School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2025

Programming Assignment 3: ZooKeeper & Distributed Computing

Due date: November 27 @ Midnight, extension without penalty until 30-November.

In this programming assignment, you will develop a simple distributed computing platform following a classical load-balancing architecture. There is a load-balancer/coordinator process, called "manager", and worker processes. The high-level objective of the platform is that a client can submit a "task/job" for computation to the platform, which will execute the task/job using one of its many workers and provide the client with the result. In principle, the manager gets execution requests from clients, distributes them across the workers and returns the results back to the clients. But the manager does not exchange messages with the clients nor the workers. Instead, they use ZooKeeper for coordination. That is, the different components in the setup (clients that submit tasks, manager the coordinates and workers that execute the tasks) are all clients to Zookeeper.

You are provided with a template code with some of the functionalities already implemented. For the purpose of this assignment, we will not consider crashes and failovers of the manager and worker processes, nor the clients, and you can assume that all processes gracefully terminate.

Please ensure that you familiarized yourself with the ZookeeperSetup-Distributed documentation and set up your ZooKeeper ensemble, and done some sanity checks on it before starting the work on the deliverable. Some useful references are given at the end of this document. If you have any additional questions, please use the Ed discussion forum for programming assignment 3.

Onetime setup of ZK nodes

Using the ZooKeeper client CLI (`zkCli.sh`), create a permanent znode `/distXX` where `XX` is your group number (e.g. group 4 would create `/dist04`). Also create permanent znodes `/distXX/tasks` and `/distXX/workers`. You may need to create additional permanent znodes as needed based on your final design.

Setting up the Template

The `zk.tar` file provided with this description already has a substantial amount of code for your project. You can untar this into some directory in one of the SOCS servers.

```
zk
|-- clnt
|   |-- compileclnt.sh
|   |-- DistClient.java
|    -- runclnt.sh
|-- dist
|   |-- compilesrvr.sh
```

```
|   |-- DistProcess.java
|    -- runsrvr.sh
 -- task
    |-- compiletask.sh
    |-- DistTask.java
     -- MCPi.java
```

- Edit the `*.java` and `*.sh` files to replace `XX` with your group number where present. Also remember to replace the ZK server names and client ports with what you had used for your ZK ensemble setup.
- Compile the `task`, `clnt`, `dist` directories in that order. Remember to set your `ZOOBINDIR` environment variable and execute the env script before compiling (see the setup documentation for this).

The Current Application

- The `task` directory contains the definitions of "Jobs" classes that the client programs can construct an object and (serialize) submit to the distributed platform through the ZooKeeper. For the purpose of this project, we have a Monte Carlo based computation of the value of `pi` defined in `MCPi.java`.
- The `clnt` directory contains the client java program (`DistClient.java`) that creates the "job" object and submits it to the platform, awaits the result and retrieves it. It does this by creating a sequential znode `/distXX/tasks/task-` that contains the "job" object (serialized) as the data and then waiting for a `result` znode to be created under it.
- You can invoke the client in the following way (make sure that the `ZOOBINDIR` is set and the env script is executed.)

  ```
  $ ./runclnt.sh 500000
  ```

  Where `500000` is the number of samples (points) that the Monte Carlo simulation should use (the larger the number, the more computation it needs to perform, but the more accurate the computed value of `pi` would be).
- Ideally, you should not have to edit any of the java files in `task` or `clnt` directories other than changing XX to your group number and for any additional debugging that you think you may need.
- The `dist` directory contains `DistProcess.java` that defines the program code that all the members in the distributed platform have. This is where most of your work would be.
- A member process can be started as follows.

```
$ ./runsrvr.sh
```

The general idea is that you start each member process from a different machine, and therefore will have several machines in your distributed platform.

- In the current implementation, if you start a `DistProcess` it tries to become the manager by creating an ephemeral znode `/distXX/manager`. A second process will detect that there is already a manager, but does nothing else as of now. That is, in this default implementation, you have a manager that will also execute the jobs requested by the clients, while the other member processes do nothing.
- The manager process listens for any child znodes under `/distXX/tasks`. It picks up these znodes, unserializes the data to build the "job" object and performs the "computation". It then serializes the job object and writes it (creates) back to `result` znode that is the child of the task znode. The manager is also currently doing this inside its callback function, which was we discussed in the class is not the best place as it blocks the ZK client library.
- In the current application the manager/worker terminates after waiting for a few seconds. In the modified application, make sure they are always up and running. (You can use the Unix `kill` command to terminate the JVM of your application if you want to shut it down - but try not to use `kill -9` unless the regular `kill` is not working). Make design choices that will reduce the amount of "cleanup" in the ZooKeeper to a minimal.

Question 1: Implementation (80 Points)

1. The first change is to make sure that any additional `DisProcess` that you start after you have a manager will become a "worker". You will have to figure out an extension to the current znode hierarchy (the ZK model) used by the application to accomplish this.
2. The manager should be enhanced to keep track of the workers in the platform (as new workers join).
3. The manager, instead of doing the computations itself (current setup) will assign the jobs to an idle worker.
4. A worker will take care of executing only one job at a given point in time. After it is done with its computation, it will go back to being idle. You will have to figure out a way of keeping track of who is idle/busy using ZK so that the manager can manage the assignment of jobs to the workers effectively.
5. If there are no idle workers but there are more jobs, the manager will have to wait until a worker becomes free to assign the next job or a new worker joins the platform (you must handle that case too). Ideally we would like to prioritize the job that was added first, but we do not have to be strict about this.

6. Finally, you need to introduce a time slicing approach. We do not want to prevent a short running task from executing when all the workers are busy with long running tasks. So the idea is that when a task arrives, a worker only executes it for a predetermined time (the time slice). If the execution time is smaller or equal to the time slice, the task is completed. Otherwise, the worker saves the current state and lets the manager know that the worker is back to idle. The remainder of the task is then rescheduled. From an implementation point of view you need to interrupt the worker thread after a time slice has expired. When the task executing gets notified about interruption, it stops at its earliest convenience, saves the in-memory state and becomes idle. You have to decide on the length of the time slice. Maybe try a few values and find one that works well. Below is some information about interruption (with links to relevant documentation):

    - Threads can be interrupted by calling the `interrupt()` method.
    - The task interface in the starter code throws the `InterruptedException`. When the thread executing a task gets interrupted, it consumes the interrupt flag by calling `interrupted()` class method. The task will clean up and save its internal state before throwing the `InterruptedException`.
    - When a task is interrupted, it saves its current state in memory, but not on Zookeeper. It is your task to make sure you save the intermediate state of a task so it can pick up where it left when it gets assigned to another worker.

7. There are some `TODO` labels throughout `DistProcesss.java` which should help you navigate to some of the locations where you need to do the above code changes. However, keep in mind that depending on your design and approach, you might have add extra code/functions.

A few things to keep in mind.

- All communication between the member processes of the distributed computing platform, including the clients of the platform will be through the ZooKeeper ensemble. I.e., by creating/removing znodes, updating the data component of the znode, etc. **There should not be any other form of communication between them such as RMI, network sockets, files, etc. (Which will result in a 0 for the project!)**
- Depending on your design, you may have to create some other additional znodes, etc., over and above the general outline given above to accomplish this project.
- Make sure to put plenty of messages (print statements) throughout the various important steps of the code so that we can observe what each process is doing just by observing their display messages. Some examples are already present in the code given to you. It should clearly demonstrate what the master did, what the worker did, etc.
- The ZooKeeper model needs to be simple, just do the task that is asked in the project description. Put an emphasis on minimizing the number of messaging overhead (between the ZooKeeper servers and the manager/workers/clients), number of znodes required in the model, etc. DO

NOT include unnecessary elements in your architecture (including the model) to handle things like failures. Points will be deducted for designing a model that contains elements not necessary for the project description or creating a model that is significantly more complex than needed (the latter is often a sign of unfamiliarity of how to use the watches and callbacks properly).

- Your system architecture should not contain any ongoing "polling" mechanism (minimum 30 points reduction). For example, continuously checking if there is a new task, idle worker, etc. Instead, you should rely on the concept of callbacks and watches to efficiently tackle the communication and get notified anytime an event happens that maybe potentially interesting to your process.

Question 2: Report and Demo (20 Points)

Report (10 Points)

The report should contain max. 2 pages of text (Times New Roman, 11pt, single-column). Again, figures can take additional space. The report mostly describes the ZooKeeper model that you designed (similar to the tree diagram in class). There should be a diagram of the model included. Explanation as to what each znode is for (including the type of the znodes), who does what, when, where.

As last section, your report must contain a detailed description of the contributions of each of the group members. That is, for each of the tasks, indicate how much and what each of the team members has contributed. Furthermore, describe your collaboration efforts. For that, you can, e.g., include the dates and durations of each of the meetings, and/or the number of emails / Slack discussion entries / messages that you had in order to come up with the overall solution. Each team member is expected to contribute significantly to the project work. If we notice a pattern of certain members not contributing consistently, they may receive a lower grade letter at the end of the course.

The report is due on Thursday, 27th November, at midnight.

Demo (10 Points)

To grade your implementation we will use live demonstrations where you show your running system to us. You can have a slide or 2 if you need that to describe your model.

The demonstrations of all groups will take place the days around the due date. A sign-up sheet will be put in place so that you can reserve a time-slot. Show up early and have your system running, using a different set of machines for the ZK ensemble and for your client and DistProcess servers (You can run multiple clients from the same machine if you would like to). We would like to see at least 3 workers, 4 (or more) client jobs being submitted at the same time. Expect questions!

All code is due on MyCourses by the time of your demo.

Assumptions

- You do not have to worry about a worker having to switch to manager when the manager shuts down.
- It is assumed that the platform is shut down (i.e. kill all the processes) only when there are no pending client jobs.
- You can assume that a worker shuts down only when the entire platform is shutdown.
- Remember, a new worker can join the platform at any time (including after some client jobs have already been submitted). You must be able to use this worker.

Some Useful Documentation/References

1. ZooKeeper TextBook (Oreilly) Chapters 2, 3. If you glance over some of the sections, they discuss the nuances of load balancer -worker(s) implementations (We are not building such a complicated setup though).
2. ZooKeeper Java API doc. `https://zookeeper.apache.org/doc/r3.8.5/`
3. ZooKeeper main APIs (create, get, etc.,). `https://zookeeper.apache.org/doc/r3.8.5/apidocs/zookeeper-server/org/apache/zookeeper/ZooKeeper.html`
4. Link to All ZK packages and classes `https://zookeeper.apache.org/doc/r3.8.5/apidocs/zookeeper-server/index.html`
5. AsyncCallback interfaces (to receive asynchronous callback) `https://zookeeper.apache.org/doc/r3.8.5/apidocs/zookeeper-server/org/apache/zookeeper/AsyncCallback.html`
6. Watcher Related `https://zookeeper.apache.org/doc/r3.8.5/apidocs/zookeeper-server/org/apache/zookeeper/Watcher.html`
   `https://zookeeper.apache.org/doc/r3.8.5/apidocs/zookeeper-server/org/apache/zookeeper/WatchedEvent.html`

Copyright