Cloud Computing Systems 2016 - Apache ZooKeeper

Implementation of fault-tolerance techniques in a distributed setting based on master/worker architecture.

Authors

- Reto Schiegg
- David Wettstein

Git repository

- https://github.com/dwettstein/zk

Deployment

ID	Owner	Group	Name	Status	Host	IPs
7415	r.schiegg1	ccs16	SchieggWettstein_ZooKeeper_3	RUNNING	newcluster-33	172.16.2.122
7413	r.schiegg1	ccs16	SchieggWettstein_ZooKeeper_2	RUNNING	newcluster-19	172.16.2.121
7412	r.schiegg1	ccs16	SchieggWettstein_ZooKeeper_1	RUNNING	newcluster-26	172.16.2.120

Description of the project

Setup

Used technologies/resources

- Apache ZooKeeper
- Python 2.7
- Kazoo library
- zk-shell

Running ZooKeeper

Use *ssh* for connecting to the appropriate VM, *cd* to the project's repository and start the Python scripts in the following order:

```
ubuntu@ubuntu:~/zk$ . reset_zookeeper.sh
ubuntu@ubuntu:~/zk$ python master.py
ubuntu@ubuntu:~/zk$ python worker.py
ubuntu@ubuntu:~/zk$ python client.py
ubuntu@ubuntu:~/zk$ zk-shell localhost     # see what's happening
```

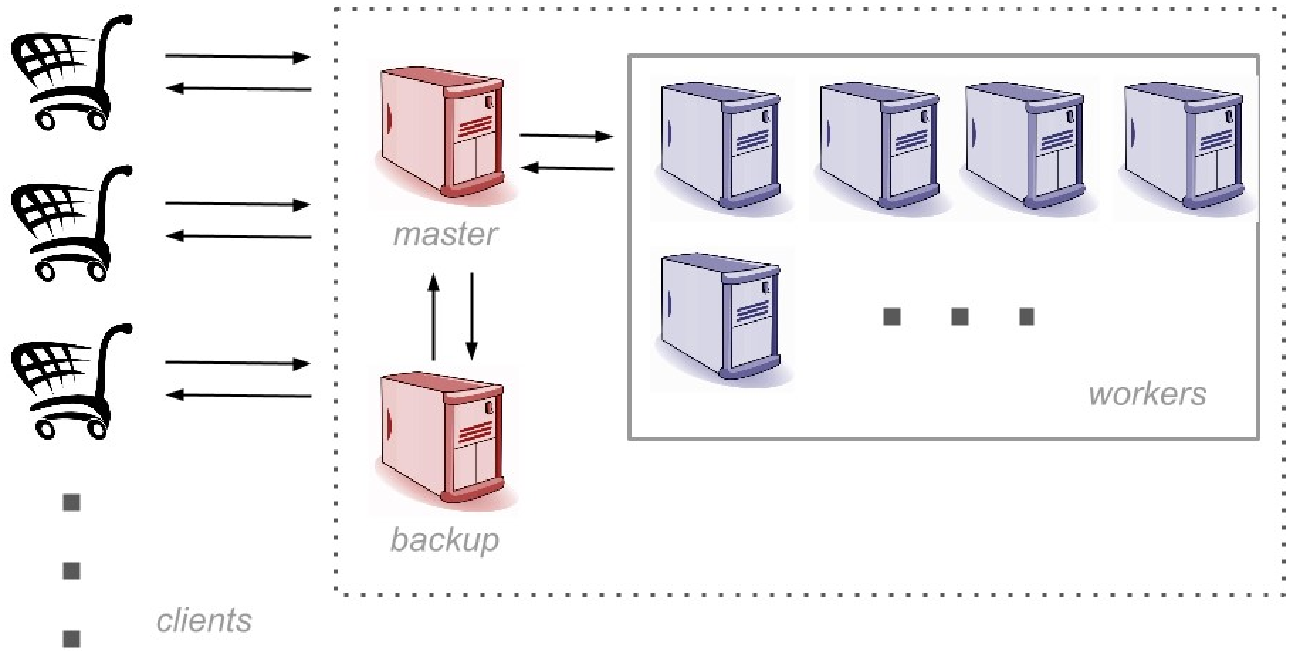
To remove/kill a master or a worker:

```
ubuntu@ubuntu:~/zk$ ps aux
ubuntu@ubuntu:~/zk$ kill {master.py_process_id | worker.py_process_id}
```

Apache ZooKeeper

Apache ZooKeeper is an open source project with the aim of providing distributed synchronization and configuration. It allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers (znodes). These registers are kept in-memory, from which it follows that the size of the database, managed by ZooKeeper, is limited by memory. On the other hand, the main advantage is a very high throughput and low latency numbers. Thanks to this, it can also be used within large distributed systems.

Master/Worker architecture

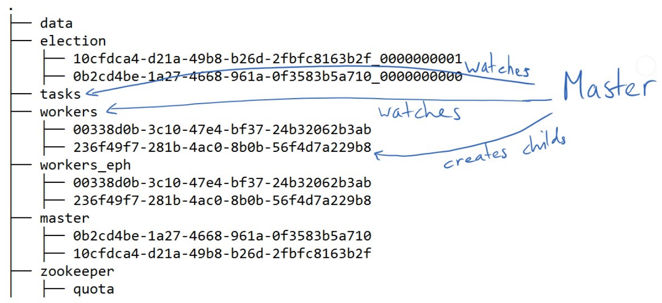


The idea of the master/worker architecture is to split the responsibilities of serving a client request. The master is responsible to assign these requests to an available worker. Then, this worker processes the actual request. Since there is only one master at once (i.e. the current leader), we need some backup instances of it in order to guarantee the reliability of the system.

Responsibilities and watchers

Master:

- Watches the `/tasks` and `/workers` nodes
- Gets unassigned tasks and assign those to workers with respect to load-balancing

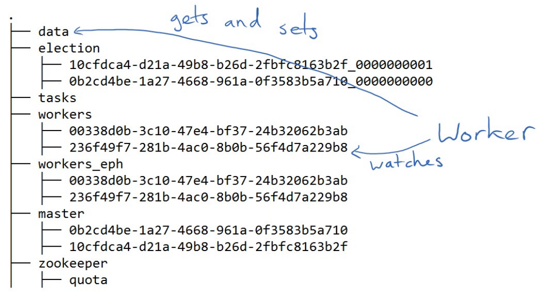


```
.
├── data
├── election
│   ├── 10cfdca4-d21a-49b8-b26d-2fbfc8163b2f_0000000001
│   ├── 0b2cd4be-1a27-4668-961a-0f3583b5a710_0000000000
├── tasks
├── workers
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
├── workers_eph
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
├── master
│   ├── 0b2cd4be-1a27-4668-961a-0f3583b5a710
│   ├── 10cfdca4-d21a-49b8-b26d-2fbfc8163b2f
├── zookeeper
│   ├── quota
```

Worker:

- Watches for child-nodes under itself (i.e. assigned tasks)

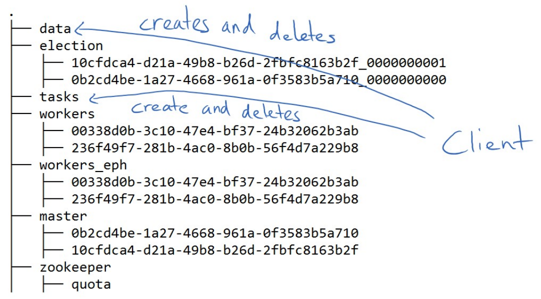
- Gets the data from `/data` node and executes assigned tasks



```
.
├── data
├── election
│   ├── 10cfdca4-d21a-49b8-b26d-2fbfc8163b2f_0000000001
│   ├── 0b2cd4be-1a27-4668-961a-0f3583b5a710_0000000000
├── tasks
├── workers
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
├── workers_eph
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
├── master
│   ├── 0b2cd4be-1a27-4668-961a-0f3583b5a710
│   ├── 10cfdca4-d21a-49b8-b26d-2fbfc8163b2f
├── zookeeper
│   ├── quota
```

Client:

- Creates new task child-nodes and watches for task completion
- Removes the according child-nodes in `/tasks` and `/data` nodes after a task result has been received



```
.
├── data
├── election
│   ├── 10cfdca4-d21a-49b8-b26d-2fbfc8163b2f_0000000001
│   ├── 0b2cd4be-1a27-4668-961a-0f3583b5a710_0000000000
├── tasks
├── workers
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
├── workers_eph
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
├── master
│   ├── 0b2cd4be-1a27-4668-961a-0f3583b5a710
│   ├── 10cfdca4-d21a-49b8-b26d-2fbfc8163b2f
├── zookeeper
│   ├── quota
```

Leader election

The idea of the leader election is to find a new master/leader after the previous master has been discontinued or disconnected. In our system, we implemented the leader election according to this example: ZooKeeper Election Example.

The main concept is that each possible master gets a number according to the node creation order. The lowest existing node is the current master/leader. When the current master dies, the next master with lowest number takes over the responsibilites.

In order to avoid a herd effect when the current master dies (i.e. every backup asks if it is the new master at once), every backup master sets its watch to the preceeding (next lower) backup master only.

In our implementation, we have a `/master` and `/election` node. When a master is initialized, it creates a child-node with the same GUID in both nodes. Additionally, the child-node in the `/election` node uses the *ephemeral* and *sequence* flags. When the election node of the current master disappears, the succeeding master will receive an event and takes over. If a backup master dies, we update the watchers such that every backup master always watches the next lower master.

Load-balancing

For assigning and executing the client requests/tasks, we use some kind of load-balancing algorithm such that the workload is evenly distributed over the workers. Whenever a worker gets a task, it creates a child-node under itself in order to remember what he has to do. When a new task arrives, the current master/leader assigns it to the worker with the least task assignments at this time. The result can be seen in the following *tree* log of *zk-shell*:

```
.
├── tasks
│   ├── ec334a07-e6aa-4d80-8142-22cfd832b3d3
│   ├── 8d57784e-7968-4961-91a2-3b15f943570f
│   ├── 6ffb7f69-6b15-42be-9e24-18e391906cac
│   ├── 4f5e8ca0-406e-4a31-b036-56ff08cc7146
│   ├── 56d4d85e-da81-4aed-8a48-e9553b2aed92
│   ├── f9d3d07e-5d36-4ac9-b719-821061c5e74c
├── workers
│   ├── 00338d0b-3c10-47e4-bf37-24b32062b3ab
│   │   ├── ec334a07-e6aa-4d80-8142-22cfd832b3d3
│   │   ├── 8d57784e-7968-4961-91a2-3b15f943570f
│   │   ├── 56d4d85e-da81-4aed-8a48-e9553b2aed92
│   ├── 236f49f7-281b-4ac0-8b0b-56f4d7a229b8
│   │   ├── 6ffb7f69-6b15-42be-9e24-18e391906cac
│   │   ├── 4f5e8ca0-406e-4a31-b036-56ff08cc7146
│   │   ├── f9d3d07e-5d36-4ac9-b719-821061c5e74c
```

Fault-tolerance

In this chapter, we will discuss the different fault-tolerance scenarios. In the more complex scenarios, a log file is provided in order to proove the correctness of our algorithm. A log file is separated by the following: zk-shell tree, master, worker, client.

c1w1m1 - A worker or client fails

If a client fail, no more tasks will be created and a new client could be started straight forward. The only side effect is, that executed tasks will not be deleted, if the result is available. However, this side effect does not conflict with functions of the system. In addition, this can be easily fixed by just iterating once over all data and look, if the result is already stored in the value of the znode. If a single worker fails, no more tasks will be executed and all tasks remain unassigned. After starting a new worker instance, the unassigned tasks will be assigned to the new worker. Therefore, there will be no further issues.

c1w2m1 - A worker fails

Log: c1w2m1 - A worker fails

After starting a client, a master and two workers, we waited until both workers had more than 3 tasks assigned. Due to our load-balancing the tasks were equally split. By sending a kill command to the worker process, one ephemeral worker node was deleted. As the master watches on changes in the "/workers_eph" node, he recognized that 4 tasks were assigned to a worker that does not exist anymore. Consequentely, the master reassigned the 4 tasks to the other worker. This behavior is more than apparent by the output of the master, where is says that 4 tasks were reassigned. In addition, the zookeeper tree indicates well that tasks were assigned to both workers, a worker got deleted and all tasks were executed (no more tasks nodes in the tree after stopping the client).

c2w2m1 - Workers compete in executing the tasks

Log: c2w2m1 - Workers compete in executing the tasks

First, we started two workers, one master and two clients. Due to our load-balancing implemented in the master, each worker gets the same amount of tasks assigned. This is cearly visible in the master log section, where the assignment of a task is almost alternately by the worker. As we store the assigned tasks underneath a workers node, the workers do not conflict in executing the tasks. After stopping the client is is clear, that all tasks were executed as no more tasks are in the zookeeper tree.

c2w2m2 - The backup resumes the job of the master upon a failure

Log: c2w2m2 - A master fails

After starting two master instance, two workers and two clients, the master node election was successfully exectued as the node with the lower sequencial number (0000000000) was initially selected as the master. Next, we waited until the master assigned a couple of tasks to both workers. Then, we exectued the kill command with the master's process-id in order to delete the master. This is more than apparent in the zookeeper tree where the election node with the number (0000000000) and the master node was deleted. In addition, it is also clearly visible in the master log section, where it says "New elected master" with the new master node id (previos the backup). To test, if still further tasks were assigned, we continued the execution of the system. In all section of the log, you can examine that the client still generates tasks, the master still assigns tasks and the worker still executes tasks.

Cluster mode

When zookeeper is in cluster mode, the zookeeper tree is shared with all servers. Therefore, zookeeper synchronizes all modifications made in the tree between the servers. As the configuration of such a cluster is done within minutes, it is very convenient to scale such a system. We tested our system by running the two instances of clients, workers and masters. However, an instance of a type (client, worker, master) is only on one server. Therefore, if a server is fails, there still exists at least an instance of each type. As the master-election nodes and the worker nodes are denoted as ephemeral, they will be deleted when the corresponding thread dies resp. when the server crashes. As a result, the master/worker system would detect the change and act accordingly.

Example: If the server with a worker instance crashes, the corresponding ephemeral node will be deleted. As the master watches on "/workers_eph", the master will detect the change and will search for tasks which are assigned to the deleted worker. Due to the worker instance on another server, there still are workers and the master can start the re-assignment accordingly to the c1w2m1-case in the fault-tolerance chapter.

Summary

Zookeeper is a very helpful system for developing scaleable applications with lots of distributed coordination. Firstly, it takes some effort to understand the complexity of the master/worker system as well as the different functionalities of zookeeper. However, after working with zookeeper a bit, you get the comprehension of working with the zookeeper tree, its znodes and the watcher-callback-functions. In addition, we were surprised how simple it is to configure a zookeeper in cluster mode. Consequentely, zookeeper is a big support in terms of implementing scaleable distributed systems.