# TDDE41 Lab2, Topic related project

Daniel Wassing (danwa223)
Robin Andersson (roban591)

May 2019

## 1 Research Question

From our testing with the ZooKeeper software and our own dummy code we concluded that it would be interesting to see how well ZooKeeper can perform its duties when subjected to large scale distributed systems. We want to find out whether or not ZooKeeper is robust with respect to race conditions. We concluded that we would test this by having large amounts of threads that create clients and modify the same data at the same time. Thus, our research question is based on finding out if ZooKeeper is robust when deploying a varying amount of clients and servers.

## 2 Setup

Our setup is the same as the one for lab 1. We improved it by using a bash script to start 7 ZooKeeper servers at once.

In order to achieve a more realistic scenario, ZooKeeper runs in replicated mode (meaning it has an ensemble of several servers set up) during these tests. For the testing, we still use IntelliJ IDEA. From IDEA we specify (within the code) how many threads (each thread is one client) we want to run. The clients are load balanced over the servers (meaning they all initially connect with an even spread) to give ZooKeeper a "fair" chance of handling the clients.

## 3 Method

Here we describe our way of apprehending the research question.

In order to find out whether or not ZooKeeper can avoid race conditions when deploying many clients and having them modify data, we:

- Defined a race condition to be the modification of data in the targeted znode by more than one client per test run.

- Asked all clients to modify the same znode.
- Had all clients connect evenly across the ZooKeeper ensemble (available servers).
- Ran many test runs (50 per setup) to account for unexpected behavior.
- Assume that if client A writes to one server, while client B writes to another server only one of them actually modifies the data.

After we ran all the setups 50 times, we saved the collected test data which is shown in this report further down, as histograms.

# 4 Implementation

In this section, we will try to describe some key implementations that we have done to apprehend our research question.

ZKConnection, ZKNodes, ZKMultithreading and Main are the 4 classes that we have either modified or created from scratch. ZKConnection was taken from the tutorialspoint page on ZooKeeper [1]. Nothing major was edited here. ZKNodes was implemented via guidelines and concrete examples from tutorialspoint that described some key elements of the ZooKeeper API. ZKNodes offers a lot of modules and functionality that we use, such as creating, deleting, and checking the status of znodes (for example, whether they exist or not). We also performed some slight updates towards the data access, to directly send strings instead of byte data. ZKMultithreading basically contains the functionality to run a thread. For every thread, ZKMultithreading runs it and saves the run result, which is either a success or failure to modify a znode. A failure is noticed by ZooKeeper throwing an exception (called KeeperError). A success (meaning modification of data) is noticed by the absence of exceptions.

The Main file is in charge of the testing. Here we modify our parameters such as how many threads we want to execute on each server and how many servers that we initially connect to. For every thread we capture the result and after all threads have been executed we output the collective results on the screen. These results are our test data, which we elaborate on further.

# 5 Results

Each test setup was run 50 times to accommodate for any irregularities that may occur. We chose to test up to 7 servers because more servers would just seem ridiculous since the laptop's hardware that we ran the tests on is not top-of-the-line. For the same reason, and for measurement's sake, we only chose to test with 8 and 16 clients per server in all cases. The test setups that we ran are shown in table 1 below.

| Test setup number | Number of servers | Number of clients per server |
|---|---|---|
| 1 | 3 | 8 |
| 2 | 3 | 16 |
| 3 | 5 | 8 |
| 4 | 5 | 16 |
| 5 | 7 | 8 |
| 6 | 7 | 16 |

Table 1: The test setups that were used.

Figures 1-3 show histograms of each of the tests that we ran. The horizontal axis shows how many clients that were allowed to update the znode while the vertical axis shows how many times that many clients were allowed to update the znode in total during our 50 tests.

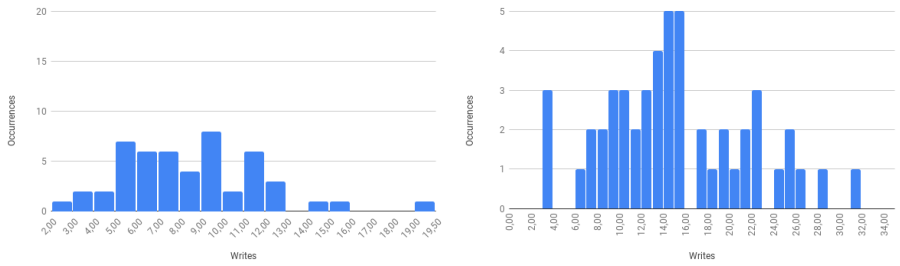


Figure 1: 3 servers running with 8 (left) and 16 (right) clients each.

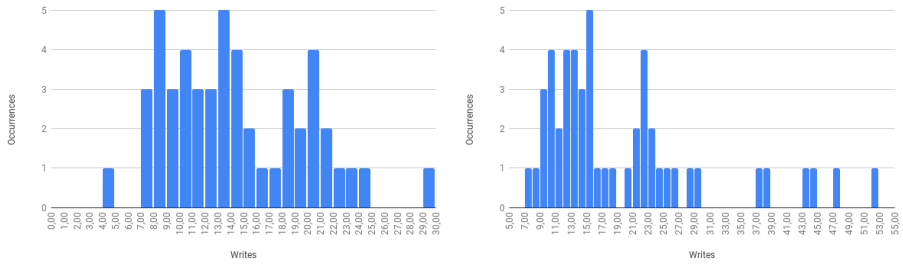


Figure 2: 5 servers running with 8 (left) and 16 (right) clients each.

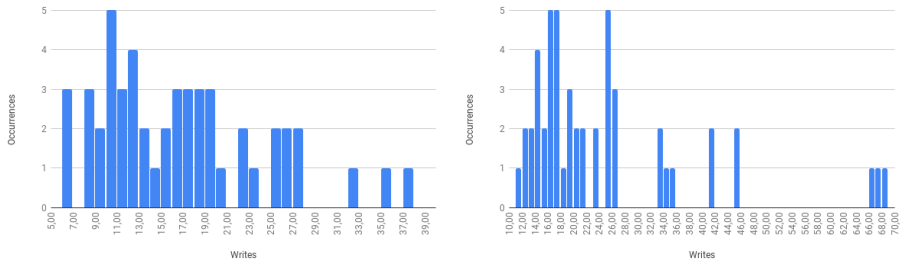


Figure 3: 7 servers running with 8 (left) and 16 (right) clients each.

# 6 Discussion

As seen in the results, when we chose to deploy from 3 to 7 servers with 8 or 16 clients each, ZooKeeper seems to be nowhere near robust when it is run on a single laptop.

However, we cannot put all the blame on ZooKeeper since the laptop's hardware is not particularly impressive, and the operating system's CPU scheduler might allow one client to modify the znode and finish executing before other clients even make an attempt at modifying it. Meaning, some threads (clients) finish executing before others even get a chance to start executing their own tasks. If this happens, then ZooKeeper does per definition work correctly since it should allow new modifications of the znode data, and is therefore not to blame for the perceived race conditions. We cannot detect whether the scheduler causes these problems or not. We do not believe that it does, but we cannot say for certain.

# References

[1] Tutorialspoint. Zookeeper - quick guide. `https://www.tutorialspoint.com/zookeeper/zookeeper_quick_guide.htm`. Accessed 2019-05-14.