University of Camerino

School of Science and Technologies

Course Master Degree In Computer Science (LM-18)

Distributed Systems

ZooKeeper Leader Election: Algorithm and Real Case Scenario

Student

Luca Bianchi

Supervisor

Prof. Michele Loreti

A.A. 2023/2024

Contents

List of Code

List of Figures

1. Introduction

In this project, I would like to talk about ZooKeeper[1] and one of its features: the **leader election algorithm**.

Personally, among all the different projects, I decided to work on ZooKeeper for several reasons:

- It is an argument we both encountered during the course of Distributed Systems and Technologies for Big Data Management, therefore, at least once, I wanted to see for myself how it works and if it was worth it.
- Distributed Systems are now one of the main topics in today's computer science world, thus using a tool that is able to coordinate and synchronize such systems is fundamental and crucial.
- Among the other different possibilities, this was the only theoretical project accompanied by its practical implementation, and many times I think that only by practically experiencing something we can really understand it.

For such reasons, the project is centered around ZooKeeper.

In particular, in the first chapter, I will first describe what ZooKeeper is, how it is structured, and how it works. Furthermore, I will focus on Kazoo, a Python library to interact with ZooKeeper, and on the leader election process in a distributed system. In fact, ZooKeeper offers many features, but the one that I was the most interested in, is how to use ZooKeeper to elect a leader among the nodes in the distributed system it coordinates. This is one of the things in which Apache ZooKeeper excels.

In the second chapter, I will explain the solution itself and how to execute it.

Lastly, I will disclose the conclusions I had the chance to apprehend during this project.

Regarding the implemented solution, I decided to work with Python. The point is that there is no actual constraint on the language used to implement the solution, the important part is to choose a language that supports the interaction with ZooKeeper,

[1]ZooKeeper documentation website: `https://zookeeper.apache.org/doc/current/index.html`

and Python, through the Kazoo library, offers such a possibility. Furthermore, given the last university projects I have worked on, I am much more experienced with such a language. Lastly, I wanted to recreate an environment in which someday I would be able to actually create a real case of a distributed system implemented in Python (e.g. working with a distributed application implemented with Apache Spark) and coordinate it using ZooKeeper.

The complete program, which will be explained later on, can be found in my repository on **GitHub**[2].

[2]GitHub repository: `https://github.com/BianchiLuca28/ZooKeeper-LeaderElection`

2. Apache Zookeeper: Leader Election Algorithm

In this chapter, I will start by introducing ZooKeeper's structure[1]. After which, I will move to Kazoo and the implementation of the leader election algorithm.

Apache ZooKeeper is a distributed, open-source coordination service for distributed applications. It is a service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It is a crucial component in distributed systems, offering a simple and high-performance coordination service. Also, the service runs in Java and has some bindings for both Java and C.

Apache ZooKeeper became a standard for organized service used by Hadoop, HBase, and other distributed frameworks. For example, Apache HBase uses ZooKeeper to track the status of distributed data.

The overall idea behind the service was to relieve distributed applications of the responsibility of implementing coordination services from scratch since this could be very difficult and complicated to do manually. In fact, there are many things that we need to take into account when dealing with such problems, therefore, a service like ZooKeeper aims to reduce the possibilities of error by taking care of everything.

2.1 ZooKeeper's Architecture and Design

As already mentioned, ZooKeeper is a distributed service. This is the first distinction we need to make, in fact, this isn't related to the distributed application.

From the theory, we know that a distributed system is composed of several nodes which interact with each other to implement a service. Each of the nodes will serve as a server to the clients that will contact the distributed system to make use of the provided service.

In ZooKeeper, the service is as well organized as a distributed system itself. As we can see in Figure 2.1, on the right, there is the structure of the ZooKeeper service. The

[1]ZooKeeper structure website: `https://zookeeper.apache.org/doc/current/zookeeperStarted.html`

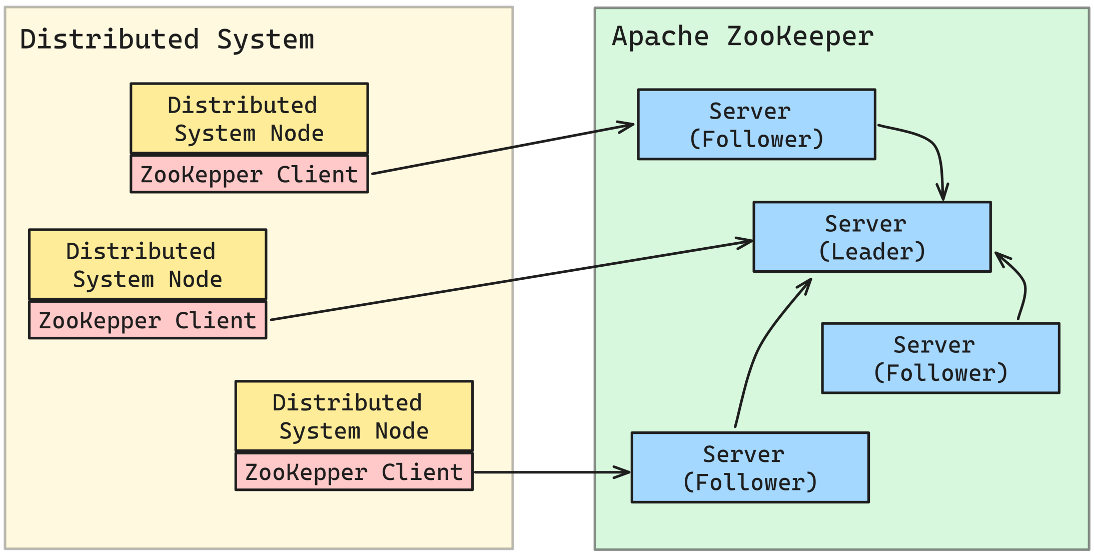


Figure 2.1: ZooKeeper's Architecture

overall structure of the service is called **ensemble** since it can either be composed of several nodes, as in the figure, or of only one node. We can think of using a single node in the development phase, in which we don't need to consider robustness and fault-tolerance, while in production, it is better to structure the service over multiple nodes (ideally the documentation talks about at least 3 nodes and an odd number of nodes to avoid problems in consensus protocols) to deal with fault-tolerance problems in which the single server goes down.

On the left, we can see an example of a distributed system composed of some nodes. What is really important to understand is that, even though the nodes in the distributed system act as servers towards the clients requiring the services of the distributed system, they also act as clients from ZooKeeper's perspective. It is important to make this distinction otherwise it is difficult to understand the structure of ZooKeeper itself.

To actually enable the nodes in the distributed system to connect to the nodes in the ZooKeeper ensemble, each of the nodes will have to instantiate a ZooKeeper Client, which will connect to one of the nodes of ZooKeeper by specifying the host and the port as in normal client-server communication.

In the ZooKeeper ensemble, in case there are multiple nodes, only one of them will serve as the leader, while all the others will be followers. Later on, I will introduce the leader election algorithm, but it is crucial to also understand that this algorithm isn't related at all to the leader of the ZooKeeper ensemble. The 2 leaders (the one in the ensemble and the elected leader in the coordinated distributed system) are completely different. The algorithm will also work even in the case there is only one node in the ZooKeeper ensemble. We won't cover how ZooKeeper manages internally the leader election of its nodes, but the one in the coordinated distributed.

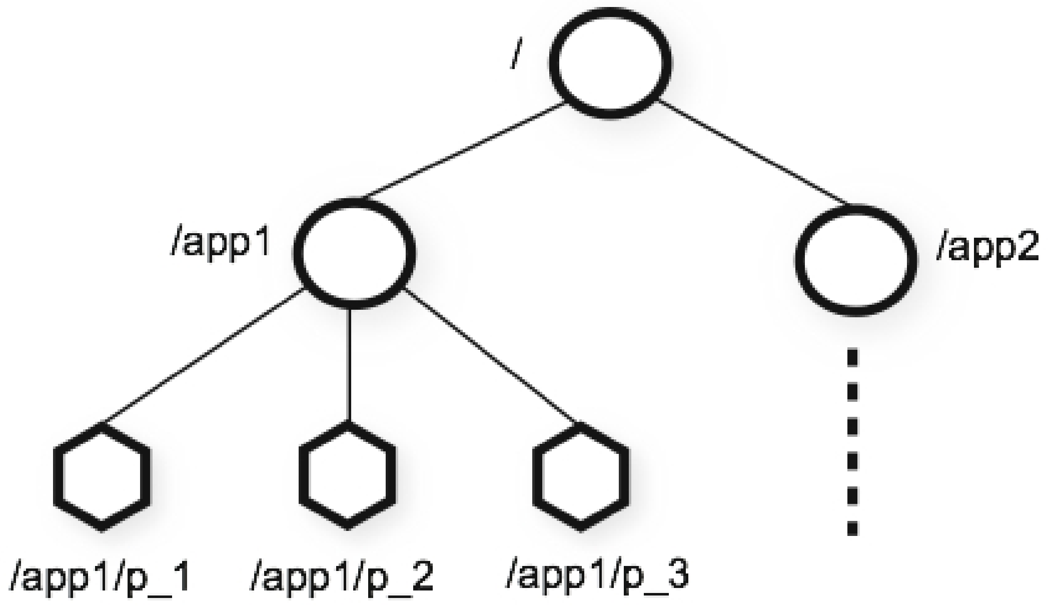


Figure 2.2: ZooKeeper's Hierarchical Namespace

2.1.1 Hierarchical Namespace

ZooKeeper provides a namespace similar to that of a file system. A name in the namespace is a sequence of path elements separated by a slash (/). Every node in ZooKeeper's namespace is identified by a path. The structure can be seen in Figure 2.2.

ZooKeeper calls each of the nodes in the namespace **znode** to distinguish them from all the other nodes described so far. But, what is actually a node in such a namespace?

These aren't the nodes that compose the ZooKeeper ensemble or the nodes in the coordinated distributed system. These znodes are just like any other node in a namespace, such as the one in the file system. Therefore, they could simply represent something akin to directories and files. Inside them, it is possible to represent many different kinds of information (e.g. unstructured data). In reality, in many cases (such as the leader election algorithm that we will see), we won't need the actual content of the nodes, we only need to know whether they exist or not. The znodes in the namespace, in fact, will be created, deleted, and read by the clients of ZooKeeper (that are the nodes in the coordinated distributed system). By creating, checking, and deleting nodes, the distributed system can, for example, easily implement a **mutual exclusion** algorithm by allowing only the node that created a certain node in the ZooKeeper's namespace to operate in the critical section.

ZooKeeper differentiates the znodes by classifying them as:

- **Persistence znode**: Persistence znode is alive even after the client, which created that particular znode, is disconnected. By default, all znodes are persistent unless otherwise specified.

- **Ephemeral znode**: Ephemeral znodes are active until the client is alive. When a client gets disconnected from the ZooKeeper ensemble, then the ephemeral znodes get deleted automatically. For this reason, only ephemeral znodes are not allowed to have children further. If an ephemeral znode is deleted, then the next suitable node will fill its position. Ephemeral znodes play an important role in the leader's election since, as we will see, after the leader node of the distributed system either crashes or is closed, the respective znode it created gets deleted and a new election process starts.

- **Sequential znode**: Sequential znodes can be either persistent or ephemeral. When a new znode is created as a sequential znode, then ZooKeeper sets the path of the znode by attaching a 10-digit sequence number to the original name. For example, if a znode with path */myapp* is created as a sequential znode, ZooKeeper will change the path to */myapp0000000001* and set the next sequence number as *0000000002*.

2.1.2 Watches

One of the most important features of ZooKeeper is the concept of **watch**. Watches are a simple mechanism for the client to get notifications about the changes in the ZooKeeper ensemble namespace. Clients can set watches while reading a particular znode. These watches will send a notification to the registered client for any of the znode (to which the client subscribed) changes.

Znode changes are modifications of data associated with the znode or changes in the znode's children. Watches are triggered only once. Once the watch forwards the notification, it will be deleted, and the client will have to issue another watch to keep getting updates.

In other words, when looking at the ZooKeeper's namespace, a client can subscribe to the changes of a node by creating a watch. Such changes can be, for example, related to the data stored in the znode itself, or even its deletion, which means that after it gets deleted the client gets notified. This is directly related to the previous concept regarding ephemeral nodes and the leader's election.

2.1.3 Simple API

ZooKeeper tries to keep very simple the interaction with the nodes of the coordinated distributed system. As such, it provides a very small set of simple operations, that are:

- **create**: Creates a znode at a certain location in the namespace.

- **delete**: Deletes a znode in the namespace.
- **exists**: Tests if a znode exists at a location.
- **get data**: Reads the data stored in a znode.
- **set data**: Writes data to a znode.
- **get children**: Retrieves a list of children of a znode in the namespace.
- **sync**: Waits for data to be propagated.

These very few operations provide everything we need to communicate with ZooKeeper.

Such operations will be called by the clients to interact with the namespace of ZooKeeper. In our case, as we will see, the clients of the distributed system will be able to call them through the Kazoo library. In fact, each of the libraries offered by each programming language (Kazoo in the case of Python) will provide different interfaces, but overall they will all implement the operations mentioned above.

2.1.4 Guarantees

Lastly, let us discuss the guarantees that Apache ZooKeeper aims to achieve. Directly from the documentation of ZooKeeper, these are:

- **Sequential Consistency**: Updates from a client will be applied in the order that they were sent.
- **Atomicity**: Updates either succeed or fail. No partial results.
- **Single System Image**: A client will see the same view of the service regardless of the server that it connects to (i.e., a client will never see an older view of the system even if the client fails over to a different server with the same session). This is why we say that, even though ZooKeeper is organized as a distributed system, this is made transparent to the clients, offering always the same up-to-date view of the namespace.
- **Reliability**: Once an update has been applied, it will persist from that time forward until a client overwrites the update.
- **Timeliness**: The client's view of the system is guaranteed to be up-to-date within a certain time bound.

2.2 Kazoo

In this section, we will briefly introduce this library[2].

[2]Kazoo documentation website: `https://kazoo.readthedocs.io/en/latest/`

Using Zookeeper in a safe manner can be difficult due to the variety of edge cases in Zookeeper. Kazoo is a Python library designed to make working with ZooKeeper a more hassle-free experience that is less prone to errors.

Overall, we could describe Kazoo as a high-level Python library that simplifies interactions with Zookeeper. It provides an easy-to-use API for connecting to Zookeeper, creating znodes, setting watches, and performing other coordination tasks. The important point is that it provides us with the operations that were described above.

2.2.1 Basic Usage

Just to give an idea of how it works, let's see some code snippets to directly showcase how it implements the interaction with ZooKeeper and enables the client to call the operations from the ZooKeeper API[3].

It all starts by creating the **client** and starting it, which is the lower part component of the distributed system's nodes that we saw in Figure 2.1. This is done as in Code 2.1.

```python
from kazoo.client import KazooClient

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()
```

Code 2.1: Kazoo Client Creation

As we are in localhost, we will connect to host **'127.0.0.1'** and port **'2181'**, as it is the default port for ZooKeeper servers.

This implementation takes into account that the ZooKeeper ensemble is composed of only one node. If we consider a production environment in which we will have more of them, and we want the client to connect to one of them, without specifying exactly which, we can do it by specifying more hosts separated by a comma.

Once the client is instantiated and has been connected to the ZooKeeper ensemble, we can perform all the operations we already mentioned.

For example, we can easily create a new znode in the namespace, as shown in Code 2.2.

```python
# Ensure a path, create if necessary
zk.ensure_path("/my/path")

# Create a new empty node
zk.create("/my/path/node1_")

# Create a new sequential ephemeral empty node
```

[3]Kazoo basic usage website: `https://kazoo.readthedocs.io/en/latest/basic_usage.html`

```
zk.create("/my/path/node2_", ephemeral=True, sequence=True)
```

Code 2.2: Kazoo Znode Creation

The operation **ensure_path()** will recursively create the node and any nodes in the path necessary along the way. In other words, it is useful to create paths in the namespace.

The first **create** operation creates a simple default node, while the second one creates an ephemeral sequential node. Being sequential means that, as mentioned before, ZooKeeper will automatically attach to the node a sequential number to uniquely identify the node, which is **node2_0000000001**.

It is also possible to perform any of the read operations as shown in Code 2.3.

```
# Determine if a node exists
if zk.exists("/my/path"):
   # Do something in case of existence

# List the children
children = zk.get_children("/my/path")
print("There␣are␣%s␣children␣with␣names␣%s" % (len(children), children))
```

Code 2.3: Kazoo Namespace Reading

The second operation will print the children under the path **"my/path"**, which, if related to the previous execution, are the nodes *node1_* and *node2_0000000001*.

And obviously, we could also call the other operations from the API, such as *delete*, *set data*, *get data*, and so on...

Everything can be found in the documentation, but to understand the code in the next chapter, this is enough.

2.2.2 Watchers

Let us now see one last important operation, which is how to implement watchers in Kazoo.

Kazoo can set watch functions on a node that can be triggered either when the node has changed or when the children of the node change. This change to the node or children can also be the node or its children being deleted, similar to what we said in the previous section.

Watchers can be set in two different ways. The first method allows us to pass custom functions as arguments to reading functions, such as the function **get_children()** we saw before, which will be called when the data on the nodes changes or the nodes are deleted.

In our case, we are interested in the second approach, which allows clients to watch for data and children modifications without re-setting the watcher every time the event is triggered. As we will see directly in the implementation, this is done by specifying a special decorator provided by Kazoo. This decorator depends on the objective, in fact, it can either be attached to a directory to watch the *children*, or to a single node, as we will see in our case. Using these decorators, when the respective function gets invoked, it will automatically pass as arguments the contents of the watched node.

2.3 Leader Election Algorithm Using Zookeeper

In this last section, we will introduce the leader election algorithm performed in a distributed system using ZooKeeper. The respective actual code and implementation will be explained in the next chapter.

In reality, ZooKeeper doesn't exactly provide a leader election algorithm, but it gives us the facilities to easily implement it. In fact, as for the implementation of a mutual exclusion algorithm, we can implement this algorithm by using the creation and deletion of nodes in ZooKeeper's namespace.

The idea of this algorithm is to let each process (executed in the coordinated distributed system), interested in the leader's position, participate in the election by performing the following steps:

1. Create a znode in the namespace. In particular, the znode needs to be an ephemeral sequential node, so that once the process that created it crushes, it will delete automatically the znode it created. The sequential feature is needed to order all the created znodes.

2. Consider the ordered list of all znodes created by the current client and all the other clients that created them before, ordering them by the sequential number assigned by ZooKeeper at the creation.

3. If the znode created by the current client is the first of the list (meaning that it is the znode with the current smallest sequence number), the client will be elected as the leader.

4. If the znode created by the current client isn't the first of the list, it means that the client will be currently a follower and it will create a *watcher* to the previous znode (the znode in the list with smaller sequence number than the current znode right before it). When this znode is deleted, the current client will perform again the election process hoping to become the leader this time. But, in this new election, it won't create a new znode since it has already created it.

The algorithm will basically elect as the leader the client that created the znode with the smallest sequence number.

The most important step is the last one. For example, if we consider the case in which the client participating in the process is the first one, it means that it will create a znode with the smallest sequence number (being also the only one at the beginning), thus electing the client as the leader. If another client joins the election process after the previous one, this client will create a znode with a higher sequence number, meaning that it will be a follower. But, if the current leader crushes, the znode created at the beginning will be deleted automatically, since we specifically defined it as *ephemeral*. Therefore, the next client who was watching that node will be notified. This will try again to be elected as a leader, and not having another node before the one it created, means that it will now go from the state of follower to the state of leader.

Note that a situation in which the clients apply for the leader's election at the same time and there is concurrency isn't at all a problem. This is because everything related to the concurrent creation of the znodes and the assignment of the sequence numbers is handled efficiently and precisely by ZooKeeper, making the overall process on the client side much easier.

3. Code and Execution

In this chapter, we will analyze the practical execution of what was formulated in the previous chapter when explaining the algorithm. Here, we will implement it.

3.1 Installation

To execute this project, we need to install the following things:

- Java JDK[1] (since ZooKeeper runs in Java).
- ZooKeeper service[4].
- Python interpreter[3].
- Kazoo library[2].

For each of the services to install, there is a respective link to reach the documentation.

3.2 Setting up the Environment

We can check if Java was installed and working by executing from the terminal the command in Code 3.1.

```
$ java -version
```

Code 3.1: Java Version Checking

After checking this, we can start by setting up the ZooKeeper ensemble[1]. In fact, before executing the Python code, we will obviously need to start the ZooKeeper ensemble to then connect to it. Before starting the server, we need to go inside the directory downloaded from ZooKeeper and create a file **zoo.cfg** under the **conf/** directory and insert the following content in Code 3.2.

[1]Setting up ZooKeeper: `https://zookeeper.apache.org/doc/current/zookeeperStarted.html`

```
tickTime=2000
dataDir=/tmp/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
```

Code 3.2: ZooKeeper Configuration File

Inside, we are specifying:

- **dataDir**: The location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database.
- **tickTime**: The basic time unit in milliseconds used by ZooKeeper. It is used to do heartbeats and the minimum session timeout will be twice the tickTime.
- **clientPort**: The port to listen for client connections.
- **initLimit**: Amount of time, in ticks, to allow followers to connect and sync to a leader. Increased this value as needed, if the amount of data managed by ZooKeeper is large.
- **syncLimit**: Amount of time, in ticks, to allow followers to sync with ZooKeeper. If followers fall too far behind a leader, they will be dropped.

These are the basic parameters, but there are also others if needed[2].

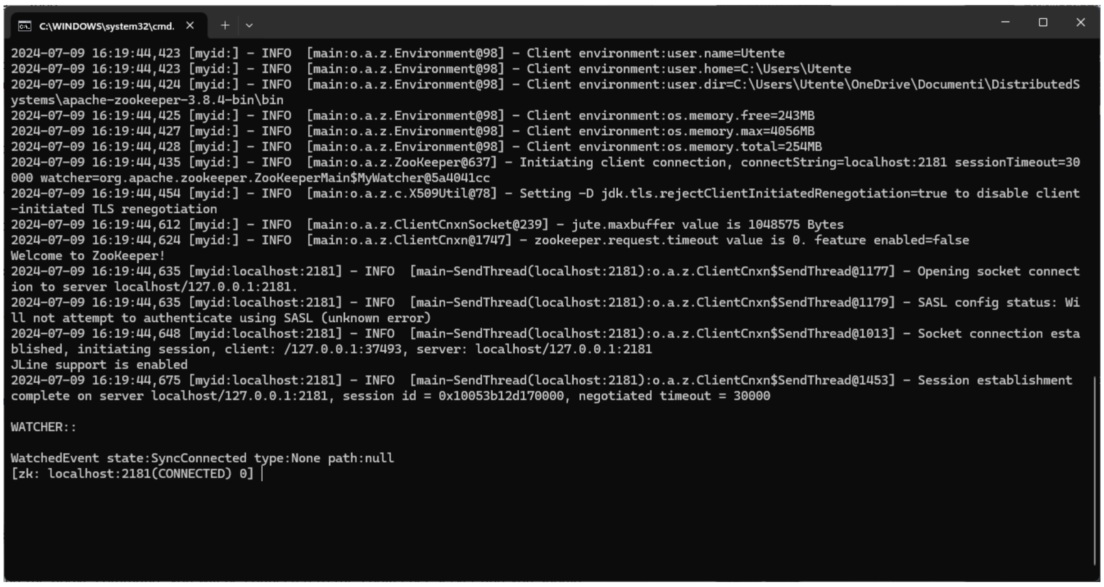
Note that, if we want instead to execute a ZooKeeper ensemble composed of more nodes (not only one), we need to specify the address of each of them in this configuration file. For example, if we had 3 different machines, with 3 different IP addresses **zoo1**, **zoo2**, and **zoo3**, we would have to modify the file as shown in Code 3.3.

```
tickTime=2000
dataDir=/tmp/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

Code 3.3: ZooKeeper Configuration File 3 Servers

For the purpose of this project, we won't set up 3 nodes in 3 different machines for simplicity. From the documentation, we know that it is also possible to execute all 3 nodes locally, but this wouldn't make sense since it won't be fault tolerant having all 3 nodes on the same machine. Therefore, we will stick with the first configuration file.

[2]ZooKeeper parameters website: `https://zookeeper.apache.org/doc/current/zookeeperAdmin.html`



```
2024-07-09 16:19:44,423 [myid:] - INFO  [main:o.a.z.Environment@98] - Client environment:user.name=Utente
2024-07-09 16:19:44,423 [myid:] - INFO  [main:o.a.z.Environment@98] - Client environment:user.home=C:\Users\Utente
2024-07-09 16:19:44,424 [myid:] - INFO  [main:o.a.z.Environment@98] - Client environment:user.dir=C:\Users\Utente\OneDrive\Documenti\DistributedS
ystems\apache-zookeeper-3.8.4-bin\bin
2024-07-09 16:19:44,425 [myid:] - INFO  [main:o.a.z.Environment@98] - Client environment:os.memory.free=243MB
2024-07-09 16:19:44,427 [myid:] - INFO  [main:o.a.z.Environment@98] - Client environment:os.memory.max=4056MB
2024-07-09 16:19:44,428 [myid:] - INFO  [main:o.a.z.Environment@98] - Client environment:os.memory.total=254MB
2024-07-09 16:19:44,435 [myid:] - INFO  [main:o.a.z.ZooKeeper@637] - Initiating client connection, connectString=localhost:2181 sessionTimeout=30
000 watcher=org.apache.zookeeper.ZooKeeperMain$MyWatcher@5a4041cc
2024-07-09 16:19:44,454 [myid:] - INFO  [main:o.a.z.c.X509Util@78] - Setting -D jdk.tls.rejectClientInitiatedRenegotiation=true to disable client
-initiated TLS renegotiation
2024-07-09 16:19:44,612 [myid:] - INFO  [main:o.a.z.ClientCnxnSocket@239] - jute.maxbuffer value is 1048575 Bytes
2024-07-09 16:19:44,624 [myid:] - INFO  [main:o.a.z.ClientCnxn@1747] - zookeeper.request.timeout value is 0. feature enabled=false
Welcome to ZooKeeper!
2024-07-09 16:19:44,635 [myid:localhost:2181] - INFO  [main-SendThread(localhost:2181):o.a.z.ClientCnxn$SendThread@1177] - Opening socket connect
ion to server localhost/127.0.0.1:2181.
2024-07-09 16:19:44,635 [myid:localhost:2181] - INFO  [main-SendThread(localhost:2181):o.a.z.ClientCnxn$SendThread@1179] - SASL config status: Wi
ll not attempt to authenticate using SASL (unknown error)
2024-07-09 16:19:44,648 [myid:localhost:2181] - INFO  [main-SendThread(localhost:2181):o.a.z.ClientCnxn$SendThread@1013] - Socket connection esta
blished, initiating session, client: /127.0.0.1:37493, server: localhost/127.0.0.1:2181
JLine support is enabled
2024-07-09 16:19:44,675 [myid:localhost:2181] - INFO  [main-SendThread(localhost:2181):o.a.z.ClientCnxn$SendThread@1453] - Session establishment
complete on server localhost/127.0.0.1:2181, session id = 0x10053b12d170000, negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0]
```

Figure 3.1: ZooKeeper's CLI

At this point, we can execute the server by executing the respective called **zk-Server.cmd** under the **bin** folder since I am working on Windows. This, in fact, depends on the operating system. In UNIX-like systems, the program to execute is *zkServer.sh*.

If everything is working so far, we can move on by trying to execute the ZooKeeper CLI to interact with the ZooKeeper ensemble. This can be done by executing the program called **zkCli.cmd** under the bin folder again. This should give us the result shown in Figure 3.1. In the next steps, we won't be actually using the CLI since we use Kazoo just to avoid this. Still, it is useful to understand if the service works. Inside the CLI we can also try to execute the commands illustrated in the previous chapter from the ZooKeeper API to interact with ZooKeeper's namespace.

3.3 Solution

After the previous steps are working correctly, we can move to the actual code which simulates a leader election process in a distributed system.

The program that we need to execute is the one shown in Code 3.4.

```python
from kazoo.client import KazooClient
import time
import os

ZK_HOSTS = '127.0.0.1:2181'
ELECTION_PATH = "/election"
current_znode_path = None
is_leader = False

```

```
def zk_client():
    zk = KazooClient(hosts=ZK_HOSTS)
    zk.start()
    return zk

def watch_node(zk, node_to_watch):
    @zk.DataWatch(f"{ELECTION_PATH}/{node_to_watch}")
    def watch_node(data, stat):
        if stat is None:
            print(f"Watched␣node␣{node_to_watch}␣went␣away,␣triggering␣re-election")
            elect_leader(zk)

def elect_leader(zk):
    global current_znode_path, is_leader
    # Create an ephemeral sequential node
    if current_znode_path is None:
        current_znode_path = zk.create(ELECTION_PATH + "/node_",
                                       ephemeral=True, sequence=True)
    nodes = zk.get_children(ELECTION_PATH)
    nodes.sort()

    # Check if the current node is the smallest one (i.e., the leader)
    if current_znode_path == f"{ELECTION_PATH}/{nodes[0]}":
        print(f"{current_znode_path}␣is␣the␣leader")
        is_leader = True
    else:
        next_node = nodes[nodes.index(current_znode_path.split('/')[-1]) - 1]
        watch_node(zk, next_node)
        print(f"{current_znode_path}␣is␣not␣the␣leader")
        is_leader = False

def run_election():
    global is_leader
    zk = zk_client()
    zk.ensure_path(ELECTION_PATH)
    elect_leader(zk)

    while True:
        if is_leader:
            print(f"Process␣{os.getpid()}␣is␣doing␣leader␣work...")
        else:
            print(f"Process␣{os.getpid()}␣is␣waiting␣or␣doing␣non-leader␣work...")

        # Wait for a short period before checking again
        time.sleep(5)

if __name__ == "__main__":
    run_election()
```

Code 3.4: Python Leader Election

Let us explain part by part how the code is structured.

Starting with the function **zk_client()**, this function is used to create the ZooKeeper client we mentioned many times and tries to connect to the host of the ZooKeeper ensemble, which in this case is *127.0.0.1:2181*. It then starts the client and returns the client itself.

Moving on to **run_election()**, this function first creates the client, ensures the path **'/election'**, under which we will create the nodes to perform the election algorithm, and calls the function to elect the leader. If the current client is the leader, then it just informs us that the process is the leader. Otherwise, it prints that the client is a follower, and this information is repeated with an infinite loop.

The function **elect_leader()** instead does most of the work. It starts by creating a new ephemeral sequential znode called **'/election/node_SEQUENCE-NUMBER'**, where the sequence number is the one assigned by ZooKeeper. This is done in case the process still hasn't created a new node. This operation ensures that when the function is called again in case of re-election, the process doesn't create a new node and sticks with the one it had already created. After this, it first collects all current znodes under the path *'/election'* and orders them by sequential numbers. Lastly, if the node is the first of the list, it communicates to the client that it is the leader and modifies the global variable **is_leader = True**. Otherwise, it first collects the direct predecessor of the znode in the ordered list and creates a *watcher* to that node, checking when this will change (or be deleted). After creating the watcher, it informs the client that it isn't the leader and and modifies the global variable **is_leader = False**.

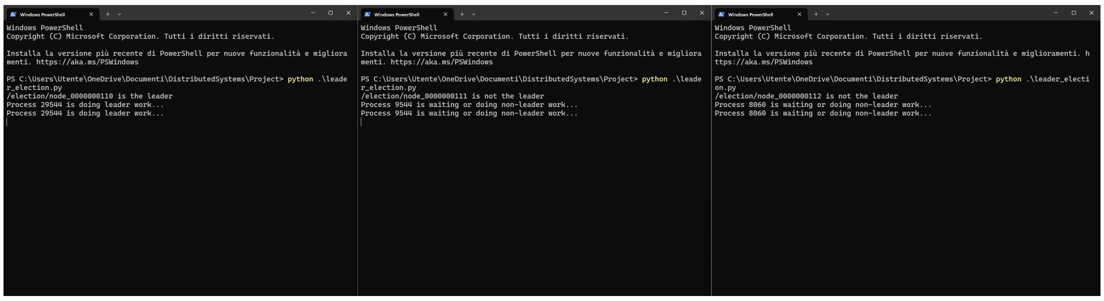
Lastly, the function **watch_node()** will create a watcher for a certain node. This is done in Kazoo using the decorator **@zk.DataWatch()** to which we can pass the path of the node to keep track of. When the observed node changes, the client will be notified and start a new election, but this time, it won't create a new znode.

Each of these points perfectly matches the described algorithm in the previous chapter.

As we can see, the code related to the actual distributed system to coordinate is literally composed of only some prints in the command line to check periodically if a client is a leader or still a follower. This is because I wanted to showcase the features offered by ZooKeeper and not focus on the distributed system itself. Still, in the section in which the process loops, we could instead specify the actual work that should be done in the process in case it is the leader or it is a follower. The program showed in Code 3.4 is just the starting point to then structure the distributed application.

3.4 Execution and Results

Once the program has been written, to actually simulate a distributed system, we will first have to open some terminals. Each of them will represent a different node in the coordinated distributed system. Once each terminal points to the directory in which the Python program is saved, we can execute it by launching the command in Code 3.5.

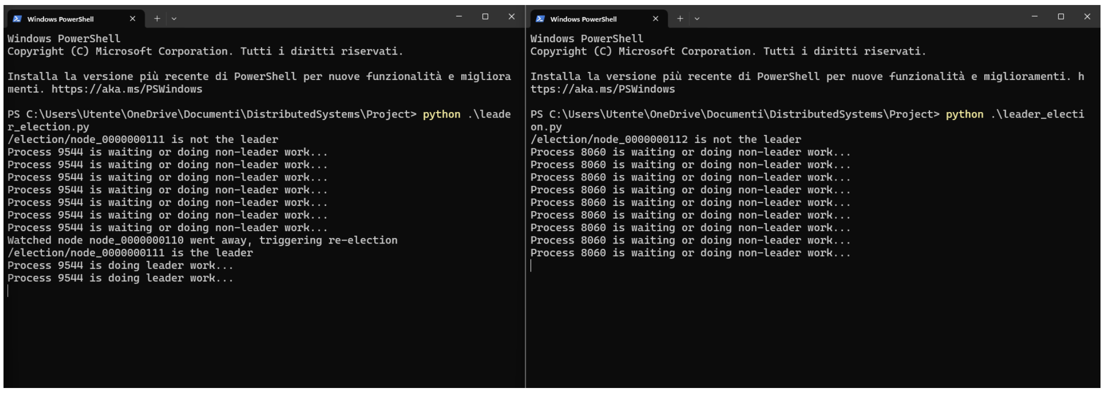


```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Installa la versione più recente di PowerShell per nuove funzionalità e migliora
menti. https://aka.ms/PSWindows

PS C:\Users\Utente\OneDrive\Documenti\DistributedSystems\Project> python .\leade
r_election.py
/election/node_0000000110 is the leader
Process 29544 is doing leader work...
Process 29544 is doing leader work...
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Installa la versione più recente di PowerShell per nuove funzionalità e migliora
menti. https://aka.ms/PSWindows

PS C:\Users\Utente\OneDrive\Documenti\DistributedSystems\Project> python .\leade
r_election.py
/election/node_0000000111 is not the leader
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Installa la versione più recente di PowerShell per nuove funzionalità e miglioramenti. h
ttps://aka.ms/PSWindows

PS C:\Users\Utente\OneDrive\Documenti\DistributedSystems\Project> python .\leader_electi
on.py
/election/node_0000000112 is not the leader
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
```

Figure 3.2: Result with all 3 Processes



```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Installa la versione più recente di PowerShell per nuove funzionalità e migliora
menti. https://aka.ms/PSWindows

PS C:\Users\Utente\OneDrive\Documenti\DistributedSystems\Project> python .\leade
r_election.py
/election/node_0000000111 is not the leader
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
Process 9544 is waiting or doing non-leader work...
Watched node node_0000000110 went away, triggering re-election
/election/node_0000000111 is the leader
Process 9544 is doing leader work...
Process 9544 is doing leader work...
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Installa la versione più recente di PowerShell per nuove funzionalità e miglioramenti. h
ttps://aka.ms/PSWindows

PS C:\Users\Utente\OneDrive\Documenti\DistributedSystems\Project> python .\leader_electi
on.py
/election/node_0000000112 is not the leader
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
Process 8060 is waiting or doing non-leader work...
```

Figure 3.3: Result after Deleting Process Leader

```
$ python leader_election.py
```

Code 3.5: Python Launch Program

This command will have to be executed by each of the terminals. This way, each of them will be part of the leader election process.

As we will see, only one of them will be a leader, while all the others will be followers, as shown in Figure 3.2. In this figure, only the process on the left has become the leader, while the other 2 are followers. In particular, having first executed the program on the left, then the middle, and lastly the right, it means that the process in the middle is watching the node created by the process on its left (the leader), while the process on the right is looking at the node created by the process in the middle.

If we now try to close the terminal representing the leader, we will observe that the other nodes will notice that each of the znodes observed by each of them will have changed, meaning that a new leader election process is issued and a new leader is elected, as shown in Figure 3.3. In the figure, we can see that after closing the leader process, a new leader has been elected (the process on the left), while the process on the right hasn't noticed any changes since the previous node that it is watching hasn't changed at all, meaning that the watcher hasn't notified anything and the process will remain a follower. The processes behaved exactly as it was intended. By closing the terminal (simulating the crash of the leader) the respective ephemeral node is deleted and the watcher of the process in the middle notifies the process, which starts a new election checking if it became the leader.

4. Conclusions

This project demonstrated how to implement a leader election algorithm using ZooKeeper and Kazoo capabilities to coordinate efficiently and easily a distributed system. In the end, the program just showed a starting point above of which we can actually develop a distributed system that leverages the leader election algorithm, thus allowing us to extend it for much more complex and structured applications.

As we have seen, ZooKeeper doesn't exactly provide this feature, but it gives us the possibility to implement it using the offered namespace in which we can create, read, and delete nodes. This makes the actual implementation very simple and comprehensible. By providing the notion of ephemeral nodes, in fact, we can easily implement the algorithm shown in the project.

Through Kazoo, instead, we can use ZooKeeper even in a Python environment, one of the most famous languages at the moment. This library gave us the possibility to easily allow for communication and interaction with the ZooKeeper ensemble by implementing the same exact API that we could have used through the CLI.

Not to mention, that this is just one of the many features offered by ZooKeeper, allowing for many other different scenarios in which by relying on ZooKeeper we can implement fault-tolerance and robust solutions to coordinate distributed systems.

Overall, I think that the project was extremely helpful in really grasping the potentialities of the ZooKeeper service. By only relying on the theory, we can understand only part of it. It is with practice that we can face problems and things that we didn't think about before. I would be very interested in deploying an actual distributed application that leverages ZooKeeper to coordinate all the nodes.

Also, even though after searching online I found out that this service isn't anymore so popular, I am more than sure that this will still be one of the starting points on which new services (akin to ZooKeeper) will be developed.

Bibliography

[1] *java-download*. URL: https://www.oracle.com/it/java/technologies/downloads/.

[2] *kazoo-installation*. URL: https://kazoo.readthedocs.io/en/latest/install.html.

[3] *python-download*. URL: https://www.python.org/downloads/.

[4] *zookeeper-download*. URL: https://zookeeper.apache.org/doc/current/zookeeperStarted.html#sc_Download.