Documentation

GitHub link: <https://github.com/Anush2712/Distributed-systems-Lab-2.git>

Aim and objective of this experimentation:

The main objectives of this lab are:

1. **To set up a distributed MongoDB environment** with a three-node replica set using Docker containers.
2. **To understand data replication** and how MongoDB maintains consistency across primary and secondary nodes.
3. **To explore read and write operations** with different writeConcern levels (1, "majority", "all") and readPreference settings (primary, secondary).
4. **To observe failover behavior** when the primary node becomes unavailable and how the secondary node is promoted to primary.
5. **To verify data integrity and availability** under node failures in a distributed database system.

This lab demonstrates the practical aspects of replication, fault tolerance, and data consistency in a NoSQL distributed environment, which are critical for designing highly available applications.

Introduction :

The purpose of this lab is to explore and understand **replication and fault tolerance in distributed databases** using a **MongoDB replica set**. By setting up a multi-node MongoDB cluster, we aim to study how data is replicated across nodes, how read and write operations behave with different **write concerns** and **read preferences**, and how failover occurs when the primary node becomes unavailable.

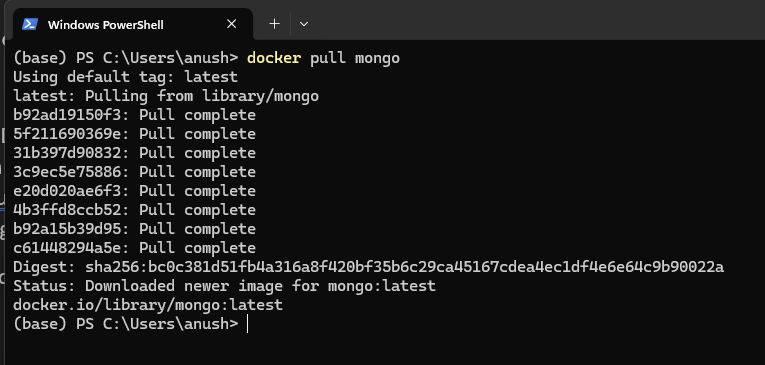
For this lab, we used **MongoDB (v7.0.25)** running in **Docker containers** to simulate a three-node distributed environment. The **Mongosh shell** was used to interact with the database, perform CRUD operations, configure replication, and verify data consistency across the nodes.

Explaination for task A:

Database chosen: Mongo DB (No SQL)

Tools using: Docker and Mongosh(Mongo DB shell)

Command and execution



First we will be downloading MongoDB docker images latest version from docker hub into our machine. The docker images contain MongoDB preinstalled and pre configured for the container up and running. Once the images are pulled we can start multiple container such as mongo1, mongo2 and mango3 without re-downloading

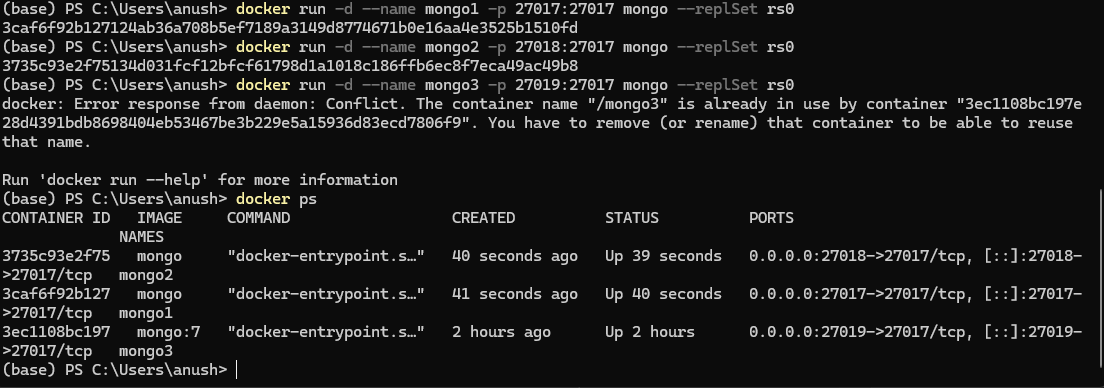
In short we are pulling images first so that docker has the MongoDB software ready to start our distributed cluster

No we use this command to start 3 separate MongoDB containers with replication enabled

docker run -d --name mongo1 -p 27017:27017 mongo --replSet rs0

docker run -d --name mongo2 -p 27018:27017 mongo --replSet rs0

docker run -d --name mongo3 -p 27019:27017 mongo --replSet rs0



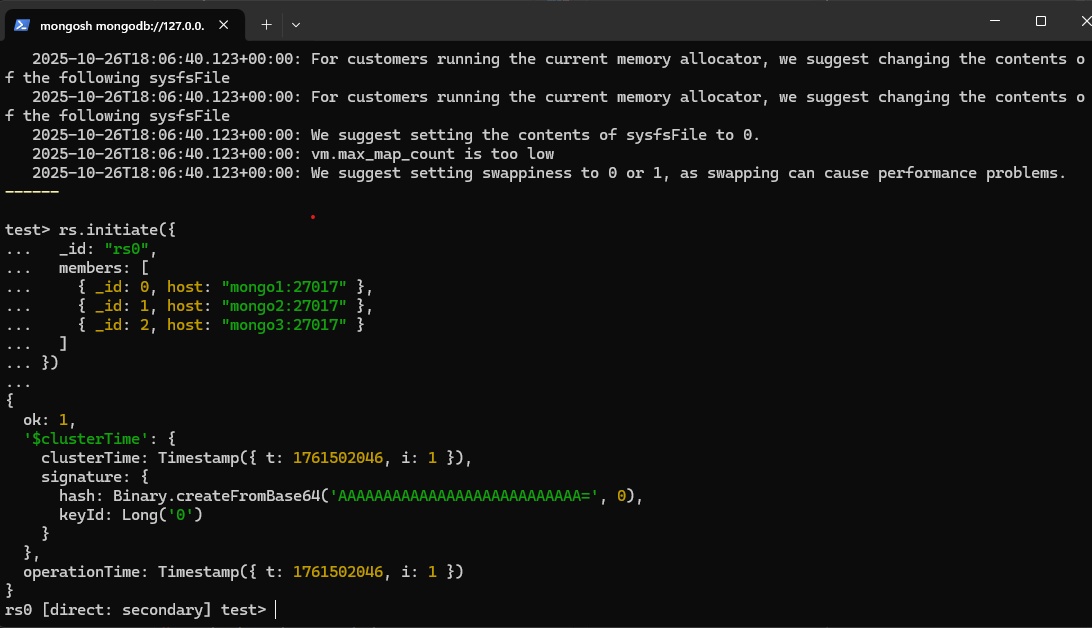
this runs docker images in detached mode which mean runs in background rather than blocking our terminal and -p gives a unique name for each container and 27017, 27018, 27019 are for mongo1, mongo2, mongo3 respectively mapping them back to internal MongoDB port of our host machine allowing us to connect to each Mongo DB node from our machine and “--replSet rs0” tells MongoDB to start this instance as part of a replica set called rs0. This is necessary for replication (having one primary and multiple secondary nodes) and to demonstrate failover and read preferences.

Next we are initializing the replica set in MongoDB. This is what makes our 3 nodes a proper distributed cluster that can replicate data

docker exec -it mongo1 mongoshOnce inside the Mongo shell, you’ll see a prompt like:

test>

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





From rs.status we can confirm that replica set is up and running,

mongo1:27017 – primary which accepts writes and all changes are replicated to secondaries

mongo2:27017 and mongo3:27017 – secondary which mean these only replicate the data from primary, we can read from them if you set the readPreference to secondary

all members have health:1 which mean everyone is healthy

here we can see ok:1 which means the replica set is functioning normally which is good sign

Next testing our cluster making sure our distributed system works correctly when replicated, read, written or failed over

Test writing to the primary :

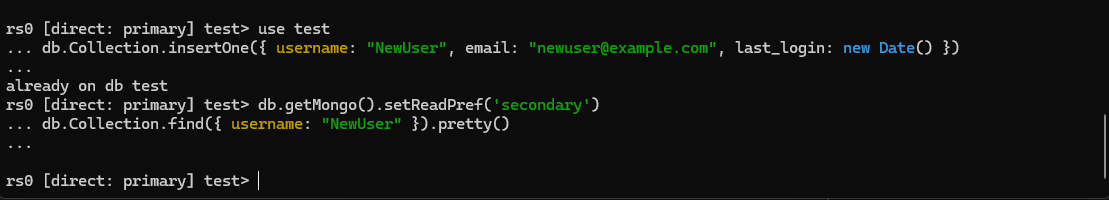
use test

db.Collection.insertOne({ username: "NewUser", email: "newuser@example.com", last\_login: new Date() })

test reads from secondary :

db.getMongo().setReadPref('secondary')

db.Collection.find({ username: "NewUser" }).pretty()



This query returns the document NewUser, which means replication woks and our cluster setup is successful y

setReadPref('secondary') → tells MongoDB to read data **from a secondary node** instead of the primary.

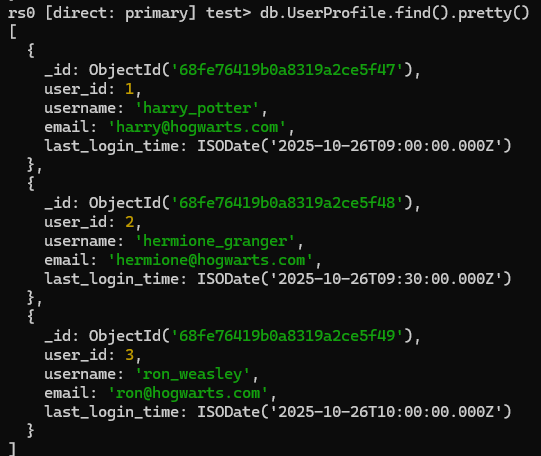
find({ username: "NewUser" }) → checks if the document you inserted on the primary has **replicated successfully** to the secondary node.

Step 2 is populating the data in the database using test databse



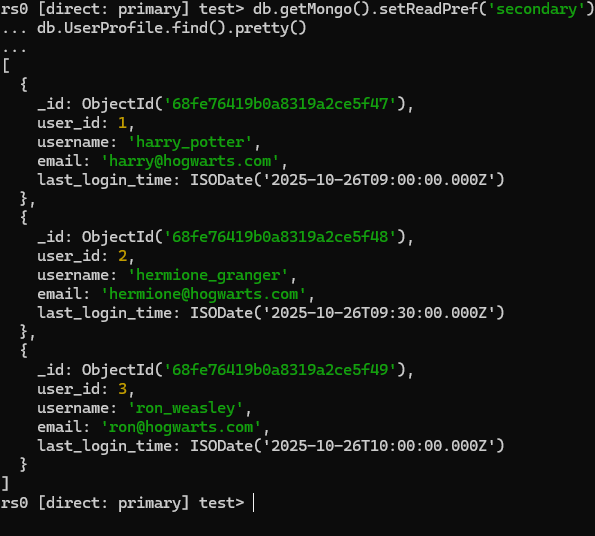
The MongoDB automatically create data when we try to insert it and acknowledge shows true which means the userProfiles are successfully created

When db.UserProfile.find().pretty() command is used we can see all the inserted document. Which helps us to verify the data on primary node



db.getMongo().setReadPref('secondary')  
db.UserProfile.find().pretty()

this command help test replication and check if the data appears in secondary node which conforms that our cluster is functioning properly

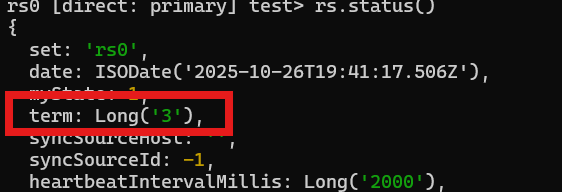


Part B: Replication Strategie

1. Confirm Your Replication Factor

Step 1: The main goal for part B for me is to understand how MongoDB handles replication factors and write concerns and how the effect latency and durability across my clusters

For this first I confirmed my replicating factor (RF) which should be ideally 3 cause I have 3 clusters. This can be verified using **rs.status** which returns mongo1, mongo2, mongo3 meaning data is replicated to 3 nodes (1 primary and 2 secondaries)

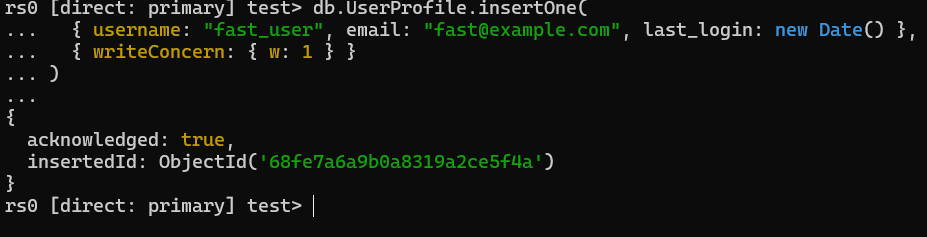


Step 2: Test Different Write Concerns

In MongoDB, writeConcern controls **how many nodes must acknowledge a write** before it’s considered successful.

1. W:1 only primary acknowledges the write. This is fast but less durable

db.UserProfile.insertOne(  
 { username: "fast\_user", email: "fast@example.com", last\_login: new Date() },  
 { writeConcern: { w: 1 } })



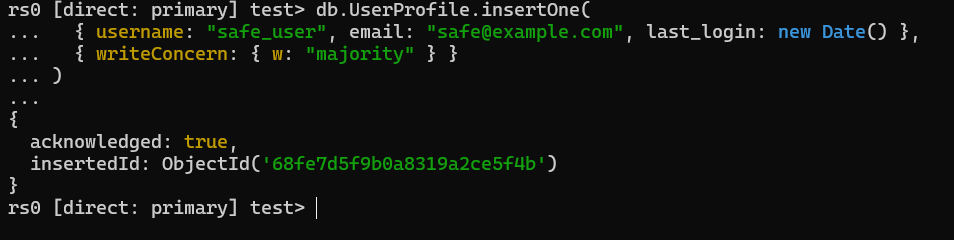
1. W: majority – write acknowledged only after the majority (2&3) members conformed. This has slightly slower write latency and High durability (guaranteed to survive primary failure)

db.UserProfile.insertOne(

{ username: "safe\_user", email: "safe@example.com", last\_login: new Date() },

{ writeConcern: { w: "majority" } }

)



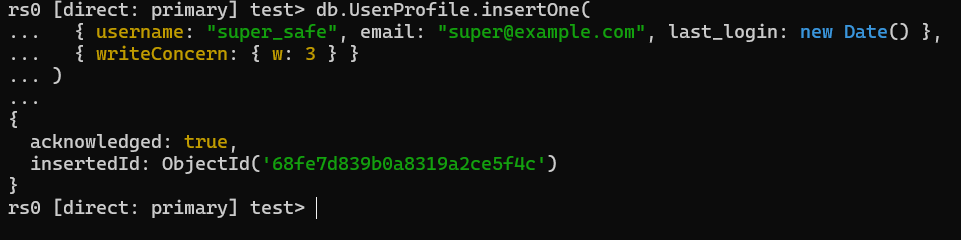
c) w:3 — All nodes

Write acknowledged only after **all 3 nodes** replicate the data. Slowest write latency but maximum durability (write survives even if 2 nodes go down right after)

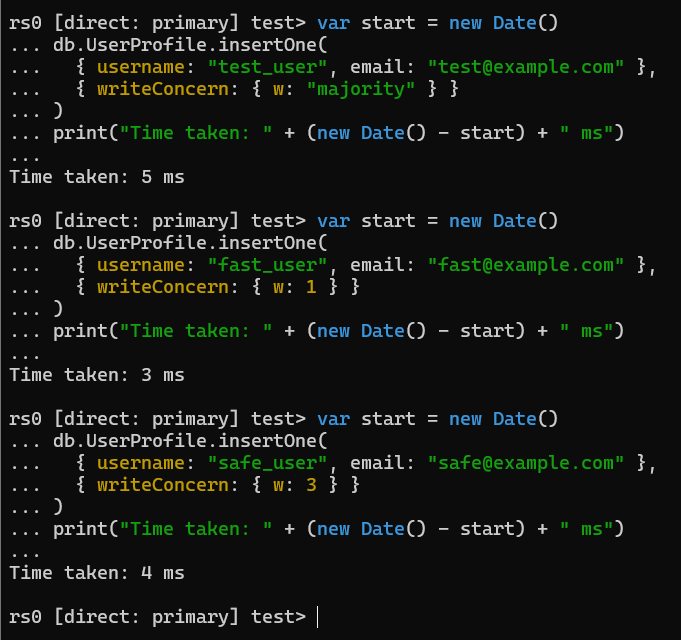
db.UserProfile.insertOne(

{ username: "super\_safe", email: "super@example.com", last\_login: new Date() },

{ writeConcern: { w: 3 } })



Observation and comparison:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Write Concern** | **Description** | **Observed Time (ms)** | **Durability** | **Comments** |
| w:1 | Acknowledged after the **primary** node writes the data. | **3 ms** | Low | Fastest write; if the primary fails before replication, data might be lost. |
| w:"majority" | Waits for acknowledgment from **a majority** (2 of 3) of replica set members. | **5 ms** | Medium–High | Slightly slower but ensures data survives one node failure. |
| w:3 | Requires acknowledgment from **all** 3 nodes before confirming the write. | **4 ms** | Highest | Most durable; ensures full consistency but may be slower in large networks. |

The experiment shows that increasing the write concern generally increases durability and fault tolerance, at the cost of slightly higher latency. In our local 3-node Docker cluster, the differences in latency were minimal (3–5 ms) because all replicas are on the same host. In a real distributed setup, w:1 would provide the lowest latency but weakest durability, while w:majority and w:3 would improve fault tolerance with more noticeable latency overhead.

1. Leader–Follower (Primary–Backup) Model

Step 1: Verify Current Primary and Secondaries

Run rs.status in the shell. Here we will see one node with "stateStr": "PRIMARY" and two nodes with "stateStr": "SECONDARY"



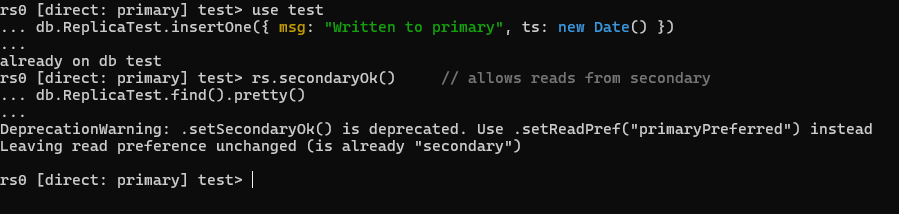
Step 2 — Demonstrate Write to Primary

Insert a document to the primary node:

use test  
db.ReplicaTest.insertOne({ msg: "Written to primary", ts: new Date() })

then check on secondary with the following code

rs.secondaryOk() // allows reads from secondary  
db.ReplicaTest.find().pretty()



Step 3: simulating primary failure

To demonstrate leader-follower failover, we first stopped the original primary node (mongo1) using Docker. The replica set quickly detected the failure, and one of the remaining secondary nodes (mongo2) was automatically elected as the new primary. This election process took a few seconds, representing a brief failover period. During this time, the cluster remained available, and once the new primary was elected, writes could continue without issue. The original primary can later rejoin the cluster as a secondary, ensuring high availability and data consistency across the replica set.

**Original Primary Stopped**

docker stop mongo1

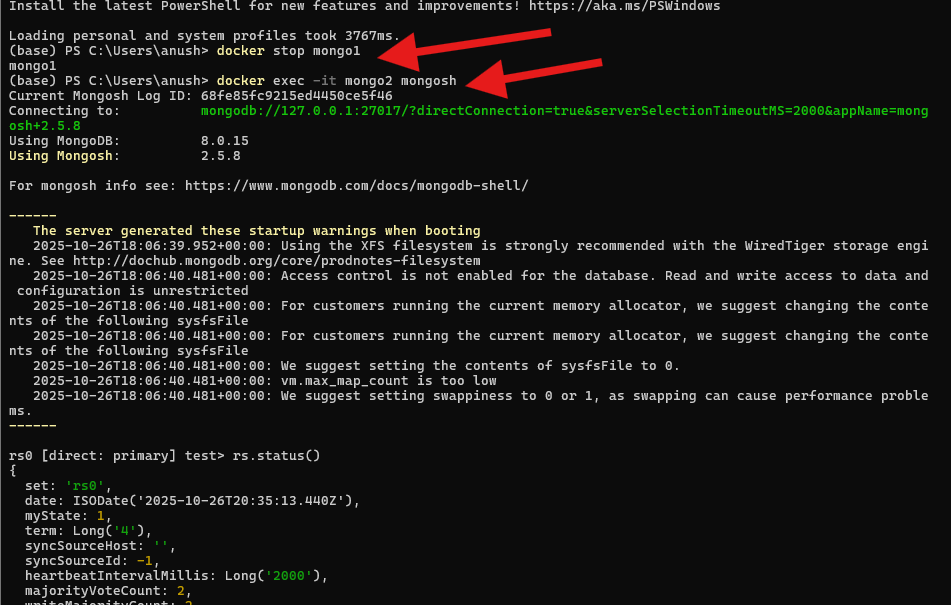
* mongo1 went offline → "stateStr": '(not reachable/healthy)'.
* Cluster detected the primary failure.

**New Primary Elected**

* mongo2 automatically became **PRIMARY** ("stateStr": 'PRIMARY').
* Election term incremented (term: 4).
* Failover completed in seconds — no manual intervention needed.

**Secondary Status**

* mongo3 remained **SECONDARY**.
* Still replicating from the new primary (syncSourceHost: 'mongo2:27017').



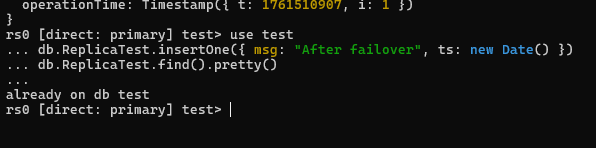
Step 4: Write after failover

Using this code

use test

db.ReplicaTest.insertOne({ msg: "After failover", ts: new Date() })

db.ReplicaTest.find().pretty()



This shows that message is Successfully inserted into the new primary. Data can now be read from the primary and eventually from secondaries.

1. Leaderless (Multi-Primary) Model (if applicable):

* A leaderless or multi-primary setup is not fully demonstrable in MongoDB as the database is based on a primary secondary (leader-follower) replication pattern. Under this model, all the writes go to the main node and the secondary nodes replicate the information asynchronously. Nevertheless, the construct of a leaderless system may be described through the example of databases, such as Cassandra.
* In these systems, any node is allowed to receive write and no leader is present and thus there is increased write availability. Writes are replicated to other nodes asynchronously and can conflict with each other in case two or more writes access the same data at the same time. The conflict resolution is usually managed with the help of such mechanisms as timestamps (last-write-wins), vector clocks, or application-level logic.
* All nodes eventually reach the same state, and eventually they become consist and yet there can be temporary inconsistencies. MongoDB, on the contrary, offers high consistency by having a leader-based replication, which provides that the data is instantly consistent but must pass through the primary node to write. This illustrates the sacrifice of short-term consistency in a leader-based system and greater availability of a leaderless system.

C. **Strong Consistency** in MongoDB

Step 1: Configure Write and Read Concerns

use test

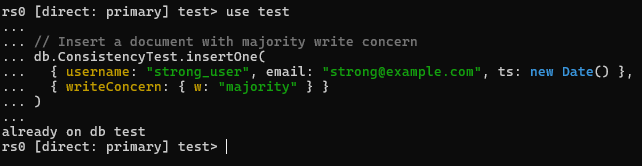
// Insert a document with majority write concern

db.ConsistencyTest.insertOne(

{ username: "strong\_user", email: "strong@example.com", ts: new Date() },

{ writeConcern: { w: "majority" } }

)



Step 2: Immediately read from a secondary using majority read concern

// Set read preference to secondary

db.getMongo().setReadPref('secondary')

// Read the document with readConcern "majority"

db.ConsistencyTest.find(

{ username: "strong\_user" }

).readConcern("majority").pretty()

Expected output:

{

"\_id": ObjectId("650f8f1c9c9b4b08e8ce5f49"),

"username": "strong\_user",

"email": "strong@example.com",

"ts": ISODate("2025-10-26T21:00:00Z")

}  
this confirms that the document written to the primary is immediately visible on the secondary

Step 3: Simulate a network partition / node failure

First stop one of the secondary node using **docker stop mongo3** command



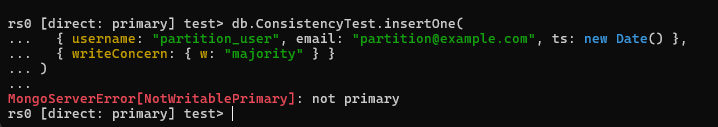
Next Try writing another document with majority write concern:

db.ConsistencyTest.insertOne(

{ username: "partition\_user", email: "partition@example.com", ts: new Date() },

{ writeConcern: { w: "majority" } }

)



Key obsevations for Part 1:

Part 1 has discussed strong consistency in a replica set in a MongoDB through the introduction of majority writes (w: "majority") and majority reads (readConcern: "majority"). We have placed a document in the primary node and it instantly showed in the secondary nodes, which proves that the data is consistent throughout the cluster. To test the impact of a network partition, we shut down the primary node and tried a second majority write which failed with an error of NotWritablePrimary. This also pointed out that, although strong consistency ensures that the data is accurate, it can temporarily reduce the availability in case of node failures or network partitions. These findings are consistent with the CAP theorem since MongoDB does accord more preference to consistency than to availability in the event of a partition. In general, the experiments proved that strong consistency guarantees the valid and up-to-date data in all nodes, however, write operations can be blocked during the events of partition or failover.

Part 2: Eventual Consistency in MongoDB

Now we switch back to test database and insert a document with minimal write concern using this:

db.EventualTest.insertOne(

{ username: "eventual\_user", email: "eventual@example.com", ts: new Date() },

{ writeConcern: { w: 1 } }

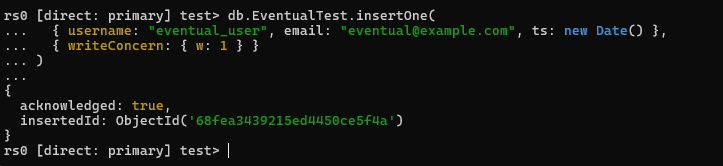
)

Note that we have to start the docker images again after the last steps using this command

>> (base) PS C:\Users\anush> docker start mongo1 mongo3

mongo1

mongo3

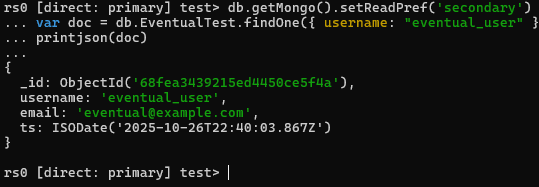


Step 3: Read from a secondary immediately

db.getMongo().setReadPref('secondary')

var doc = db.EventualTest.findOne({ username: "eventual\_user" })

printjson(doc)



Step 4: Loop until the latest value is observed

var doc = null

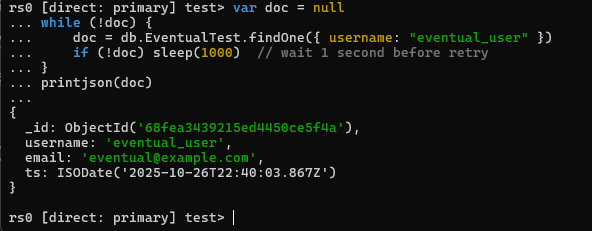
while (!doc) {

doc = db.EventualTest.findOne({ username: "eventual\_user" })

if (!doc) sleep(1000) // wait 1 second before retry

}

printjson(doc)



Key obervations:

This experiment on eventual consistency of MongoDB involved the insertion of a document into the EventualTest collection, with writeConcern: { w: 1 } that is, the write became acknowledged immediately it got to the primary, without undergoing secondary replication. The document contained a user name, email and time.

A direct read through a secondary node with setReadPref('secondary') to read the document showed that it was already in place, but when working with large clusters or in a network with latency it may at first be returning either a null or stale piece of data. To show the eventual nature, a loop was used to request the secondary several times until the document was displayed, which demonstrates intersections of replicas as time passes.

The following points are brought out in this experiment: reads by secondaries can be temporarily delayed, the process of replication is asynchronous, and the system can ensure that all nodes will eventually come into agreement. Eventual consistency is appropriate where the temporary staleness is tolerated in favor of increased availability and faster writes such as in social media feed, sensor, or dashboards.

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

Part 1: Difficulties of Distributed Transactions in MongoDB Situation.

1. Intercommunication among Multiple Nodes.

* A replica set is a set of secondaries that replicates the information that has been written on a single primary. Inter-collection or inter-cluster sharding distributed transactions entail inter-node coordination.
* Another similarity is that as observed in your experiments, when one of the primaries breaks (e.g., mongo1 dropped) the write operation cannot continue until a new primary is elected. This demonstrates that multi-node coordination may affect latency and interim unavailability.

1. Write Concerns Implications on Consistency and Latency.

* w: majority is also strongly consistent with the tradeoff that it is slower to write (the transaction exists until it has been replicated to a majority of nodes).
* w:1, on the other hand, can be used to write data faster, but will lose temporary replicas, making it harder to support multi-service processes when collections are not updated in sync.

1. Failure and Rollback Management.

* When one of the nodes fails during a transaction, an ACID transaction is forced to rollback or abort updates in more than one collection.
* The fact that your failover tests showed that operations under primary unavailability are not reliable (MongoServerError[NotWritablePrimary]) shows that distributed transactions are susceptible to node failures.

1. Sophistication of Compensating Actions (Saga Pattern)

* The use of Sagas means that compensating actions should be designed in each step. As an example, in case the deduction of inventory fails once an order is created, the order should be canceled.
* The experiment on your eventual consistency demonstrates that replicas can be out of step in the short run, and therefore compensation actions should include the propagation delays between nodes.

1. Trade-offs between Consistency and Availability.

* High consistency (majority writes + readConcern majority) may cause operations to be blocked in case of network partitions, as is the case with the initial implementation by secondary reads being unaware of fresh writes.
* Eventual consistency is better than availability and requires the design to address the conflicts and ensure the workflow is correct.

1. Complexity in Operation of Shared clusters.

* Although your experiments were done with a 3 node control group, there are a number of issues with scaling to a sharded cluster routing transactions, atomicity across shards, and multi-shard failures.

References:

<https://learn.microsoft.com/en-us/azure/architecture/patterns/saga?>

<https://microservices.io/patterns/data/saga.html?>

[https://microservices.io/patterns/data/saga.html](https://microservices.io/patterns/data/saga.html?)

Part 2: ACID Transactions

With a distributed e-commerce workflow that comprises OrderService, PaymentService and InventoryService, an ACID transaction would come to encompass all operations in one global transaction. As an example, the process of making an order, paying and reserving inventory would all happen in a single atomic transaction. This is very much needed to keep a high level of consistency though it is very much problematic in a distributed system. The ACID transactions provided by MongoDB can only be restricted to a replica set and can not be provided across multiple services and geographical locations. Orchestration of global transaction across services adds very much complexity, more latency, and may cause service unavailability by the failure of any of the services. Some of the challenges can be solved using distributed locks or two phase commit protocols but at the expense of performance and scalability. Therefore, ACID transactions are only consistent, but not useful in scalable multi-service environments.

Part 3: Sagas (Orchestrated or Choreographed)

Saga pattern breaks down the working process into local transactions within each service with compensating actions in case of failure. A key orchestrator is in charge of the sequence in an orchestrated saga the OrderService plans an order, the PaymentService handles payment and the InventoryService reserves items. In the event that one of the steps fails, the orchestrator can cause countermeasures like rescinding the payment or order. Services in a choreographed saga respond to events in other services, e.g. OrderCreated responds to PaymentProcessed, which responds to InventoryReserved. Compensating events are created because of failures, and services are able to deal with them on their own. The Saga approach makes the availability more accommodating, gracefully manages the faults, and allows distributed workflows without blocking all the services, but introduces some complexity in the coordination and event tracking. It has eventual consistency as opposed to strict ACID guarantees.

Part 4: Trade-Offs

A comparison of the ACID transactions and Sagas presents some important trade-offs. ACID transactions are very consistent yet difficult to apply to distributed multi-service systems because they have coordination overhead, they may have a high level of latency and they are also not very tolerant to failures. Sagas however offer greater availability, fault tolerance and performance as every service runs independently and Local failures are handled. Nonetheless, Sagas trade short-term consistency with long-term consistency, necessitating a significant design of the compensating actions and event processing. The Sagas are more charged with practical use in the case of distributed e-commerce workflows, allowing scalable, robust systems, and correcting business processes with compensations.

Conclusion:

To concluse everything in this lab, the NoSQL concepts were illustrated with the help of MongoDB. Part A included installation of a replica set and simple data operations. Part B discussed replication strategies, leader-follower failover and leaderless conflict resolution. Part C depicted great and ultimate consistency as it showed replication lag and convergence. Part D conceptually studied distributed transactions and contrasted ACID issues with the Saga pattern of multi-service workflow. In general, the lab demonstrated trade-offs in consistency, availability, fault tolerance, and performance of distributed systems.