UV Distributed Information Management
Summer term 2021

Hands-On: MongoDB Replication in Action

This document contains the individual steps that are required 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).

Commands and queries are wrapped in a framed listing environment which also specifies the used command-line tool at the beginning of the title (separated by a dash –). The following listing shows an example that executes the command `ls` in a Linux `terminal` (explanatory comments are shown in gray and are not part of the command):

`terminal` – Show directories.

```
ls    # this is a comment (i.e., not part of the command)
```

1 General Setup

In order to replicate this hands-on exercise, we need to simulate n nodes that are part of our cluster [1]. In our hands-on exercise, we assume $n = 3$ nodes, but this should also work for $n > 3$. For our simulation, we create a cluster with 3 nodes locally on a single physical machine (i.e., our workstation, laptop, ...). To accomplish this, we need 3 folders, each of which represents a node in our cluster [2]. When we start MongoDB in replication mode, we simply tell MongoDB to use these 3 folders to store the data that resides on the respective node. The following listing shows how to create these 3 folders inside the `home` folder of your user:

`terminal` – Set up 3 directories (in home) that represent the 3 nodes of our cluster.

```
cd /home/<username>   # switch to home directory
ls -lah               # show directories

mkdir mongo           # create a new directory mongo
cd mongo              # switch to mongo directory

mkdir replica1        # create subdirectory mongo/replica1 (representing node 1)
mkdir replica2        # create subdirectory mongo/replica2 (representing node 2)
mkdir replica3        # create subdirectory mongo/replica3 (representing node 3)

ls -lah               # show directories
```

Subsequently, we may need multiple terminal instances (i.e., processes) simultaneously since each terminal simulates a single node. We will use these names to refer to the respective

[1] In practice, this is a set of machines that are connect via network, e.g., Ethernet.
[2] No need to establish connections, since they all reside on the same physical machine.

terminals, and the following table shows a mapping of terminal to node or functionality, respectively:

Terminal	Node / Functionality
`N1`	Node 1; runs the first `mongod` in the background.
`N2`	Node 2; runs the second `mongod` in the background.
`N3`	Node 3; runs the third `mongod` in the background.
`mongo1`	Runs the first `mongo` terminal (connected to node 1, i.e., port 42000).
`mongo2`	Runs the second `mongo` terminal (connected to node 2, i.e., port 42001).
`mongo3`	Runs the third `mongo` terminal (connected to node 3, i.e., port 42002).

Tabelle 1: Terminal-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 terminal names given in Table 1.

2 Replicate the Hands-On Exercise

2.1 Data Import

Before we can import the data, we must start the 3 MongoDB nodes in replication mode and initiate our 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 3 nodes are indeed separated from a storage point of view). Therefore, we specify three options when we start the nodes: (1) We use the so-called *replica set* option, `--replSet`, to specify that our 3 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 simulating 3 nodes). (3) Each node runs on a different port, `--port`. The following three listings show how to start the Mongo daemons on our 3 nodes with the corresponding parameters:

`N1` – Start a Mongo daemon as part of replica set `ReplicationTest` on node 1 (port: `42000`, subdirectory: `mongo/replica1`).

```
# The backslash "\" is used to have a multi-line command in bash.
mongod --replSet ReplicationTest \
  --dbpath="/home/<username>/mongo/replica1" \
  --port 42000
```

`N2` – Start a Mongo daemon as part of replica set `ReplicationTest` on node 2 (port: `42001`, subdirectory: `mongo/replica2`).

```
# The backslash "\" is used to have a multi-line command in bash.
mongod --replSet ReplicationTest \
  --dbpath="/home/<username>/mongo/replica2" \
  --port 42001
```

N3 – Start a Mongo daemon as part of replica set ReplicationTest on node 3 (port: 42002, subdirectory: mongo/replica3).

```
# The backslash "\" is used to have a multi-line command in bash.
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 1 with the mongo terminal, (b) initiate the replication, (c) add the other nodes to the replica set. Therefore, we first connect to node 1 using the mongo terminal:

terminal – Connect to node 1.

```
# Node 1 is identified by the port in the connection string (42000).
# Subsequently, this terminal is referred to as ‘‘mongo1’’ (cf. Table 1).
mongo --port 42000
```

Once we are in the mongo terminal, we can use the following command to check the status of our replica set (denoted rs) and should see the following output:

mongo1 – Check the status of our replica set (rs).

```
rs.status()

# The following output should be shown, denoting that the replica set is yet to
# be initialized.
# {
#   "ok" : 0,
#   "errmsg" : "no replset config has been received",
#   "code" : 94,
#   "codeName" : "NotYetInitialized"
# }
```

The rs.status() command can used in between all the steps we show next to always see the current status of the replica set. Naturally, the next step is to initiate the replica set (at node 1) *and* to add the other nodes to the replica set that is initialized at node 1. This can be done as follows:

mongo1 – Check the status of our replica set (rs).

```
rs.initiate() # initiate replica set at node 1

rs.add("localhost:42001") # add node 2 (identified by port 42001)
rs.add("localhost:42002") # add node 3 (identified by port 42002)

rs.status()
```

Afterwards, the rs.status() command should provide more output and the key "members" should map to a list that contains 3 entries: localhost:42000 (i.e., node 1) as PRIMARY, and localhost:42001/2 (i.e., node 2/3) as SECONDARY. This confirms that the replica set has been initiated successfully and we can 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 [3] and the MongoDB documentation [4]. The following command

[3] https://dbresearch.uni-salzburg.at/teaching/2021ss/dim/assignment2.pdf
[4] https://developer.mongodb.com/how-to/mongoimport-guide/

imports the (relatively small) arXiv collection [5] into the database of node 1 (i.e., it is stored in `mongo/replica1`):

`terminal` – Import the plain arXiv JSON files into node 1 of our replica set.

```
# The backslash "\" is used to have a multi-line command in bash.
# Node 1 is identified by the port in the connection string (42000).
mongoimport "mongodb://localhost:42000" \
  --db replicationTest
  --collection arxiv
  --file <path-to-file>/arxiv.json
```

2.2 Connect the `mongo` Terminals

After the arXiv JSON file has been imported successfully, our replica set ensures that the data is replicated from node 1 (on which we imported the data) to node 2 and 3 (which are part of the replica set), respectively. We can verify this by logging into the respective nodes. To connect to a particular node using the `mongo` command-line tool, we must use the `--port` option as follows:

`terminal` – Connect to node 2.

```
# Node 2 is identified by the port in the connection string (42001).
# Subsequently, this terminal is referred to as ‘‘mongo2’’ (cf. Table 1).
mongo --port 42001
```

`terminal` – Connect to node 3.

```
# Node 3 is identified by the port in the connection string (42002).
# Subsequently, this terminal is referred to as ‘‘mongo3’’ (cf. Table 1).
mongo --port 42002
```

Remark: By now, we should have six terminals opened, three of which run the nodes of our clustering (`N1` – `N3`; showing debug output recurrently) and three of which are `mongo` terminal that are connected to the three nodes (`mongo1` – `mongo3`; 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 nodes 2 and 3:

`mongo1/2/3` – Verify that the data is replicated on all three nodes.

```
# This should show the 3 default databases (admin, config, local) and our
# database (replicationTest).
show dbs

# Switch to our "replicationTest" database.
use replicationTest

# This should show 1 collection "arxiv".
show collections

# This should show 3 entries in the arXiv collection.
db.arxiv.find().count()
```

[5] Downloadable from our Nextcloud: `https://kitten.cosy.sbg.ac.at/index.php/s/4gdSoq5rFCb57Xw`

Remark: In case that one of the above commands results in the following error, please execute the `rs.secondaryOk()` [6] command to resolve it.

`mongo1/2/3` – `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)
#     }
#   }
# }
```

3 Replication of a New Document

The replica set consists of two types of members: `PRIMARY` [7] and `SECONDARY` [8]. In fact, there is another type of node, a so-called *arbiter* [9], but we ignore this type for this exercise. In a nutshell, the replication in MongoDB [10] works as follows: There is a single `PRIMARY` node (`N1` in our case) and multiple `SECONDARY` nodes (`N2` and `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.

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 1, i.e., using `mongo1` (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):

`mongo1/2/3` – Check if the document is present on node 1/2/3.

```
# Verify that our arXiv collection does not contain a document of type
# "Lehrveranstaltung" on node 1/2/3.
db.arxiv.find({ "type": "Lehrveranstaltung" })
```

[6] https://docs.mongodb.com/manual/reference/method/rs.secondaryOk/
[7] https://docs.mongodb.com/manual/core/replica-set-primary/
[8] https://docs.mongodb.com/manual/core/replica-set-secondary/
[9] https://docs.mongodb.com/manual/core/replica-set-arbiter/
[10] https://docs.mongodb.com/manual/replication/

Once we verified this, we can insert the new document on node 1 as follows:

mongo1 – Insert a new document on node 1.

```
# Insert a new document of type "Lehrveranstaltung" with name
# "Verteiles Informationsmanagement" into the arXiv collection.
db.arxiv.insertOne({
  "name": "Verteiltes Informationsmanagement",
  "type": "Lehrveranstaltung"
})

# Look up the newly inserted document.
db.arxiv.find({ "type": "Lehrveranstaltung" })
```

Although we cannot insert a new document on our SECONDARY nodes, the data is automatically being 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:

mongo2/3 – Search for the newly inserted document on node 2/3.

```
# Verify that the new document has been replicated onto nodes 2 resp. 3.
db.arxiv.find({ "type": "Lehrveranstaltung" })

# You should see the very same result as on node 1.
```

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 [11]. 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 and see if and how MongoDB deals with it. Therefore, we simply terminate the mongod process that represents N1 (using CTRL + C or just closing the terminal). This triggers the election mechanism of MongoDB and the remaining two nodes (i.e., the previous SECONDARY nodes). If we now try to execute a query in mongo1, we will get an error message because node 1 is not available (it is down). Contrariliy, the other two nodes are still available and we can continue with mongo2/3.

First, we can check which of the two nodes is the new PRIMARY node:

mongo2/3 – Check the status of our replica set to find the new PRIMARY.

```
rs.status()
```

Regardless of which nodes is the new PRIMARY node, we observe that our cluster is still available and can be used.

Remark: In practice, we would like to connect to our cluster *transparently*, i.e., we do not connect to a specific node but to the cluster. In other words, if one node becomes unavailable, 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). This can be accomplished

[11] https://docs.mongodb.com/manual/core/replica-set-elections

by adapting the connection options as follows:

`terminal` – Connect to our cluster (i.e., replica set) transparently.

```
1  mongo --host ReplicationTest/localhost:42000 # Our new mongo1 terminal.
```

Although we still connect to node 1 (i.e., port 42000), we get a transparent connection by specifying the name of the replica set (`ReplicationTest`) explicitly. If we kill the `PRIMARY` node, the election algorithm determines a new `PRIMARY` and we can continue to use this connection without reconnecting.