

# COMP41720 Distributed Systems Lab 2 Report

## Spring Boot Implementation with Spring Data MongoDB

### Introduction

### Source Code Repository Link

- Markdown version of this report is highly recommended: <https://github.com/zeli8888/COMP41720-Distributed-Systems/tree/main/lab2/readme.md>
- The complete code for this lab is available at: <https://github.com/zeli8888/COMP41720-Distributed-Systems/tree/main/lab2>
- To fetch the code and run on your computer, use:

```
git clone https://github.com/zeli8888/COMP41720-Distributed-Systems.git
cd lab2
```

### Lab's purpose

- Understanding data storage challenges across multiple nodes.
- Comparing different write concern levels and replication strategies (primary-backup vs leaderless)
- Experimenting with consistency models (strong, eventual, causal).
- Analyzing CAP Theorem trade-offs in specific replication and consistency settings.
- Applying architectural thinking to justify design decisions.

### Tools & Environment

- NoSQL Database: MongoDB with Replica Sets
- Client Application: Spring Boot with Spring Data MongoDB
- Environment: Docker Compose

# Setup & Configuration

## Part A: Setup & Baseline

### 1. Database Cluster Setup:

- Set up a distributed cluster for MongoDB with 3 nodes
- Documentation of setup process (Docker Compose files, commands, configurations for replication)
  - a. Create localhost volume directory:

```
mkdir -p ./data_volume/mongodb1/db
mkdir -p ./data_volume/mongodb1/configdb
mkdir -p ./data_volume/mongodb2/db
mkdir -p ./data_volume/mongodb2/configdb
mkdir -p ./data_volume/mongodb3/db
mkdir -p ./data_volume/mongodb3/configdb
```

- b. Create Docker Compose file to define 3 MongoDB nodes and a Spring Boot application container.

version: '3.8'

services:

mongodb1:

image: mongo:7.0.5

container\_name: mongodb1

command: ["--replSet", "rs0", "--bind\_ip\_all", "--port", "27017"]

volumes:

- mongodb1\_data:/data/db
- mongodb1\_config:/data/configdb

ports:

- "27017:27017"

networks:

- mongo-network

mongodb2:

image: mongo:7.0.5

container\_name: mongodb2

command: ["--replSet", "rs0", "--bind\_ip\_all", "--port", "27018"]

volumes:

- mongodb2\_data:/data/db
- mongodb2\_config:/data/configdb

ports:

- "27018:27018"

networks:

- mongo-network

mongodb3:

image: mongo:7.0.5

container\_name: mongodb3

command: ["--replSet", "rs0", "--bind\_ip\_all", "--port", "27019"]

volumes:

- mongodb3\_data:/data/db
- mongodb3\_config:/data/configdb

ports:

- "27019:27019"

networks:

- mongo-network

spring-app:

image: maven:3.8.5-openjdk-17

container\_name: spring-app

working\_dir: /app

volumes:

```
- ./lab2:/app
ports:
  - "8080:8080"
command: /bin/bash
stdin_open: true
tty: true
networks:
  - mongo-network
```

volumes:

```
mongodb1_data:
  driver: local
  driver_opts:
    type: none
    o: bind
    device: ./data_volume/mongodb1/db
mongodb1_config:
  driver: local
  driver_opts:
    type: none
    o: bind
    device: ./data_volume/mongodb1/configdb
mongodb2_data:
  driver: local
  driver_opts:
    type: none
    o: bind
    device: ./data_volume/mongodb2/db
mongodb2_config:
  driver: local
  driver_opts:
    type: none
    o: bind
    device: ./data_volume/mongodb2/configdb
mongodb3_data:
  driver: local
  driver_opts:
    type: none
    o: bind
    device: ./data_volume/mongodb3/db
mongodb3_config:
  driver: local
  driver_opts:
```

```
type: none
o: bind
device: ./data_volume/mongodb3/configdb
```

```
networks:
  mongo-network:
    driver: bridge
```

c. Start the cluster:

```
docker compose up -d
```

d. Connect to one MongoDB instance and initiate the replica set:

```
docker exec -it mongodb1 mongosh
rs.initiate(
  {
    _id: "rs0",
    members: [
      { _id: 0, host: 'mongodb1:27017' },
      { _id: 1, host: 'mongodb2:27018' },
      { _id: 2, host: 'mongodb3:27019' }
    ]
  }
)
rs.status()
```

parts of output:

```
members: [
  {
    _id: 0,
    name: 'mongodb1:27017',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 45,
    optime: { ts: Timestamp({ t: 1760443988, i: 15 }), t: Long('1') },
    optimeDate: ISODate('2025-10-14T12:13:08.000Z'),
    lastAppliedWallTime: ISODate('2025-10-14T12:13:08.654Z'),
    lastDurableWallTime: ISODate('2025-10-14T12:13:08.654Z'),
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: 'Could not find member to sync from',
    electionTime: Timestamp({ t: 1760443988, i: 1 }),
    electionDate: ISODate('2025-10-14T12:13:08.000Z'),
    configVersion: 1,
    configTerm: 1,
    self: true,
    lastHeartbeatMessage: ''
  },
  {
    _id: 1,
    name: 'mongodb2:27018',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
    uptime: 11,
    optime: { ts: Timestamp({ t: 1760443977, i: 1 }), t: Long('-1') },
    optimeDurable: { ts: Timestamp({ t: 1760443977, i: 1 }), t: Long('-1') },
    optimeDate: ISODate('2025-10-14T12:12:57.000Z'),
    optimeDurableDate: ISODate('2025-10-14T12:12:57.000Z'),
    lastAppliedWallTime: ISODate('2025-10-14T12:12:57.916Z'),
    lastDurableWallTime: ISODate('2025-10-14T12:12:57.916Z'),
    lastHeartbeat: ISODate('2025-10-14T12:13:08.468Z'),
    lastHeartbeatRecv: ISODate('2025-10-14T12:13:08.967Z'),
    pingMs: Long('0'),
    lastHeartbeatMessage: '',
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    configVersion: 1,
  }
]
```

```

        configTerm: 1
    },
    {
        _id: 2,
        name: 'mongodb3:27019',
        health: 1,
        state: 2,
        stateStr: 'SECONDARY',
        uptime: 11,
        optime: { ts: Timestamp({ t: 1760443977, i: 1 }), t: Long('-1') },
        optimeDurable: { ts: Timestamp({ t: 1760443977, i: 1 }), t: Long('-1') },
        optimeDate: ISODate('2025-10-14T12:12:57.000Z'),
        optimeDurableDate: ISODate('2025-10-14T12:12:57.000Z'),
        lastAppliedWallTime: ISODate('2025-10-14T12:12:57.916Z'),
        lastDurableWallTime: ISODate('2025-10-14T12:12:57.916Z'),
        lastHeartbeat: ISODate('2025-10-14T12:13:08.467Z'),
        lastHeartbeatRecv: ISODate('2025-10-14T12:13:08.967Z'),
        pingMs: Long('0'),
        lastHeartbeatMessage: '',
        syncSourceHost: '',
        syncSourceId: -1,
        infoMessage: '',
        configVersion: 1,
        configTerm: 1
    }
]

```

## 2. Simple Data Model:

- UserProfile (String user\_id, String username, String email, Long last\_login\_time)
- Initial Data Insertion
  - a. Connect to primary node

```
docker exec -it mongodb1 mongosh
```

### b. Insert initial data

```

use testdb;
db.UserProfile.insertOne({
    user_id: "user1",
    username: "john_doe",
    email: "john@example.com",
    last_login_time: new Date().getTime()
});

```

output:

```
{
  acknowledged: true,
  insertedId: ObjectId('68ee3e79a1ab7ca189331f75')
}
```

# Replication & Consistency Experiments

## Part B: Replication Strategies

### 1. Replication Factor / Write Concern:


- use replication factor (RF) at 3
- Demonstrate how different write concerns/levels (w:1, w:majority, w:all in MongoDB) affect write latency and durability across the cluster. Provide observations
  - a. Replication factor 3 is already set during initial cluster setup.
  - b. Run the Spring Boot application with write-concerns test:


```
docker compose exec spring-app /bin/bash
```

```
mvn spring-boot:run -Dspring-boot.run.arguments="--service=write-concern"
```

### c. Experiment Results for different write concerns:



 Testing Write Concern: w:all (averaging 500 runs, 400 docs per run)


 w:all Results:


Average Latency: 10 ms  
Min Latency: 6 ms  
Max Latency: 111 ms  
Std Deviation: 5.88 ms  
Success Count: 500/500  
Total Documents: 200000

 Testing Write Concern: w:1 (averaging 500 runs, 400 docs per run)

 w:1 Results:

Average Latency: 7 ms  
Min Latency: 5 ms  
Max Latency: 39 ms  
Std Deviation: 2.44 ms  
Success Count: 500/500  
Total Documents: 200000

 Testing Write Concern: w:majority (averaging 500 runs, 400 docs per run)

 w:majority Results:

Average Latency: 9 ms  
Min Latency: 6 ms  
Max Latency: 93 ms  
Std Deviation: 4.41 ms  
Success Count: 500/500  
Total Documents: 200000

#### d. Executive Summary & Key Observations

- Latency: w:1 (7ms avg) is the fastest, followed by w:majority (9ms avg), and finally w:all (10ms avg) as the slowest.
- Durability & Consistency: w:all and w:majority provide stronger guarantees, while w:1 prioritizes speed at the cost of potential data loss.
- Throughput & Tail Latency: While average latencies are close, the higher standard deviation and maximum latency for w:all and w:majority indicate they are more susceptible to performance degradation from node latency or network issues.

#### e. Detailed Analysis of Architectural Trade-offs

- Write Concern w:1
  - Behavior: The write operation returns as soon as the primary node has applied the write. The replication to the two secondary nodes happens asynchronously in the background.

- Data Visibility: Data is immediately visible on the primary. It might not be immediately visible on secondaries, leading to stale reads if an application reads from a secondary.
- Behavior During Failures: This is the most risky mode. If the primary node fails immediately after acknowledging the write and before the data is replicated, the data can be lost forever. A new primary will be elected from the secondaries that may not have received the write.
- Architectural Trade-off: This configuration prioritizes Availability and Partition Tolerance (AP) from the CAP theorem. It sacrifices strong Consistency (C) for low latency and high availability of writes. The system remains writable even if some secondaries are disconnected, as long as the primary is alive.
- Write Concern w:majority
  - Behavior: The write operation returns only after a majority of nodes (2 out of 3 in this RF=3 cluster) have acknowledged the write.
  - Data Visibility: The written data is guaranteed to be visible on at least 2 nodes. This prevents stale reads if your read concern is also set to majority. It offers strong consistency.
  - Behavior During Failures: This configuration is highly resilient. Even if the primary node fails, the data is already on at least one other node. During a network partition, if a majority of nodes (including the primary) can still communicate, writes can continue. If the primary is in the minority partition, it becomes read-only.
  - Architectural Trade-off: This is the classic CP (Consistency & Partition Tolerance) configuration. It ensures data consistency and durability even during failures, but at the cost of higher latency and potentially reduced availability. If a majority of nodes cannot be reached, writes will fail, making the system less available for write operations.
- Write Concern w:all
  - Behavior: The write operation returns only after all nodes in the replica set (all 3 in this case) have acknowledged the write.
  - Data Visibility: The data is guaranteed to be on every single node. This is the strongest possible durability guarantee.
  - Behavior During Failures: This configuration is the most fragile in terms of availability. If any single node becomes unavailable or slow, all write operations will block or fail. The system's write availability is directly tied to the health of every single node.
  - Architectural Trade-off: This is an even stricter form of CP. While it offers the ultimate durability guarantee, it severely impacts Availability (A). The latency is

the highest, and the system's ability to tolerate any node failure is zero for write operations.

f. Justifying Configuration Choice for Business Requirements


- When to Choose w:majority (The Balanced CP Choice)
  - Business Requirement: Financial transactions, user account data, inventory management systems, or any application where data correctness is paramount.
  - Justification: This is the most common choice for applications requiring strong consistency. It provides an excellent balance between durability and acceptable latency. It accepts a slight performance penalty to gain the guarantee that once a write is acknowledged, it will survive a single node failure and will be consistent across the cluster. The cost (slightly higher latency) is a worthy trade-off for the benefit (data safety).
- When to Choose w:1 (The Performance AP Choice)
  - Business Requirement: High-volume telemetry data, application logs, social media activity streams, real-time analytics, or any scenario where throughput and speed are more critical than perfect durability.
  - Justification: I would choose this when the business can tolerate occasional, small amounts of data loss in exchange for blistering performance. For example, losing a few log lines or a single user's "click" event is acceptable if it means you can process millions of such events per second. This embodies the "why" of prioritizing user experience and system scalability over absolute data correctness for non-critical data. The cost (risk of data loss) is accepted for the benefit (system speed and responsiveness).
- When to Choose w:all (The Strict CP / Niche Choice)
  - Business Requirement: Extremely critical configuration data, master encryption keys, or legal records where every single copy must be identical at all times.
  - Justification: This is rarely used for general application writes due to its severe impact on availability and high tail latency. The reason here is often regulatory, requirement for absolute certainty. The business is explicitly stating that the integrity of this specific data is so critical that they are willing to have their entire write process halt if any single database node has a problem.


2. Leader-Follower (Primary-Backup) Model:


- demonstrate writes and reads against the primary and how data propagates to followers
- Simulate a primary node failure and observe how the system elects a new primary and handles ongoing operations. Note any downtime or data inconsistencies
- MongoDB doesn't support Leaderless (Multi-Primary) Model, so this part is skipped.
  - a. Run the Spring Boot application with replication test:


```
docker compose exec spring-app /bin/bash
mvn spring-boot:run -Dspring-boot.run.arguments="--service=replication"
```


b. Experiment Results for writes against primary and propagates to followers (using default write concern w:1)


 TEST 1: WRITE PROPAGATION TO FOLLOWERS

 Writing 50 documents to primary...


 Primary write completed in 427ms

 Verifying data propagation to secondaries...


 Data Propagation: 50/50 documents

 Success rate: 100.00%


c. Experiment Results for reads against primary and followers

 TEST 2: READ PREFERENCE BEHAVIOR


Testing: Primary

 Document found - Latency: 0ms

Testing: Secondary

 Document found - Latency: 0ms

Testing: Primary Preferred

 Document found - Latency: 0ms

d. Experiment Results for primary node failure, new primary election and downtime data inconsistencies

### 🔧 TEST 3: PRIMARY FAILOVER WITH stepDown COMMAND

Current Primary: mongodb1:27017

Starting concurrent operations during failover...

🔥 Triggering stepDown on primary...

✅ StepDown command sent

📊 Monitoring failover process...

```
2025-10-13T23:34:57.608Z INFO 356 --- [lab2] [-mongodb1:27017] org.mongodb.driver.
ET_OTHER, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=
alAddress=mongodb1:27017, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019], p
2025-10-13T23:34:57.608Z INFO 356 --- [lab2] [-mongodb1:27017] org.mongodb.driver.
ET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVers
onicalAddress=mongodb1:27017, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-13T23:34:57.623Z INFO 356 --- [lab2] [onPool-worker-1] org.mongodb.driver.
tor, topology description: {type=REPLICA_SET, servers=[{address=mongodb1:27017, typ
2025-10-13T23:34:58.607Z INFO 356 --- [lab2] [main] org.mongodb.driver.
rSelector{readPreference=primary}, topology description: {type=REPLICA_SET, servers
2025-10-13T23:35:06.411Z INFO 356 --- [lab2] [-mongodb3:27019] org.mongodb.driver.
ET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVers
nicalAddress=mongodb3:27019, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-13T23:35:06.412Z INFO 356 --- [lab2] [-mongodb2:27018] org.mongodb.driver.
ET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVers
nicalAddress=mongodb2:27018, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-13T23:35:07.898Z INFO 356 --- [lab2] [-mongodb3:27019] org.mongodb.driver.
ET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVers
nicalAddress=mongodb3:27019, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-13T23:35:07.905Z INFO 356 --- [lab2] [-mongodb3:27019] org.mongodb.driver.
ET_PRIMARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersic
calAddress=mongodb3:27019, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019],
2025-10-13T23:35:07.905Z INFO 356 --- [lab2] [-mongodb3:27019] org.mongodb.driver.
```

⚡ New primary elected after 10300ms

📊 Concurrent operations: 20 successful, 0 failed

New Primary: mongodb3:27019

✅ Failover successful

### e. Executive Summary & Key Observations

#### ◦ Write Propagation

- 100% Success Rate: All 50 documents written to the primary were successfully replicated to secondaries
- Data Visibility: Immediate consistency on primary, near-real-time on secondaries

#### ◦ Read Preferences

- Primary Reads: Guaranteed latest data with minimal latency (0ms)

- Secondary Reads: Potentially stale data but with same low latency
- Primary Preferred: Automatic failover to secondaries if primary unavailable
- Failover Process
  - Election Time: ~10.3 seconds for new primary election
  - Downtime Impact: 20 concurrent operations experienced brief unavailability
  - Data Consistency: Zero data loss or inconsistencies observed
  - Recovery Pattern: Seamless transition with driver-level retry mechanisms
- f. Detailed Analysis of Architectural Trade-offs
  - Write Propagation Behavior
    - Asynchronous Replication: The primary acknowledges writes immediately (default w:1) and replicates to secondaries in the background
    - Eventual Consistency: Secondaries eventually receive all writes, but may lag behind the primary
  - Read Consistency Models
    - Strong Consistency (Primary): Reading from primary guarantees the most recent writes
    - Eventual Consistency (Secondary): Reading from secondaries may return stale data during replication lag
    - High Availability (Primary Preferred): Provides automatic failover while preferring strong consistency when available
  - Failure Handling & Availability
    - Failover Time: 10.3-second election represents significant write unavailability
    - Read Availability: Secondary reads remained available during primary failure
    - Data Durability: Zero failed operations suggests driver-level retry mechanisms handled the transition
  - CAP Theorem Positioning
    - CP During Normal Operations: Prioritizes Consistency (C) and Partition Tolerance (P) by ensuring a single primary for writes and tolerating node failures
    - AP During Failover: Becomes Available for reads (via secondaries) but sacrifices write availability until a new primary is elected
    - Partition Tolerance: The system can handle node failures and network partitions, but write availability is compromised during primary elections
- g. Justifying Configuration Choice for Business Requirements
  - When to Prefer Strong Consistency (Primary Reads)
    - Business Requirement: Financial systems, e-commerce checkouts, reservation systems where reading the latest data is critical

- Justification: primary reads can guarantee data freshness, which makes this ideal for transactional systems. The "why" is risk avoidance - the cost of acting on stale data (e.g., double-spending, overbooking) far outweighs the performance benefits of reading from secondaries.
- When to Accept Eventual Consistency (Secondary Reads)
  - Business Requirement: Social media feeds, product catalogs, analytics dashboards where slight staleness is acceptable
  - Justification: low latency on secondary reads makes this suitable for read-scaling. The "why" is scalability - distributing read load to secondaries improves overall system throughput without significant data freshness compromise.
- When This Leader-Follower (Primary-Backup) Model: Fits
  - Business Requirement: Applications where data consistency and integrity are non-negotiable, such as financial transaction systems, inventory management, or user account data.
  - Justification: The single-primary design is ideal when the business cannot tolerate data inconsistencies or conflicts. The temporary write unavailability during failover (10.3 seconds in our test) is an acceptable trade-off to ensure that once a write is acknowledged, it becomes the single source of truth. The system provides excellent fault tolerance for data protection while maintaining straightforward operational semantics.
- When to Consider Alternative Leaderless (Multi-Primary) Model
  - Business Requirement: Applications requiring sub-second failover or multi-region writes
  - Justification: The 10+ second election time may be unacceptable for high-availability requirements. The inability to write to multiple nodes simultaneously limits geographic distribution. The "why" is latency sensitivity - for global applications, the delay in write availability during failover could impact user experience or business operations. In such cases, a multi-primary model would provide better availability at the cost of increased complexity and potential consistency issues.

## Part C: Consistency Models

### 1. Strong Consistency:

- Configure both writes and reads to demand strong consistency (w:majority for writes and readConcern:majority for reads in MongoDB).
- Perform a write operation on one node and immediately attempt to read it from another node. Verify that the data is immediately consistent.

- Introduce a network partition or node failure during this experiment. Observe the impact on write/read operations (e.g., does it block, throw an error, become unavailable?). Relate this observation directly to the CAP theorem.

a. Run the Spring Boot application with consistency test:

```
docker compose exec spring-app /bin/bash  
mvn spring-boot:run -Dspring-boot.run.arguments="--service=consistency"
```

b. Experiment Results for Strong Consistency with immediate read from secondary after write in primary



STRONG CONSISTENCY TEST



Write completed with w:majority



Strong consistency verified - immediate read from secondary



Document content: username\_strong\_write

c. Experiment Results for Strong Consistency with network partition during write/read



Testing during network partition...

🔥 StepDown command sent to simulate network partition

```
2025-10-14T01:18:07.999Z INFO 1655 --- [lab2] [-mongodb3:27019] org.mongodb.driver
SET_OTHER, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=0,
localAddress=mongodb3:27019, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019], p
2025-10-14T01:18:08.000Z INFO 1655 --- [lab2] [-mongodb3:27019] org.mongodb.driver
SET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=0,
localAddress=mongodb3:27019, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-14T01:18:08.018Z INFO 1655 --- [lab2] [onPool-worker-2] org.mongodb.driver
rSelector{readPreference=primary}, topology description: {type=REPLICA_SET, servers=
2025-10-14T01:18:08.018Z INFO 1655 --- [lab2] [onPool-worker-1] org.mongodb.driver
tor, topology description: {type=REPLICA_SET, servers=[{address=mongodb1:27017, typ
2025-10-14T01:18:17.101Z INFO 1655 --- [lab2] [-mongodb1:27017] org.mongodb.driver
SET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=0,
localAddress=mongodb1:27017, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-14T01:18:17.110Z INFO 1655 --- [lab2] [-mongodb2:27018] org.mongodb.driver
SET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=0,
localAddress=mongodb2:27018, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-14T01:18:18.854Z INFO 1655 --- [lab2] [-mongodb1:27017] org.mongodb.driver
SET_SECONDARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=0,
localAddress=mongodb1:27017, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019]
2025-10-14T01:18:18.862Z INFO 1655 --- [lab2] [-mongodb1:27017] org.mongodb.driver
SET_PRIMARY, cryptd=false, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=0,
localAddress=mongodb1:27017, hosts=[mongodb2:27018, mongodb1:27017, mongodb3:27019],
2025-10-14T01:18:18.862Z INFO 1655 --- [lab2] [-mongodb1:27017] org.mongodb.driver
Read operations: 5 successful, 0 failed
Write operations: 5 successful, 0 failed
```

#### d. Executive Summary & Key Observations

- Immediate Data Visibility: With w:majority and readConcern:majority, data written to primary was immediately visible on secondary nodes, demonstrating strong consistency.
- Latency Impact: Strong consistency operations showed slightly higher latency due to the coordination required between nodes to achieve consensus.
- Failure Behavior: During network partitions, the system experienced temporary unavailability while the replica set reorganized and elected a new primary. However, once a new primary was established, both read and write operations resumed successfully with zero data loss.
- CAP Theorem Alignment: The behavior confirms CP (Consistency & Partition Tolerance) characteristics - during partitions, the system prioritized consistency over availability, temporarily blocking operations until consistency could be guaranteed.

#### e. Detailed Analysis of Architectural Trade-offs

- Behavior: Write operations block until a majority of nodes acknowledge the write. Read operations only return data that has been replicated to a majority of nodes, ensuring consistency.
- Data Visibility: Data becomes visible on secondaries after replication latency. While not instantaneous, readConcern:majority prevents reading uncommitted or rolled-back data, providing strong consistency guarantees despite replication delay.
- Failover Mechanism: This configuration demonstrates classic CP behavior from the CAP theorem:
  - The isolated primary detects it can no longer reach a majority of nodes
  - The primary steps down to prevent inconsistent writes
  - The majority partition undergoes leader election
  - No writes are accepted anywhere in the cluster until a new primary with guaranteed consistency is established
- Architectural Trade-off:
  - This configuration explicitly prioritizes Consistency (C) and Partition Tolerance (P) over Availability (A).
  - The trade-off is clear: during partitions, the system may become temporarily unavailable to maintain data consistency.
  - The "why" behind this choice is that for certain applications, data correctness is non-negotiable, even if it means occasional downtime.

#### f. Justifying Configuration Choice for Business Requirements

- When to Choose Strong Consistency (CP Configuration)
  - Business Requirement: Financial transactions, medical systems, legal compliance data, or any scenario where data accuracy has significant consequences.
  - Justification: Choose strong consistency when the cost of data inconsistency exceeds the cost of temporary unavailability. For financial systems, incorrect balances could lead to regulatory penalties and loss of trust. The brief unavailability during failover is preferable to data corruption.

## 2. Eventual Consistency:


- Configure writes and reads for eventual consistency (w:1 for writes and default read concern in MongoDB).
- Perform a write operation on one node. Immediately attempt to read it from another node. Observe if you can read stale (old) data before it propagates.
- Implement a simple loop that repeatedly reads the data until the latest value is observed, demonstrating the "eventual" nature.


- Discuss scenarios where eventual consistency is acceptable or even desirable (e.g., social media likes, sensor data) and why this choice is beneficial in those contexts (e.g., availability, performance).

a. Run the Spring Boot application with consistency test:

```
docker compose exec spring-app /bin/bash
mvn spring-boot:run -Dspring-boot.run.arguments="--service=consistency"
```

b. Experiment Results for Eventual Consistency with read from secondary after write in primary

 EVENTUAL CONSISTENCY TEST

- ✓ Write completed with w:1
- ✓ Document propagated after 104ms on attempt 2
-  Final document content: username\_eventual\_write

c. Executive Summary & Key Observations

- Stale Data Visibility: Initial reads from secondary nodes returned outdated data, confirming the presence of stale data during replication windows.
- Propagation Latency: The system demonstrated a 104ms replication delay, with data becoming consistent across all nodes after the second read attempt.
- Eventual Nature Proven: The retry loop successfully captured the transition from inconsistent to consistent state, validating the "eventual" guarantee.
- CAP Theorem Alignment: The behavior confirms AP (Availability & Partition Tolerance) characteristics - the system remains available for reads and writes even during partitions, accepting temporary inconsistencies.

d. Detailed Analysis of Architectural Trade-offs

- Behavior: Write operations return immediately after primary acknowledgment (w:1), while read operations may return stale data from secondary nodes until replication completes.
- Data Visibility: The system exhibits temporary inconsistency windows where different nodes show different versions of the same data. This is the fundamental trade-off for achieving higher availability and lower latency.
- Architectural Trade-off:
  - This configuration explicitly prioritizes Availability (A) and Partition Tolerance (P) over Consistency (C).
  - The trade-off is intentional: temporary data inconsistency is acceptable in exchange for continuous system availability and better performance.

- The "why" behind this choice is that for many modern applications, being always available provides better user experience than guaranteed consistency, especially when the cost of inconsistency is low.

#### e. Justifying Configuration Choice for Business Requirements

- When to Choose Eventual Consistency (AP Configuration)
  - Business Requirement: Social media interactions (likes, comments), recommendation systems and any scenario where temporary data staleness doesn't impact core functionality.
  - Justification: Choose eventual consistency when system availability and performance are more critical than immediate data accuracy. For social media platforms, showing a like count that's a few seconds old doesn't harm user experience, but site unavailability would. The business benefits from low write latency and high read throughput that this model enables.

### 3. Causal Consistency (Optional / Bonus):

- design an experiment to demonstrate that causally related operations are observed in order, even if other concurrent operations are not.

#### a. Run the Spring Boot application with consistency test:

```
docker compose exec spring-app /bin/bash
mvn spring-boot:run -Dspring-boot.run.arguments="--service=consistency"
```

#### b. Experiment Results for Causal Consistency with related operations observed in order

##### CAUSAL CONSISTENCY TEST

- ✓ Created cause document: user\_causal\_1\_1760404699902
- ✓ Created effect document: user\_causal\_2\_1760404699902 (references user\_causal\_1\_1760404699902)
- Verifying causal order:
  - Round 1: Cause=FOUND, Effect=FOUND
- ✓ Causal consistency verified - related operations maintain order

#### c. Executive Summary & Key Observations

- Causal Relationship Preservation: The system consistently maintained the causal relationship between documents - the effect document was never observed without its cause document present.
- Order Guarantee: Causally related operations (creation of cause followed by effect) were always observed in the correct temporal sequence across all read attempts.
- CAP Theorem Positioning: Causal consistency represents a pragmatic middle ground in the CAP spectrum, offering stronger guarantees than eventual consistency while avoiding the availability costs of strong consistency.

#### d. Detailed Analysis of Architectural Trade-offs

- Behavior: The system tracks causal dependencies between operations, ensuring that if operation A causally precedes operation B, then any node that observes B will also observe A and will see them in the correct order.
- Data Visibility: Readers always see a causally consistent view of the data - no causal violations occur, but the system may present different consistent snapshots to different readers at the same time.
- Architectural Trade-off:
  - This configuration balances consistency requirements with system availability and performance.
  - The trade-off is nuanced: we accept that different clients might see different orders of concurrent operations, but we strictly preserve causal relationships.
  - The "why" behind this choice is that many real-world applications require causal ordering (which matches human intuition about cause-and-effect).
- e. Justifying Configuration Choice for Business Requirements
  - When to Choose Causal Consistency (Practical CP Configuration)
    - Business Requirement: Collaborative applications (Google Docs, multiplayer games), social media feeds and any scenario where causal relationships matter.
    - Justification: Choose causal consistency when we need stronger guarantees than eventual consistency but cannot afford the availability penalties of strong consistency. For collaborative editing applications, it's critical that if someone edits a paragraph (cause) and then someone else comments on that edit (effect), the comment never appears without the edit. The business benefits from natural alignment with how humans perceive cause-and-effect relationships

## Distributed Transactions

### Part D: Distributed Transactions (Conceptual / Optional Coding)

1. Review the challenges of distributed transactions (e.g., Saga pattern)
  - Challenges of Two/Three-Phase Commit Pattern:
    - a. Performance Bottlenecks: Long-lived locks held on resources across all participating services for the entire transaction duration.
    - b. Single Points of Failure: The central transaction coordinator becomes a critical bottleneck and a single point of failure.

- c. Scalability Issues: Distributed locks and synchronous communication hinder horizontal scaling.
- d. Blocking Problem: If the coordinator fails, participants may remain in a blocking state with locks held, awaiting a decision.
- e. Protocol Overhead: Multiple rounds of network communication (prepare, commit) increase latency.
- Challenges of the Saga Pattern:
  - a. Lack of Isolation: The absence of long-lived locks can lead to dirty reads, lost updates, or other phenomena if not managed.
  - b. Complexity of Compensating Actions: Designing and implementing correct, idempotent rollback logic for each step adds significant development complexity.
  - c. Eventual Consistency: The system is only eventually consistent, which can be challenging for business logic that requires immediate, strong consistency.
  - d. Debugging and Monitoring: Tracing a long-running, distributed business flow across multiple services is more complex than monitoring a single database transaction.
  - e. Dependency Management: In choreography-based Sagas, the event-driven flow can become complex and difficult to manage as the number of services grows.

## 2. Conceptual Exercise: for a simple multi-service workflow (e-commerce order involving OrderService, PaymentService, InventoryService)

- Describe how this workflow would be managed using:
  - a. ACID transactions (and why it's problematic in a truly distributed system).
    - In a traditional ACID model, this workflow would be managed as a single distributed transaction spanning all three services. This would typically employ protocols like Two-Phase Commit (2PC) or Three-Phase Commit (3PC) to maintain atomicity across service boundaries.
    - Key Characteristics
      - Atomicity: All operations commit or rollback together
      - Consistency: Strong consistency guarantees throughout
      - Isolation: Intermediate states are not visible externally
      - Durability: Once committed, changes are permanent
    - Problems
      - Availability Impact: The entire transaction holds resources (locks) across services until completion, creating potential deadlocks and reducing system availability.
      - Performance Bottlenecks: Synchronous coordination between services introduces significant latency due to network round-trips for coordination messages, blocking while waiting for participant responses and lock contention across service boundaries

- Scalability Limitations: Tight coupling prevents independent scaling of services and creates single points of failure.
- Operational Complexity: Recovery from coordinator failures requires complex reconciliation procedures, often necessitating manual intervention.

b. Sagas (Orchestrated or Choreographed).

Sagas break the distributed transaction into a sequence of local transactions, each with corresponding compensation actions for rollback scenarios.

a. Orchestrated Saga Implementation

- Workflow Execution:
  - Saga Orchestrator initiates the workflow
  - Sends command to OrderService → creates order in "PENDING" state
  - Sends command to PaymentService → processes payment
  - Sends command to InventoryService → reserves inventory
  - Updates order to "CONFIRMED" state upon success
- Compensation Sequence:
  - If inventory reservation fails: triggers payment refund → order cancellation
  - If payment fails: triggers immediate order cancellation
  - Each compensation is a separate transaction with its own durability

b. Choreographed Saga Implementation

- Event-Driven Flow:
  - OrderService publishes OrderCreated event
  - PaymentService consumes event → processes payment → publishes PaymentProcessed
  - InventoryService consumes payment event → reserves stock → publishes InventoryReserved
  - OrderService consumes inventory event → updates order status
- Compensation via Events:
  - Services listen for failure events and execute their compensation logic
  - PaymentService listens for InventoryReservationFailed → triggers refund
  - OrderService listens for compensation events → updates order status accordingly

- Analyze the trade-offs between these approaches in terms of consistency, complexity, fault tolerance, and performance. You do not need to implement this part; a detailed conceptual explanation is sufficient.

a. Consistency Trade-offs

- ACID Transactions:
  - Strong Consistency: All services see consistent state at all times
  - Immediate Rollback: Failed transactions leave no side effects

- Simplified Reasoning: Linear, predictable execution paths
- Saga Pattern:
  - Eventual Consistency: Temporary inconsistencies during execution
  - Compensation Latency: Rollback is asynchronous and may have delays
  - Complex Failure States: Must handle partial failures and compensation failures
- b. Complexity Trade-offs
  - ACID Transactions:
    - Lower Business Logic Complexity: Simple commit/rollback semantics
    - Higher Infrastructure Complexity: Requires sophisticated distributed transaction managers
    - Simplified Error Handling: Binary success/failure outcomes
  - Saga Pattern:
    - Higher Business Logic Complexity: Must design and test compensation logic
    - Lower Infrastructure Complexity: Leverages existing messaging infrastructure
    - Complex Error Recovery: Must handle compensation failures and idempotency
  - Orchestrated vs. Choreographed Complexity:
    - Orchestrated: Centralized control simplifies monitoring but creates single point of failure
    - Choreographed: Better decoupling but harder to debug and monitor distributed logic
- c. Fault Tolerance Evaluation
  - ACID Transactions:
    - Brittle Failure Modes: Single service failure blocks entire system
    - Recovery Challenges: In-doubt transactions require manual resolution
    - Cascading Failures: Performance issues in one service affect all participants
  - Saga Pattern:
    - Resilient Design: Individual service failures don't block entire system
    - Graceful Degradation: Can operate partially while waiting for recovery
    - Recovery Automation: Built-in compensation mechanisms enable self-healing
- d. Performance Characteristics
  - ACID Transactions:
    - High Latency: Synchronous coordination and locking overhead
    - Poor Throughput: Limited by slowest participant service
    - Resource Intensive: Long-held locks and connections
  - Saga Pattern:
    - Better Responsiveness: Asynchronous execution improves user experience
    - Higher Throughput: Parallel execution opportunities in choreographed approach
    - Resource Efficiency: Shorter-lived transactions and connections



#### e. Strategic Considerations

- Use ACID Transactions When:
  - Strong consistency is non-negotiable (financial settlements, regulatory requirements)
  - Transaction scope is limited to services with co-located databases
  - Performance requirements permit synchronous coordination
  - Operational team has expertise in distributed transaction management
- Use Saga Pattern When:
  - Services require autonomy and independent scalability
  - Eventual consistency is acceptable for the business domain
  - High availability and fault tolerance are critical requirements
  - System must remain responsive under partial failures

## Conclusion

- Write Concerns
  - Different write concern levels demonstrated clear trade-offs between latency and durability
  - w:1 offered the best performance but with potential data loss risk
  - w:majority and w:all provided stronger consistency guarantees at the cost of higher latency and reduced availability.
- Replication Models
  - Leader-Follower (Primary-Backup) Model provided strong consistency and straightforward conflict resolution but suffered from write unavailability during failover.
  - Leaderless (Multi-Primary) Model offers improved availability and write distribution but introduces complexity in conflict resolution and consistency management.
- Consistency Models
  - Strong consistency (via w:majority and readConcern:majority) ensures that write and read were acknowledged by a majority of nodes, establishing an agreed-upon truth for the system. This comes at the cost of higher latency and, during network partitions, sacrificing the availability of clients connections.
  - Eventual consistency (w:1) showed temporary data staleness but offered better performance.
  - Causal consistency emerged as a practical middle ground, preserving logical operation ordering without strong consistency's availability costs.
- Distributed Transactions Analysis
  - The conceptual comparison revealed that while ACID transactions provide simplicity through strong consistency, the Saga pattern offers better scalability and fault tolerance for distributed workflows, albeit with increased complexity in compensation logic.