UV Distributed Information Management
Summer semester 2023

Hands-On: MongoDB Replication in Action

This document contains the instructions to replicate the hands-on exercise **MongoDB Replication in Action**.

Remark: This hands-on exercise has only been tested on Debian Linux (but probably works in a similar manner on other systems).

Formatting Conventions Commands for the Linux command-line tool (`terminal`) and the command-line tool of MongoDB (`mongosh`) written in `TrueType` font[1]. In addition, all commands are in a box that specifies the used command-line tool at the beginning of the title (separated by a dash –, i.e., **`terminal`** for Linux and **`mongosh`** for MongoDB). Listing 1 shows an example command executed in the Linux `terminal`:

Listing 1: **`terminal`** – Show directories.

```
dbtutorial@database-tutorial:~# ls -l
```

Contrarily, Listing 2 exemplifies a command in MongoDB's command-line tool, i.e., `mongosh`, which stands for *Mongo shell*. It shows an example that executes the command `show dbs` on a database named `test`:

Listing 2: **`mongosh`** – Show all databases.

```
test> show dbs
```

For more details on the terminals, we kindly refer to the description of Assignment 2[2].

1 Setup

In order to replicate this hands-on exercise, we need to simulate n nodes that are part of a cluster[3]. In our hands-on exercise, we assume $n = 3$ nodes, but this should also work for $n > 3$. For simplicity, we create a cluster with three nodes locally on a single physical machine (i.e., our workstation, laptop, ...). To this end, we need three directories, each of which represents a node in our cluster[4]. When we start MongoDB in replication mode, we tell MongoDB to use

[1] `TrueType` font: `https://en.wikipedia.org/wiki/TrueType`
[2] Assignment 2: `https://dbresearch.uni-salzburg.at/teaching/2023ss/dim/assignment2.pdf`
[3] In practice, this is a set of machines that are connect via network, e.g., Ethernet.
[4] No need to establish connections, since they all reside on the same physical machine.

these three directories to store the data that resides on the respective node. Listing 3 shows how to create these three directories inside the `home` directory of your user. Line 1 switches to the `home` directory of your user and we list all directories that exist in this directory (line 2). Then, we create a new directory named `mongo` and navigate into this newly created directory (lines 4–5). Finally, we create three subdirectories named `replica1`, `replica2`, and `replica3`, which mimic the data that is stored on node 1, node 2, and node 3, respectively (lines 7–9), and we validate that these directories exist (line 11).

Listing 3: **`terminal`** – Set up 3 directories (in home) that represent 3 nodes in our cluster.

```
1  dbtutorial@database-tutorial:~# cd /home/<username>
2  dbtutorial@database-tutorial:~# ls -lah
3
4  dbtutorial@database-tutorial:~# mkdir mongo
5  dbtutorial@database-tutorial:~# cd mongo
6
7  dbtutorial@database-tutorial:~# mkdir replica1
8  dbtutorial@database-tutorial:~# mkdir replica2
9  dbtutorial@database-tutorial:~# mkdir replica3
10
11 dbtutorial@database-tutorial:~# ls -lah
```

Subsequently, we need six to seven terminal instances (i.e., processes) simultaneously: (i) Three terminals (1–3) are used to simulate a cluster of three nodes (each terminal mimics a node in our cluster). (ii) Three terminals (4–6) are used to connect to the nodes that run the respective MongoDB daemons. (iii) Optionally, a separate terminal to import the data into our cluster using the `mongoimport` command-line tool. Table 1 lists the names of the respective terminals and a mapping from terminal to node or functionality.

Terminal	**Node / Functionality**
`node N1`	Runs the first `mongod` instance on port `42000` in terminal 1.
`node N2`	Runs the second `mongod` instance on port `42001` in terminal 2.
`node N3`	Runs the third `mongod` instance on port `42002` in terminal 3.
`mongosh 1`	Runs the first `mongosh` in terminal 4 (connected to `node N1`, i.e., port `42000`).
`mongosh 2`	Runs the second `mongosh` in terminal 5 (connected to `node N2`, i.e., port `42001`).
`mongosh 3`	Runs the third `mongosh` in terminal 6 (connected to `node N3`, i.e., port `42002`).

Table 1: Terminal-to-node/functionality mapping.

Remark: On most systems, you can change the title of the terminal (which is shown on top). This is particularly useful if you have many terminal instances like in this exercise. For a better overview, change the titles of the respective terminals to match the names given in Table 1.

2 Replicate the Hands-On Exercise

In order to replicate the hands-on exercise, we have to (1) import some data into our MongoDB cluster with three nodes, (2) connect the `mongosh` terminals to our cluster, (3) insert a new document and verify that it is replicated in our cluster, and (4) kill the `PRIMARY` node of our cluster and confirm that our cluster is still operating correctly.

2.1 Data Import

Before we can import the data, we must start the three MongoDB nodes in replication mode and initiate the replication (i.e., tell the nodes that the data should be replicated to the other nodes). Furthermore, we must also specify that each of our nodes writes its data to a different location (to simulate that the three nodes are indeed separated from a storage point of view). Therefore, we specify three options when starting the nodes: (1) We use the so-called *replica set* option, `–replSet`, to specify that our three nodes are part of a common replication (i.e., MongoDB servers that are part of the same replication set take care of the same data by replicating it). (2) We specify the location of the database, `–dbpath`, such that each terminal (i.e., process) maintains its own data (effectively mimicing three separate nodes). (3) Each node runs on a different port, `–port`[5]. Listings 4–6 show how to start the MongoDB daemons that represent our 3 nodes with the corresponding parameters (the `dbtutorial@database-tutorial` prefixes are omitted to fit each command into a single line).

Listing 4: **terminal** – Start a Mongo daemon as part of replica set `ReplicationTest` on node 1 (port: `42000`, subdirectory: `mongo/replica1`).

```
1 :~# mongod --replSet ReplicationTest --dbpath="/home/<username>/mongo/replica1" --port 42000
```

Listing 5: **terminal** – Start a Mongo daemon as part of replica set `ReplicationTest` on node 2 (port: `42001`, subdirectory: `mongo/replica2`).

```
1 :~# mongod --replSet ReplicationTest --dbpath="/home/<username>/mongo/replica2" --port 42001
```

Listing 6: **terminal** – Start a Mongo daemon as part of replica set `ReplicationTest` on node 3 (port: `42002`, subdirectory: `mongo/replica3`).

```
1 :~# mongod --replSet ReplicationTest --dbpath="/home/<username>/mongo/replica3" --port 42002
```

Next, we need to initialize the replica set, i.e., (a) we connect to `node N1` with one `mongosh` terminal (`mongosh 1`), (b) initiate the replication, and (c) add the other nodes to the replica set. Therefore, we first connect to `node N1` using the `mongosh` terminal as shown in Listing 7.

Listing 7: **terminal** – Connect to `node N1`.

```
1 dbtutorial@database-tutorial:~# mongosh -port 42000
```

Once connected, we refer to this specific terminal as `mongosh 1`, and we can use the command shown in Listing 8 to check the status of our replica set (denoted `rs`). MongoDB will report an output that is similar to the output that is shown at the end of Listing 8 (lines 2–7).

Listing 8: **mongosh 1** – Check the status of our replica set (rs).

```
1 test> rs.status()
2 {
3 "ok" : 0,
4 "errmsg" : "no replset config has been received",
5 "code" : 94,
6 "codeName" : "NotYetInitialized"
7 }
```

[5]Port: `https://en.wikipedia.org/wiki/Port_(computer_networking)`

The `rs.status()` command can be used any time in between the subsequent command to chck the current status of our replica set. MongoDB reports that *no replset config has been received*, i.e., the replication set has not yet been initialized. Naturally, the next step is to initialize the replica set (at `node N1`) *and* to add the other nodes to this replica set. Listing 9 shows how to accomplish this.

Listing 9: **`mongosh 1`** – Initialize our replica set (rs) and add the other nodes (`node N2/N3`).

```
test> rs.initiate()

test> rs.add("localhost:42001")
test> rs.add("localhost:42002")

test> rs.status()
```

Afterwards, the `rs.status()` command should provide more output and the key "`members`" should map to a list that contains three entries: `localhost:42000` (i.e., `node N1`) as `PRIMARY`, and `localhost:42001/2` (i.e., `node N2/N3`) as `SECONDARY`. This confirms that the replica set has been successfully initiated and we can now continue with the actual data import.

For a general description on how to import JSON data into a MongoDB database, we refer to the description of Assignment 2[6] and the MongoDB documentation[7]. The command in Listing 10 imports the (relatively small) arXiv collection[8] into the database of `node N1` (i.e., it is stored in the corresponding directory `mongo/replica1`; identified by the port in the connection string "`42000`"). The backslash "\" is used to have a multi-line command in the Linux terminal.

Listing 10: **`terminal`** – Import the plain arXiv JSON files into `node N1` of our replica set.

```
dbtutorial@database-tutorial:~# mongoimport "mongodb://localhost:42000" \
  --db replicationTest --collection arxiv --file <path-to-file>/arxiv.json
```

2.2 Connect the `mongosh` Terminals

After the arXiv JSON file has been successfully imported, our replica set ensures that the data is replicated from `node N1` (on which we imported the data) to `node N2` and `N3` (which are part of our replica set), respectively. We can verify this by opening two additional `mongosh` instances that connect to the respective nodes. To connect to a particular node using the `mongosh` command-line tool, we must use the `–port` option as shown in Listings 11 and 12, respectively.

Listing 11: **`terminal`** – Connect to `node N2`.

```
dbtutorial@database-tutorial:~# mongo --port 42001
```

Listing 12: **`terminal`** – Connect to `node N3`.

```
dbtutorial@database-tutorial:~# mongo --port 42002
```

Remark: By now, we should have six terminals opened, three of which run the nodes of our cluster, i.e., nodes (`node N1` – `node N3`; showing debug output recurrently) and three of which

[6] Assignment 2: `https://dbresearch.uni-salzburg.at/teaching/2023ss/dim/assignment2.pdf`
[7] The `mongoimport` tool: `https://developer.mongodb.com/how-to/mongoimport-guide/`
[8] Data of Assignment 2: `https://kitten.cosy.sbg.ac.at/index.php/s/bMLHK7JfxjLGwok/download`

are mongosh terminal that are connected to the three nodes (mongosh 1 – mongosh 3; waiting for user input).

Now that we have the six terminals set up, we can start using the cluster. First, we verify that the data was indeed replicated to node N2 and node N3; cf. Listing 13. Line 1 queries the MongoDB instance for the available databases; in lines 2–3, we switch to a new database named replicationTest and show its collections (which should show one collection named arxiv); line 4 counts the documents of the arxiv collection, and MongoDB should report three documents in this collection.

Listing 13: **mongosh 1/2/3** – Verify that the data has been replicated to all nodes N1/N2/N3.

```
1 test> show dbs
2 test> use replicationTest
3 replicationTest> show collections
4 replicationTest> db.arxiv.find().count()
```

Remark: In case that one of the above commands results in the exception shown in Listing 14, please execute the command rs.secondaryOk()[9] to resolve it. This should work for MongoDB version 3.6.

Listing 14: **mongosh 1/2/3** – NotPrimaryNoSecondaryOk exception.

```
uncaught exception: Error: listDatabases failed:{
  "topologyVersion" : {
    "processId" : ObjectId("60b775579c140578afb0f4bd"),
    "counter" : NumberLong(4)
  },
  "operationTime" : Timestamp(1622636167, 1),
  "ok" : 0,
  "errmsg" : "not master and slaveOk=false",
  "code" : 13435,
  "codeName" : "NotPrimaryNoSecondaryOk",
  "$clusterTime" : {
    "clusterTime" : Timestamp(1622636167, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  }
}
```

2.3 Replication of a New Document

The replica set consists of two types of members: PRIMARY[10] and SECONDARY[11]. In fact, there is another type of node, a so-called *arbiter*[12], but we ignore this type for this exercise. In a nutshell, the replication in MongoDB[13] works as follows: There is a single PRIMARY node (node N1 in our case) and multiple SECONDARY nodes (node N2 and node N3 in our case). All write operations are served by the PRIMARY node, and the data is replicated to the SECONDARY nodes (specifically, the log files are replicated). The replication happens asynchronously.

[9] https://docs.mongodb.com/manual/reference/method/rs.secondaryOk/
[10] Replica Set Primary: https://docs.mongodb.com/manual/core/replica-set-primary/
[11] Replica Set Secondary Members: https://docs.mongodb.com/manual/core/replica-set-secondary/
[12] Replica Set Arbiter: https://docs.mongodb.com/manual/core/replica-set-arbiter/
[13] Replication: https://docs.mongodb.com/manual/replication/

Note that only the PRIMARY node can serve write operations, whereas read operations can be served by one of the SECONDARY nodes. If we assume this for now, we can only insert a new document on node N1, i.e., using mongosh 1 (since it is the PRIMARY that initiated the replica set; executing the same command on one of the SECONDARY nodes will result in a NotWritablePrimary exception). Before, however, we verify that none of our nodes contains this document (as of yet); cf. Listing 15.

Listing 15: **mongosh 1/2/3** – Check if the document is present on node N1/N2/N3.

```
replicationTest> db.arxiv.find({ "type": "Lehrveranstaltung" })
```

Once we verified this, we can insert the new document on node N1 as shown in Listing 16.

Listing 16: mongosh 1 – Insert a new document on node N1.

```
replicationTest> db.arxiv.insertOne({
          "name": "Verteiltes Informationsmanagement",
          "type": "Lehrveranstaltung"
        })

replicationTest> db.arxiv.find({ "type": "Lehrveranstaltung" })
```

Although we cannot insert a new document on our SECONDARY nodes, the data is automatically replicated from the PRIMARY node to the SECONDARY nodes, and we can verify this by searching for the very same document on our SECONDARY nodes; cf. Listing 17.

Listing 17: **mongosh 2/3** – Search for the newly inserted document on nodes N2/N3.

```
replicationTest> db.arxiv.find({ "type": "Lehrveranstaltung" })
```

Then, you should see the very same result as on node N1, and we conclude that MongoDB fully replicated the new document automatically and transparently to the user.

2.4 Killing the PRIMARY

If the PRIMARY node is unavailable (for whatever reason), one of the SECONDARY nodes holds an election to elect itself as the new PRIMARY[14]. Until a new PRIMARY is elected, the replica set cannot process write operations. However, read operations can still be served by the SECONDARY nodes (if configured appropriately).

In the final step of this hands-on exercise, we kill the PRIMARY node and see if and how MongoDB deals with it. Therefore, we simply terminate the mongod process that represents node N1 (using CTRL + C or by simply closing the terminal). This triggers the election mechanism of MongoDB and the remaining two nodes (i.e., the former SECONDARY nodes). If we now try to execute a query in mongosh 1, we will get an error message because node N1 is not available (it is down). Contrarily, the other two nodes N1 and N2 are still available and we can continue with mongo 2/3.

First, we can check which of the two nodes is the new PRIMARY node; cf. Listing 18.

[14]Replica Set Elections: https://docs.mongodb.com/manual/core/replica-set-elections

Listing 18: **mongosh 2/3** – Check the status of our replica set to find the new PRIMARY.

```
replicationTest> rs.status()
```

Regardless of outcome of the election process (i.e., the specific node that is the new PRIMARY node), we observe that our cluster is still available and functional.

Remark: In practice, we would like to connect to our cluster *transparently*, i.e., we do not connect to a specific node (e.g., node N1) but to the cluster. In other words, if one node goes offline, we do not want to update the connection or reconnect to our cluster. Instead, we want our connection to stay valid as long as the cluster is running (available). Listing 19 shows how to use mongosh to establish a connection transparently.

Listing 19: **terminal** – Connect to our cluster (i.e., replica set) transparently.

```
dbtutorial@database-tutorial:~# mongosh --host ReplicationTest/localhost:42000
```

Although we still connect to node N1 (using port 42000), we get a transparent connection by specifying the name of the replica set ReplicationTest in the connection string explicitly. If we now kill the PRIMARY node, the election algorithm determines a new PRIMARY and we can continue to use this connection without reconnection.

We conclude this hands-on exercise with another option for a transparent connection that uses all three nodes in the connection string (Linux user prefix omitted due to space constraints); cf. Listing 20.

Listing 20: **terminal** – Connect to our cluster (i.e., replica set) transparently (variant 2).

```
:~# mongosh "mongodb://localhost:42000,localhost:42001,localhost:42002/?replicaSet=ReplicationTest"
```