Zachary Siciliani (260845463)
Eric Pelletier (260895863)
December 1st, 2022
COMP 512

Project 3: ZooKeeper and Distributed Computing

For our ZooKeeper implementation, we maintained a very simple znode structure. Under our dist35 node, we created two permanent nodes: tasks and workers. During the execution of our program, we create ephemeral sequential znodes that are children of workers. There is also one ephemeral node that goes directly under dist35 to be the master. Meanwhile, the client is responsible for adding child znodes to tasks when it has a job that needs to be done. The znodes created by the client are permanent sequential. While earlier versions of our design also had a third permanent node, called assign, this was removed in our final implementation to make the overall system simpler. A possible configuration of our ZooKeeper system can be seen in Figure 1.

```
dist35 (permanent)
|--master (ephemeral)
|--workers (permanent)
|   |--worker-0000000000 (ephemeral sequential)
|    --worker-0000000001 (ephemeral sequential)
|--tasks (permanent)
    |--task-0000000000 (permanent sequential)
    |--task-0000000001 (permanent sequential)
     --task-0000000002 (permanent sequential)
```

Figure 1: ZNode structure of our ZooKeeper setup

First, a new server that connects to the system needs to determine whether it's the master or a worker. To do so, it attempts to create the master ephemeral node. If that node does not yet exist, then this process becomes the master. If it already exists, it knows there is already a master and it becomes a worker instead. It then creates an ephemeral sequential worker node, and sets a watch on its own node.

The master is responsible for maintaining a list of idle workers as well as a list of tasks. When it first connects to the system, it will check to see if there are any permanent task children that were already stored as nodes. If so, it will add them to the queue and it will add the related data to a matching task data queue. It then waits for workers to join by maintaining a watch on the workers node. Once a worker joins, it will set a watch on the worker's data to know if that specific worker is idle or actively executing a job.

When the master receives a WatchedEvent notification, it starts by checking what type of event it was. If the type was NodeChildrenChanged, it knows that a new child znode was added to one of the znodes it previously had a watch on. It then checks the path to determine whether this was a new worker or a new task. Whichever it was, it will reset the appropriate

watch and then get the children of that node using an asynchronous call. When the callback method processResult of that asynchronous call runs, it will add the new child to the relevant queue or list and will get the data from the node asynchronously. When getting task data, the master will check whether there are any idle workers in its local list. Similarly, when getting worker data, it will check whether there are any tasks in the queue. If there is no element in the opposite list, it will add the task to the queue or the worker to the idle list. On the other hand, if there is now both an idle worker and a task to be assigned, then the master will set the task as the data at the worker node. It does so by writing the taskId and the taskData to an outputStream and storing the byte array of this output stream at the worker node with a setData call. It will pick the first task in the queue and the first worker in the list, which should be the ones that have been waiting the longest. It will also remove the task from the queue and the worker from the idle workers list as they are no longer available.

The worker, meanwhile, will have a watch set up on its own znode. If the worker receives a WatchedEvent, it will check to make sure that the event type is NodeDataChanged. It does so by creating a ByteArrayInputStream, along with two empty byte arrays; one for the task ID and one for the task data. Since the task ID is known to be exactly 15 characters long, with format “task-XXXXXXXXXX”, it can be read first. Then, an ObjectInputStream can be used to convert the data bytes to an ObjectInput, which is then read and cast to a DistTask. If so, it knows that a new task has been assigned to it by the Master. It will then call the asynchronous getData method. In the callback function of the worker, it will launch a separate thread with the data of the job. It will also reset the watch on its node. This thread will then execute the job using the provided DistTask class and reserialize it. It can then write this completed job result to the task node data, using the same process as described above. From here, the client can read the node of the task that it has created and find the result there. Finally, the worker will write “unassigned” to the data of its own node to show that it has completed the job.

This will yet again trigger the worker’s watch on its own node. By seeing that the data has been changed to be equal to “unassigned”, it knows it has nothing left to do. Additionally, the master will also receive a WatchEvent for this change. By seeing that the data has been returned to “unassigned”, it knows the worker is yet again idle. It can now add the worker back to its list of idle workers. If there are still tasks available, it can assign it to the worker right away. If not, it will wait for a new task to come in.

Work Distribution:
Eric was responsible for management of idle workers and available tasks and assignment of tasks to workers. Zachary was responsible for asynchronous data calls and creating the threading for computing. Both team members performed debugging and helped with the report.