

Distributed Systems Lab Report – MongoDB Replica Set & Consistency Experiments

Introduction

This lab explores distributed database concepts through a MongoDB replica set setup.

The primary objectives are:

- To configure and analyze a replicated MongoDB cluster.
- To examine the impact of replication and failover on data availability and consistency.
- To experimentally demonstrate strong vs. eventual consistency models.
- To conceptually understand distributed transactions through the ACID and Saga frameworks.

Part A: Setup & Baseline

1. Database Cluster Setup

A docker file – docker-compose.yml was created to define the 3-node MongoDB cluster. All nodes belong to the replica set rs0. The ports mapped were: 27017, 27018, 27019 to 27017 (internal). The shared network of mongo-net was defined to allow inter-node communication.

Docker Compose Configuration

```
version: "3.8"

services:
  mongol:
    image: mongo:6
    container_name: mongol
    ports:
      - 27017:27017
    command: ["mongod", "--replSet", "rs0", "--bind_ip_all"]
    networks:
      - mongo-net

  mongo2:
    image: mongo:6
    container_name: mongo2
    ports:
      - 27018:27017
    command: ["mongod", "--replSet", "rs0", "--bind_ip_all"]
```

```

networks:
  - mongo-net

mongo3:
  image: mongo:6
  container_name: mongo3
  ports:
    - 27019:27017
  command: ["mongod", "--replSet", "rs0", "--bind_ip_all"]
  networks:
    - mongo-net

networks:
  mongo-net:
    driver: bridge

```

To start the cluster, run `docker compose up -d` in the terminal
Then check all Docker containers are running by using `docker ps` command in the terminal

```

(base) pejatan@PejateMacBook-Air distributed-systems-a2 % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
5f6f786468d2   mongo:6   "docker-entrypoint.s..." About a minute ago Up About a minute   0.0.0.0:27019->27017/tcp, [::]:27019->27017/tcp   mongo3
63f78b187976   mongo:6   "docker-entrypoint.s..." About a minute ago Up About a minute   0.0.0.0:27018->27017/tcp, [::]:27018->27017/tcp   mongo2
69576a38bad9   mongo:6   "docker-entrypoint.s..." About a minute ago Up About a minute   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp   mongo1

```

distributed-systems-a2
 /Users/pejatan/ds/distributed-systems-a2

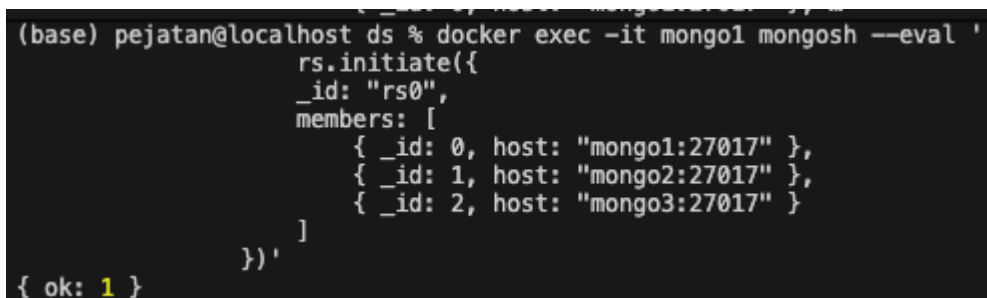
View configurations

<div> mongo2 </div> <div> mongo:6 27018:27017 </div> <div> </div>	<pre>{ "target": "mongo1:27017", "maxHeartbeatRetries": 2, "error": { "code": 2, "codeName": "BadValue", "errmsg": "Received heartbeat from member with the same member ID as ourself: 1" } }</pre>
<div> mongo1 </div> <div> mongo:6 27017:27017 </div> <div> </div>	<pre>{ "t": { "\$date": "2025-10-22T22:03:43.090+00:00" }, "s": "I", "c": "REPL_HB", "id": 23974, "ctx": "ReplCoord-1", "msg": "Heartbeat failed after max retries", "attr": { "target": "mongo1:27017", "maxHeartbeatRetries": 2, "error": { "code": 2, "codeName": "BadValue", "errmsg": "Received heartbeat from member with the same member ID as ourself: 2" } } }</pre>
<div> mongo3 </div> <div> mongo:6 27019:27017 </div> <div> </div>	<pre>{ "t": { "\$date": "2025-10-22T22:03:43.589+00:00" }, "s": "I", "c": "REPL_HB", "id": 23974, "ctx": "ReplCoord-3", "msg": "Heartbeat failed after max retries", "attr": { "target": "mongo1:27017", "maxHeartbeatRetries": 2, "error": { "code": 2, "codeName": "BadValue", "errmsg": "Received heartbeat from member with the same member ID as ourself: 1" } } }</pre>
<div> mongo3 </div>	<pre>{ "t": { "\$date": "2025-10-22T22:03:45.096+00:00" }, "s": "I", "c": "REPL_HB", "id": 23974, "ctx": "ReplCoord-1", "msg": "Heartbeat failed after max retries", "attr": { "target": "mongo1:27017", "maxHeartbeatRetries": 2, "error": { "code": 2, "codeName": "BadValue", "errmsg": "Received heartbeat from member with the same member ID as ourself: 2" } } }</pre>

After the containers are up:

Replica Set Initialization

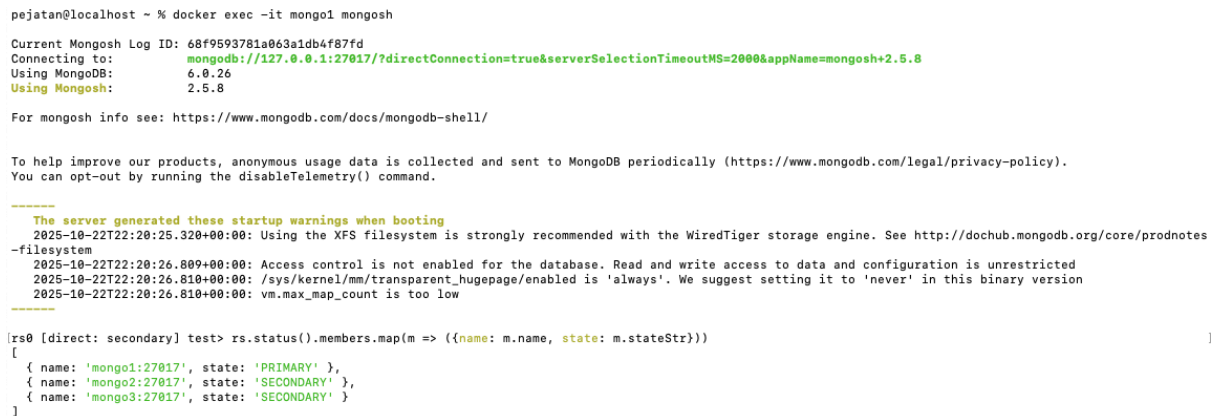
```
terminal> docker exec -it mongo1 mongosh --eval '
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo1:27017" },
    { _id: 1, host: "mongo2:27017" },
    { _id: 2, host: "mongo3:27017" }
  ]
})'
```



```
(base) pejatan@localhost ds % docker exec -it mongo1 mongosh --eval '
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo1:27017" },
    { _id: 1, host: "mongo2:27017" },
    { _id: 2, host: "mongo3:27017" }
  ]
})'
{ ok: 1 }
```

After running the code above, The replica set is then initialized, with **mongo1** elected as the **primary node**, while **mongo2** and **mongo3** act as **secondary nodes**.

Data replication is automatically synchronized across all members of the set.



```
pejatan@localhost ~ % docker exec -it mongo1 mongosh
Current Mongosh Log ID: 68f9593781a063a1db4f87fd
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Using MongoDB:      6.0.26
Using Mongosh:      2.5.8

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

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

-----
The server generated these startup warnings when booting
2025-10-22T22:20:25.320+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes
-filesystem
2025-10-22T22:20:26.809+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2025-10-22T22:20:26.810+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never' in this binary version
2025-10-22T22:20:26.810+00:00: vm.max_map_count is too low
-----

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

The `rs.status().members.map(m => ({name: m.name, state: m.stateStr}))` command shows the replica status and confirms that there is 1 primary node (mongo1) and 2 secondary nodes (mongo2 and mongo3). All nodes are healthy and synchronised.

2. Simple Data Mode

The first two steps of the `replication_experiment.py` script correspond to this stage. These steps were designed to initialise the MongoDB replica set connection and

insert a set of baseline user data before any replication or failover tests were performed. The script contains detailed comments explaining each operation. When executed, it confirms that the data has been successfully inserted into the cluster.

```
terminal > docker run -it --rm \
-v $(pwd):/app \
-w /app \
--network distributed-systems-a2_mongo-net \
python:3.11 \
bash -c "pip install pymongo && python setup_baseline.py"
```

Run the testing py file on Docker called setup_baseline.py.

Sample data inserted in these initial steps are shown below:

```
[Step 1] Connecting to MongoDB replica set: mongod://mongo1:27017,mongo2:27017,mongo3:27017/?replicaSet=rs0
Replica Set connection established.
MongoDB Replica Set is ready and accessible.

[Step 2] Inserting baseline documents BEFORE failover...
Documents in userProfile before insert: 0
Collection is currently empty.
Inserted u001
Inserted u002
Inserted u003
Inserted u004
Inserted u005

All documents in userProfile after baseline insertion:
[{'_id': 'u001',
  'user_id': 'u001',
  'username': 'Tan_1',
  'email': 'Tan_1@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761172858.5304768},
 {'_id': 'u002',
  'user_id': 'u002',
  'username': 'Tan_2',
  'email': 'Tan_2@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761172859.0483346},
 {'_id': 'u003',
  'user_id': 'u003',
  'username': 'Tan_3',
  'email': 'Tan_3@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761172859.5567427},
 {'_id': 'u004',
  'user_id': 'u004',
  'username': 'Tan_4',
  'email': 'Tan_4@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761172860.062927},
 {'_id': 'u005',
  'user_id': 'u005',
  'username': 'Tan_5',
  'email': 'Tan_5@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761172860.5685704}]
```

Part B – Replication and Failover Experiments

1. Replication Factor / Write Concern:

Code in the file called replication_experiment.py

Testing Result:

```
[Step 3] Comparing write latency for w=1, w='majority', and w=3...
WriteConcern = 1      → avg latency = 0.0009s, std dev = 0.0005s
WriteConcern = majority → avg latency = 0.0026s, std dev = 0.0007s
WriteConcern = 3      → avg latency = 0.0029s, std dev = 0.0004s

Latency comparison summary (average of 50 runs):
w=1          → 0.0009s (±0.0005s)
w=majority   → 0.0026s (±0.0007s)
w=3          → 0.0029s (±0.0004s)
```

Results Observation

The latency results clearly show that the average write time increases slightly as the write concern level strengthens.

In the three-node MongoDB replica set, with $w=1$, the operation is acknowledged by the primary node only and achieves the lowest latency (approximately 0.9 ms).

For $w='majority'$, the write must be confirmed by at least two nodes, the primary and one secondary, resulting in a moderate delay (around 2.6 ms).

Finally, $w=3$ requires confirmation from all three replicas, producing the highest latency (about 2.9 ms).

These results confirm that stronger write concerns introduce minor replication delays as additional acknowledgements are required before a write is considered successful.

2. Leader-Follower (Primary-Backup) Model

Testing procedure and result:

Before failover:

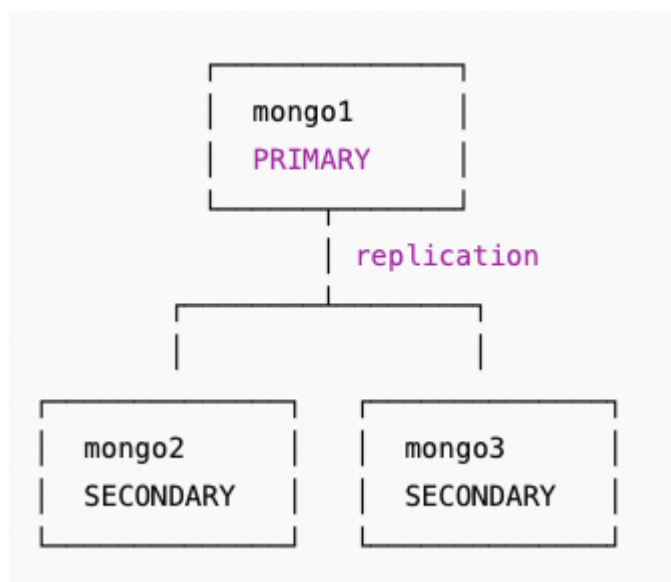
mongo1 acted as the Primary, accepting all write operations (baseline data: u001–u005).

```
[Step 2] Checking for existing documents and inserting baseline data if needed...
Documents currently in 'userProfile': 5
Existing data found – baseline insertion skipped.
Existing documents (sample):
[{'user_id': 'u001',
  'username': 'Tan_1',
  'email': 'Tan_1@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177844.0970042},
{'user_id': 'u002',
  'username': 'Tan_2',
  'email': 'Tan_2@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177844.61839},
{'user_id': 'u003',
  'username': 'Tan_3',
  'email': 'Tan_3@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177845.123994},
{'user_id': 'u004',
  'username': 'Tan_4',
  'email': 'Tan_4@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177845.630352},
{'user_id': 'u005',
  'username': 'Tan_5',
  'email': 'Tan_5@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177846.137749}]
```

During failover:

The primary node (mongo1) was intentionally stopped using `docker stop mongo1`.

The replica set automatically elected a new Primary, which in this case was mongo2.



When mongo1 stops, election promotes mongo2 → new PRIMARY.

[Step 4] Simulating PRIMARY node failure programmatically...
Please run the following command in another terminal to stop the primary node:

```
docker stop mongo1
```

Press Enter once you have stopped mongo1...
Waiting for automatic re-election...

[Step 5] Waiting for new PRIMARY election...
- mongo1:27017 → (not reachable/healthy)
- mongo2:27017 → PRIMARY
- mongo3:27017 → SECONDARY

New PRIMARY elected: mongo2:27017

After failover:

New data (u006) was successfully written to the new primary (mongo2).

[Step 6] Inserting one new document after failover (w='majority')...
Inserted new document: u006

All documents in userProfile after post-failover insertion:

```
[{'user_id': 'u001',  
  'username': 'Tan_1',  
  'email': 'Tan_1@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177844.0970042},  
{ 'user_id': 'u002',  
  'username': 'Tan_2',  
  'email': 'Tan_2@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177844.61839},  
{ 'user_id': 'u003',  
  'username': 'Tan_3',  
  'email': 'Tan_3@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177845.123994},  
{ 'user_id': 'u004',  
  'username': 'Tan_4',  
  'email': 'Tan_4@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177845.630352},  
{ 'user_id': 'u005',  
  'username': 'Tan_5',  
  'email': 'Tan_5@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177846.137749},  
{ 'user_id': 'u006',  
  'username': 'user_6',  
  'email': 'user_6@example.com',  
  'inserted_after_failover': True,  
  'last_login_time': 1761177933.2412446}]
```

[Step 7] Restoring the old PRIMARY node (mongo1)...
Please run the following command in another terminal to start the stopped container:

```
docker start mongo1
```

Press Enter once you have started mongo1...
Waiting 8 seconds for the node to fully recover and sync with the replica set...

[Step 8] Verifying data consistency across replica set...

Current primary node: mongo2:27017

Documents from PRIMARY (mongo2:27017):

```
[{'user_id': 'u001',  
  'username': 'Tan_1',  
  'email': 'Tan_1@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177844.0970042},  
{ 'user_id': 'u002',  
  'username': 'Tan_2',  
  'email': 'Tan_2@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177844.61839},  
{ 'user_id': 'u003',  
  'username': 'Tan_3',  
  'email': 'Tan_3@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177845.123994},  
{ 'user_id': 'u004',  
  'username': 'Tan_4',  
  'email': 'Tan_4@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177845.630352},  
{ 'user_id': 'u005',  
  'username': 'Tan_5',  
  'email': 'Tan_5@example.com',  
  'inserted_before_failover': True,  
  'last_login_time': 1761177846.137749},  
{ 'user_id': 'u006',  
  'username': 'user_6',  
  'email': 'user_6@example.com',  
  'inserted_after_failover': True,  
  'last_login_time': 1761177933.2412446}]
```

Once the old primary (mongo1) was restarted using `docker start mongo1`, it automatically re-synchronized and joined the replica set as a Secondary.

```
Documents from SECONDARY (sample read):
[{'user_id': 'u001',
  'username': 'Tan_1',
  'email': 'Tan_1@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177844.0970042},
 {'user_id': 'u002',
  'username': 'Tan_2',
  'email': 'Tan_2@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177844.61839},
 {'user_id': 'u003',
  'username': 'Tan_3',
  'email': 'Tan_3@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177845.123994},
 {'user_id': 'u004',
  'username': 'Tan_4',
  'email': 'Tan_4@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177845.630352},
 {'user_id': 'u005',
  'username': 'Tan_5',
  'email': 'Tan_5@example.com',
  'inserted_before_failover': True,
  'last_login_time': 1761177846.137749},
 {'user_id': 'u006',
  'username': 'user_6',
  'email': 'user_6@example.com',
  'inserted_after_failover': True,
  'last_login_time': 1761177933.2412446}]
```

```
Replication experiment completed successfully!
All nodes should now be in sync with identical data.
```


Interpretation:

This behaviour clearly demonstrates MongoDB's **Leader–Follower (Primary–Backup)** replication model:

- The **leader** (primary) handles all write operations.
- The **followers** (secondaries) replicate data from the leader.
- When the leader fails, **automatic re-election** ensures that another node takes over, maintaining write availability.

Conclusion

MongoDB's replication mechanism is a practical implementation of the Leader–Follower model, ensuring:

- **Strong consistency** – writes always occur through a single primary node.
- **High availability** – automatic failover allows continuous operation.
- **Durability** – followers serve as synchronized backups of the primary's data.

3. Leaderless (Multi-Primary) Model Discussion (Not Applicable to MongoDB)

MongoDB's replica set architecture follows a **single-primary, multiple-secondary** design, meaning that all write operations are directed to a single elected primary node. This architecture enforces **strong consistency** under normal operation but does not support **leaderless** or **multi-primary** writes.

In contrast, **leaderless databases** such as *Apache Cassandra* allow writes to occur on any node. Consistency is achieved through **quorum-based read/write acknowledgements** (e.g., $R + W > N$), leading to **eventual consistency** rather than strict consistency.

Although this experiment was not performed due to MongoDB's architectural constraints, it is valuable to discuss the conceptual trade-offs between both designs:

Property	Leaderless / Multi-Primary (e.g., Cassandra)	Single-Primary (e.g., MongoDB)
Write Location	Any replica node	Only primary node
Consistency Model	Eventual consistency (tunable via quorum)	Strong consistency (primary-led)
Conflict Resolution	Client-side reconciliation or "last-write-wins"	No conflicts (single write source)
Fault Tolerance	High (no single point of failure)	Dependent on primary election

Latency	Low writes (local node)	Slightly higher (must reach primary)
---------	-------------------------	--------------------------------------

Conclusion

MongoDB prioritizes **data integrity** and **deterministic ordering** through its single-primary replication model, whereas leaderless systems trade strict consistency for **higher availability** and **fault tolerance**.

This highlights that the achievable balance on the **CAP spectrum** is defined by **architectural design choices**, rather than the replication mechanism alone.

Part C – Consistency Models

1. Strong Consistency

Code in the file called strong_consistency_experiment.py

First restart the docker container again

```
terminal > docker-compose down -v
```

```
terminal > docker-compose up -d
```

```
terminal > docker exec -it mongo1 mongosh --eval '
```

```
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo1:27017" },
    { _id: 1, host: "mongo2:27017" },
    { _id: 2, host: "mongo3:27017" }
  ]
})'
```

```
terminal > docker run -it --rm \
```

```
-v $(pwd):/app \
```

```
-w /app \
```

```
--network distributed-systems-a2_mongo-net \
```

```
python:3.11 \
```

```
bash -c "pip install pymongo && python strong_consistency_experiment.py"
```

Run the testing py file on Docker called strong_consistency_experiment.py.

Step 1 and Step 2

```
[Step 1] Connecting to MongoDB replica set: mongodb://mongo1:27017,mongo2:27017,mongo3:27017/?replicaSet=rs0
Connected to MongoDB replica set.

[Step 2] Testing Strong Consistency (w='majority', readConcern='majority')
→ Inserting new user document 'uid_1761241284' using w='majority'...
Write acknowledged by majority of nodes (strong consistency).

→ Immediately reading from a secondary node (expect same value).
Secondary read result:
{'_id': 'uid_1761241284',
'email': 'tan@example.com',
'inserted_for_strong_test': True,
'ts': 1761241284.1218247,
'user_id': 'uid_1761241284',
'username': 'tan'}

Observation:
• The read reflects the latest written value.
• Writes take slightly longer due to majority confirmation.
• Confirms strong consistency – Consistency prioritized over Availability (C > A).
```

Description:

The experiment began by connecting to a three-node MongoDB replica set (mongo1, mongo2, and mongo3) using the URI

`mongodb://mongo1:27017,mongo2:27017,mongo3:27017/?replicaSet=rs0`.

A successful connection confirmed that the replica set was properly initialized and that the client could identify the current primary node.

A new user document (e.g., `uid_1761241284`) was then inserted using `WriteConcern("majority")`, ensuring that the write was acknowledged by at least two out of three nodes before being confirmed.

Immediately after insertion, a read operation was executed on a secondary node with `ReadConcern("majority")`, which returns only data committed by the majority of nodes.

This setup enforces synchronous replication and guarantees that reads reflect the latest consistent state of the cluster.

Observation:

The secondary node immediately returned the newly inserted document, demonstrating that the most recent write had already been replicated and confirmed by multiple nodes.

This verifies that MongoDB's replica set enforces strong consistency, where the system always returns the most up-to-date committed data.

Although write latency increases slightly due to multi-node acknowledgment, the data is guaranteed to be consistent across replicas.

This behaviour reflects the Consistency > Availability (C > A) trade-off described in the CAP theorem.

Step 3 ~ 5

```
[Step 3] Simulating Primary node failure (manual intervention required)
Please run the following command in another terminal:

    docker stop mongo1

Then press Enter to continue once the primary has been stopped.

" Waiting for user confirmation... (press Enter when ready)

Waiting 8 seconds for new Primary election...

[Step 4] Attempting write operation during/after failover...
Insert a new data...
Write succeeded – new primary elected, consistency preserved.

[Step 5] Recovery and Verification
Please restart the old primary in another terminal using:

    docker start mongo1

" Press Enter once mongo1 has been restarted...

Waiting for node to rejoin and synchronize...

Reading back latest documents from primary:
[{'_id': 'uid_1761241422', 'ts': 1761241421.31609, 'username': 'tan_failover'},
 {'_id': 'uid_1761241284', 'ts': 1761241284.1218247, 'username': 'tan'}]

Strong Consistency Experiment completed successfully.
System maintained consistency through failover (C > A under CAP theorem).
```

Description

To examine MongoDB's behaviour during a failure, the current primary (mongo1) was manually stopped using `docker stop mongo1`.

The cluster automatically initiated a leader re-election, selecting a new primary (e.g., mongo2) to maintain system continuity.

During this process, write operations were temporarily unavailable until a new primary was established, ensuring that no inconsistent writes occurred.

After the election, a new document (e.g., uid_1761241422) was inserted using the same majority write concern, testing the system's ability to maintain strong consistency after recovery.

Finally, the old primary (mongo1) was restarted with `docker start mongo1` and automatically rejoined as a secondary node, syncing its data from the new primary.

Observation

The write operation after failover succeeded only once a new primary was elected, confirming that MongoDB temporarily sacrifices availability to preserve a consistent system state.

When the stopped node was restarted, it seamlessly synchronized with the current primary and contained both pre- and post-failover documents (uid_1761241284 and uid_1761241422).

This verified that MongoDB's replication mechanism maintains durability and data integrity even under node failures.

Overall, the experiment demonstrated that MongoDB enforces strong consistency through leader election, synchronous replication, and automatic recovery—aligning with the $C > A$ behaviour predicted by the CAP theorem.

Conclusion

In summary, the strong consistency experiment verified that MongoDB's replica-set architecture ensures immediate data consistency across nodes when configured with `WriteConcern("majority")` and `ReadConcern("majority")`.

All writes were synchronously replicated and acknowledged by multiple nodes before confirmation, and any read from a secondary node reflected the most recent committed state.

During a primary node failure, the system temporarily suspended write operations until a new leader was elected, demonstrating that MongoDB prioritizes Consistency over Availability ($C > A$) as defined by the CAP theorem.

Once the failed node rejoined the cluster, all data was automatically synchronized, confirming the durability and reliability of MongoDB's replication protocol under strong consistency guarantees.

2. Eventual Consistency:

Code in the file called `eventual_consistency_experiment.py`

```
terminal > docker-compose down -v
```

```
terminal > docker-compose up -d
```

```
terminal > docker exec -it mongo1 mongosh --eval '
```

```
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo1:27017" },
    { _id: 1, host: "mongo2:27017" },
    { _id: 2, host: "mongo3:27017" }
  ]
})'
```

```
terminal > docker run -it --rm \
```

```
-v $(pwd):/app \
```

```
-w /app \
```

```
--network distributed-systems-a2_mongo-net \
```

```
python:3.11 \
```

```
bash -c "pip install pymongo && python eventual_consistency_experiment.py"
```

```
[Step 1] Connecting to MongoDB replica set: mongod://mongo1:27017,mongo2:27017,mongo3:27017/?replicaSet=rs0
Connected to MongoDB replica set.

[Step 2] Testing Eventual Consistency (w=1, ReadConcern='local')
→ Writing document 'uid_1761250312' with w=1 (fast, non-majority)...
Write acknowledged by primary only (not guaranteed replicated yet).

→ Immediately reading from a secondary node (may see old data).
Secondary read result (initial):
None

Polling every 1s until secondary catches up...
Read 0: None
Read 1: {'_id': 'uid_1761250312', 'user_id': 'uid_1761250312', 'username': 'rose_eventual', 'email': 'rose_eventual@example.com', 'inserted_for_eventual_test': True, 'ts': 1761250312.1261337}
Secondary caught up – eventual consistency achieved.

Observation:
• The first read from the secondary may return None or old data.
• After several seconds, the document becomes visible once replication completes.
• Demonstrates Eventual Consistency, where all nodes converge over time.
• Reflects Availability prioritized over Consistency ( $A > C$ ) per CAP theorem.
• Such design is acceptable for use cases like social media likes, view counts, or IoT sensor updates.

Eventual Consistency Experiment completed successfully.
```

Description

In this experiment, MongoDB was configured for eventual consistency by setting WriteConcern(1) and ReadConcern("local").

Under this configuration, write operations are acknowledged only by the primary node and are not required to be replicated before confirmation.

A new document (e.g., uid_1761250312) was inserted into the collection with this weak write concern to achieve fast acknowledgement.

Immediately afterward, a read operation was performed on a secondary node to verify whether the data had already propagated across the replica set.

Finally, a polling loop was implemented to repeatedly read the same document from the secondary node every second until it appeared, illustrating how the system converges to a consistent state over time.

Observation and Conclusion

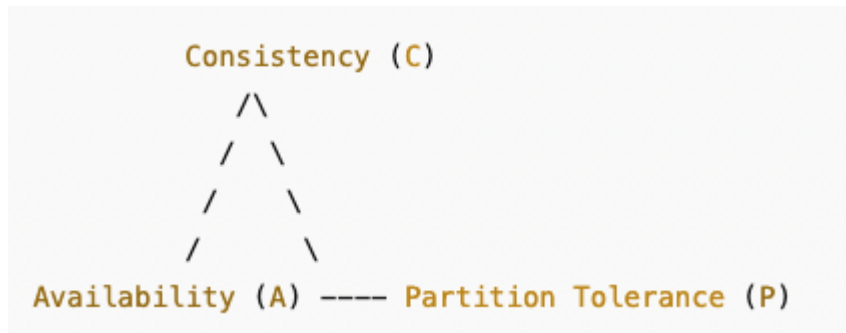
The first read from the secondary returned None, indicating that the recent write had not yet been replicated, an expected temporary inconsistency under eventual consistency.

After approximately one second, the secondary returned the new document, confirming that replication had completed and that all nodes had converged to the same state.

This behaviour clearly demonstrates eventual consistency, where data written to one node becomes visible to others after a short delay.

In this configuration, MongoDB prioritizes Availability over Consistency ($A > C$) according to the CAP theorem, writes are accepted immediately, ensuring system responsiveness, even though replicas may briefly hold stale data.

Such a design is beneficial for high-availability or latency-sensitive applications such as social media likes, sensor readings, and real-time analytics, where immediate synchronization is not critical but continuous uptime is essential.



MongoDB Strong Consistency → near “C”

MongoDB Eventual Consistency → between “A” and “P”

CAP theorem trade-offs demonstrated in MongoDB experiments. Strong consistency prioritizes $C > A$, while eventual consistency prioritizes $A > C$.

3. Causal Consistency

Code in the file called consistency_causal.py

```
terminal > docker-compose down -v
```

```
terminal > docker-compose up -d
```

```
terminal > docker exec -it mongo1 mongosh --eval '
```

```
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo1:27017" },
    { _id: 1, host: "mongo2:27017" },
    { _id: 2, host: "mongo3:27017" }
  ]
})'
```

```
terminal > docker run -it --rm \
```

```
-v $(pwd):/app \
```

```
-w /app \
```

```
--network distributed-systems-a2_mongo-net \
```

```
python:3.11 \
```

```
bash -c "pip install pymongo && python consistency_causal.py"
```

```
[Step 1] Connecting to MongoDB replica set: mongod --port 27017,mongo2:27017,mongo3:27017/?replicaSet=rs0
Connected to MongoDB replica set successfully.

[Step 2] Demonstrating causal consistency using session...
Client A wrote 'Hello world'.
Client B wrote 'Nice post!' in response.

[Step 3] Reading documents within the same session (causal order preserved):
[{'_id': 'msg1',
  'author': 'ClientA',
  'text': 'Hello world',
  'ts': 1761254150.1349578},
 {'_id': 'msg2',
  'author': 'ClientB',
  'reply_to': 'msg1',
  'text': 'Nice post!',
  'ts': 1761254150.155539}]

Observation:
• The dependent document ('Nice post!') appears only after the original ('Hello world').
• MongoDB ensures this ordering within the same causally consistent session.
• Causal consistency guarantees that related operations are observed in order,
  even across distributed replicas, without requiring full global synchronization.
• This model provides a balance between consistency and availability in distributed systems.

Causal Consistency Experiment completed successfully.
```

Description

This experiment demonstrates MongoDB's causal consistency using session-based operations.

Within a causally consistent session, Client A first inserted a document ("Hello world"), followed by Client B who read that message and replied ("Nice post!").

The session ensured that the reply could only be observed after the original message had been successfully written and acknowledged.

This behavior reflects MongoDB's ability to maintain the logical order of causally related operations across distributed replicas without enforcing a strict global ordering.

Observation

The results confirmed that the dependent document ("Nice post!") appeared only after the original ("Hello world") within the same session.

MongoDB preserved this causal order even though the operations could have occurred on different nodes.

This demonstrates that causal consistency guarantees ordered visibility for related events, while still allowing independent operations to proceed asynchronously. Such a model provides a balanced compromise between consistency and availability, making it suitable for collaborative systems, message threads, and event-based data flows where the logical sequence of interactions matters more than real-time synchronization.

Part D: Distributed Transactions

Description

Distributed transactions involve coordinating multiple independent services to maintain data consistency across a system that lacks a single shared database. In a microservices or distributed database environment, ensuring *ACID* (Atomicity, Consistency, Isolation, Durability) properties across all nodes becomes extremely complex and often impractical due to network latency, failures, and partial connectivity.

To illustrate, consider a simplified **e-commerce workflow** that involves three independent services:

- **OrderService** – creates a new order record.
- **PaymentService** – charges the customer's payment method.
- **InventoryService** – reserves or deducts items from stock.

All three operations must succeed or fail together for the system to remain consistent.

ACID Transactions in a Distributed Setting

Under a strict **ACID transaction**, all three services would need to participate in a global transaction (often implemented via a *two-phase commit protocol*).

Mechanism:

1. **Prepare phase:** Each service confirms it can commit (e.g., sufficient balance, stock available).
Commit phase: The coordinator instructs all participants to finalize the operation.

Challenges in Distributed Systems:

- **Network partitioning:** If one node or network segment fails mid-transaction, locks may remain held indefinitely, reducing availability.
- **High latency and coordination overhead:** Two-phase commit introduces synchronous blocking communication between all participants.
- **Single point of failure:** The central transaction coordinator becomes a critical bottleneck.
- **Limited scalability:** ACID-style locking is incompatible with the independent, asynchronous nature of microservices.

Conclusion:

While ACID ensures strong consistency, it is *problematic in distributed systems* because it sacrifices **availability** and **fault tolerance**, violating the CAP theorem's constraints in wide-area deployments.

Saga Pattern (Orchestrated or Choreographed)

The Saga pattern offers a practical alternative by decomposing a global transaction into a sequence of local transactions that communicate through events or messages. Each local transaction updates its own data and publishes an event to trigger the next step in the workflow.

Two main implementations exist:

1. Orchestrated Saga

A central orchestrator service coordinates the workflow:

1. Sends command to OrderService → create order.
2. If successful, triggers PaymentService → process payment.
3. Then triggers InventoryService → reserve stock.
4. If any step fails, orchestrator issues compensating transactions (e.g., cancel payment, release inventory).

2. Choreographed Saga

Each service listens for events and reacts independently:

- OrderService emits OrderCreated event.
- PaymentService listens, charges customer, emits PaymentCompleted.
- InventoryService listens, reserves stock, and emits confirmation.
- Failures are handled locally via compensating actions without central coordination.

Trade-off Analysis

Aspect	ACID Transactions	Saga Pattern
Consistency Model	Strong, atomic consistency	Eventual or causal consistency
Fault Tolerance	Low — failure of one node may block all	High — each service compensates independently
Complexity	Centralized, simple logic but hard to scale	Distributed, requires compensation logic
Performance	Slower due to locking and coordination	Faster, asynchronous and non-blocking
Scalability	Limited (tightly coupled)	High (loosely coupled microservices)
Use Case	Single database, tightly coupled system	Distributed microservices, cloud applications

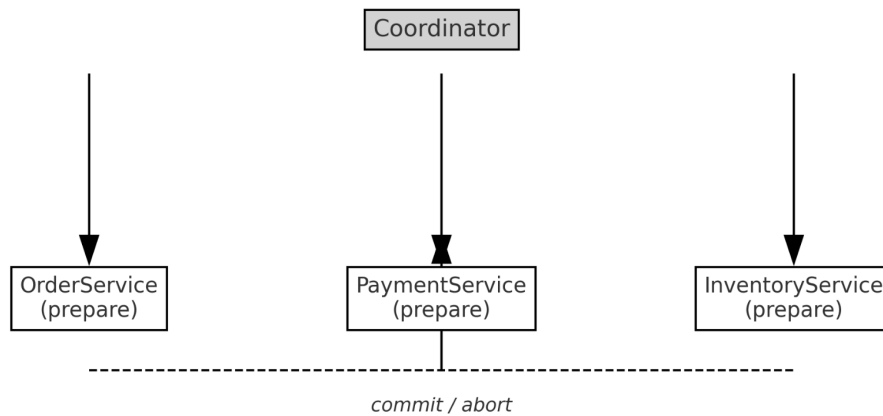


Figure 1. ACID Two-Phase Commit Workflow

Figure 1. ACID Two-Phase Commit Workflow.

A centralized coordinator ensures atomicity by managing a two-phase prepare and commit/abort process across distributed services.

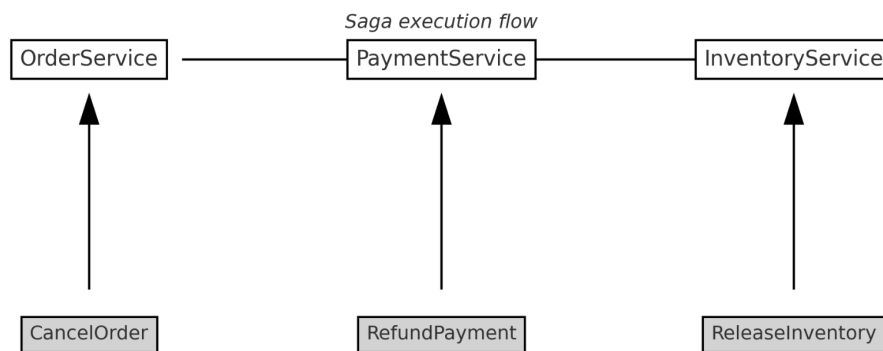


Figure 2. Saga Pattern (Orchestrated Model)

Figure 2. Saga Pattern (Orchestrated Model).

Each service performs a local transaction and triggers the next step, while compensating actions handle failures without global locks.

Observation and Conclusion

The **ACID approach** enforces immediate consistency but fails gracefully in distributed environments due to its blocking nature and dependency on a global coordinator.

In contrast, the **Saga pattern**—whether orchestrated or choreographed—embraces **eventual consistency** by allowing services to operate independently and recover through compensating actions.

Although Sagas increase implementation complexity, they provide far greater **fault tolerance, scalability, and availability**, aligning with the **CAP theorem's real-world trade-offs**.

In practice, distributed systems (e.g., e-commerce, banking, logistics) adopt Saga-based workflows to balance data integrity and system resilience without relying on global ACID semantics.