do not need to make code changes in `scenario3.py`

⚠️ Ensure that you source `exercise2_script.sh` before running the scenario.

Perform the following steps:

```
# 1. Clean up any docker compose environment, and restart
docker compose down -v; docker compose up -d;

# 2. Initiate replica set with long election and heartbeat timers
e2_rs_initiate_slow_election_detection

# 3. Monitor the status on another terminal until mongo1 is PRIMARY, and other n
odes are in SECONDARY state
# This will take longer than in previous tests as we updated election and heartb
eat timers
e2_rs_monitor

# 4. Break links between mongo1<->mongo2 and mongo1<->mongo3 and run the next st
ep (step 5) IMMEDIATELY
e2_link_block_m1_to_m2_and_m3

# 5. Write using the replica set client before the system detects anything has g
one wrong
python scenario3.py --write

# 6. Read data from all nodes using direct connections to each client.
# Check which nodes have the data and which don't and what value is written
python scenario3.py --read

# 7. Wait for mongo2 or mongo3 to become PRIMARY, you can check that by using th
e command below.
e2_rs_monitor mongo2

# 8. After mongo2 or mongo3 have become PRIMARY, perform another write
python scenario3.py --write

# 9. Read data from all nodes
# Check which nodes have the data and which don't and what value is written
python scenario3.py --read

# 10. Restore the links on mongo1 and wait a few seconds (about 10 seconds)
e2_link_restore_m1

# 11. Read data from all nodes
# Check which nodes have the data and which don't and what value is written
python scenario3.py --read
```

**Questions:**

1. Describe your observations of the data read from each node in steps 6, 9, and 11. Specifically, what data is visible on each node. Explain why. (5p)
2. Describe the sequence of state changes and what leads to the rollback. (5p)
3. Where is the rollback data stored? Provide logs or screenshots. (Hint: Check contents of `/data/db` directory on mongo1) (2p)
4. How can rollbacks be avoided, and are those methods applicable in this scenario? Explain why. (3p)

---

Last update: @February 2, 2026