University of Camerino

School of Science and Technology

Master Degree in Computer Science (Class LM-18)

Distributed Systems

ZooKeeper Leader Election

From Algorithm to Distributed System Prototype

Student

Sofia Scattolini

Supervisor

Prof. Michele Loreti

A.Y. 2024/2025

Abstract

This project explains how a distributed **leader election algorithm** was designed and implemented using **Apache ZooKeeper**.

Leader election is important in distributed systems because it helps nodes work together and remain consistent. A prototype was developed in Python using the Kazoo client to communicate with ZooKeeper.

Using **ephemeral sequential znodes** and **watchers**, the system automatically selects a leader and replaces it if it fails. This creates a reliable and consistent coordination method which can be used in real-world distributed systems.

Contents

List of Figures

Listings

List of Tables

1. Introduction

1.1 Project Objective

In distributed systems, it is important that all processes work together in a coordinated way. This helps the system stay active and correct, even when some parts fail. One common solution is the **leader election** mechanism. This means choosing one process as the *leader*, which does the main coordination work, while the other processes (called *followers*) wait and are ready to take over if needed.

The goal of this project is to learn how the leader election works in **Apache ZooKeeper**, a tool often used to manage coordination in distributed systems. To do this, a simple application was developed using Python and the **Kazoo library**, which makes it easier to use ZooKeeper from Python.

In the application, each client connects to ZooKeeper and creates a temporary and ordered node (called an *ephemeral sequential znode*). The client that creates the node with the lowest number becomes the leader. The others become followers and monitor the node created just before theirs. If the leader stops working, ZooKeeper automatically starts a new election and a new leader is chosen. In this way, the system can continue to work without stopping.

This project shows how ZooKeeper makes it easier to manage leader election in distributed systems, helping them to be more stable and fault-tolerant.

`The full implementation is available on the GitHub repository [1]:`
`https://github.com/Sophisss/ZooKeeper_LeaderElection.`

1.2 Structure of the Report

This report is organized as follows:

- **Chapter 2**: explains what Apache ZooKeeper is, how it works, and how leader election is done;
- **Chapter 3**: shows how the Python application was built, with code examples and tests;
- **Chapter 4 (Conclusions)**: gives a summary of what was learned, and explains the strengths and possible limits of this approach.

2. Apache Zookeeper: Leader Election Algorithm

This chapter introduces Apache ZooKeeper, a coordination service for distributed systems. It begins with an overview of ZooKeeper's architecture, including its server ensemble structure, the znode-based data model, and the consistency guarantees it provides.

Then, the chapter focuses on one of the most common coordination mechanisms: leader election. This is the process in which one node is elected as the leader among a group of distributed processes. ZooKeeper doesn't provide a built-in command for this, but it makes the implementation easy using ephemeral sequential znodes and watchers.

Finally, the Kazoo library is introduced as a Python interface that simplifies the implementation of coordination tasks, including leader election, in distributed applications.

2.1 What is ZooKeeper?

Apache ZooKeeper [2] is an open-source service that helps distributed applications manage coordination tasks, such as electing a leader, synchronizing access to shared resources, and so on.

In distributed systems, many processes or nodes work together over a network. Often, they run on different machines and need to stay in sync and agree on shared decisions. Doing this correctly is difficult, especially when the nodes can fail or become temporarily unreachable. ZooKeeper was created to solve this kind of problem and to make it easier for developers to manage coordination between nodes.

ZooKeeper follows the **CP model** of the **CAP theorem**, which means it guarantees:

- **Consistency**: all clients see the same data, no matter which server they connect to;
- **Partition Tolerance**: the system continues to work correctly even if some parts of the network fail;
- **Availability** might be affected during some failures in order to keep the data consistent and correct.

Due to these properties, ZooKeeper is often used in systems where correct coordination matters more than constant availability.

2.2 Architecture Overview

ZooKeeper is organized as a *distributed service*, made up of several servers called an **ensemble**. These servers run together to provide a single, consistent coordination system for all clients.

- In every ensemble, **one server is elected as the leader**, and the others are **followers**.
- Clients connect to one of the servers in the ensemble to perform operations.
- Although the service is distributed internally to clients, it looks like a single reliable system.

ZooKeeper uses a **quorum-based system** to make decisions. That is why it is recommended to run ZooKeeper with an **odd number of nodes** (for example, 3, 5 or 7). This ensures that the system can tolerate failures and still reach an agreement.

The ensemble handles internal tasks such as leader election (inside ZooKeeper itself), data replication, and client request processing.

From the client's point of view, it only needs to know the address of the ensemble (usually a list of servers) and interact with it through a simple set of operations.

2.2.1 Data Model and ZNodes

ZooKeeper stores all data in a structure called a **hierarchical namespace**, which works similarly to a file system.

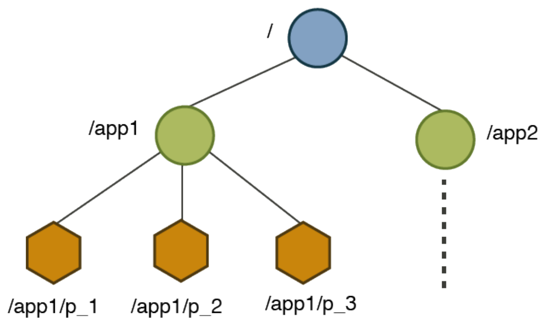


Figure 2.1: ZooKeeper's Hierarchical Namespace

Each data item is stored in a **znode**, which is like a file or folder. Every znode has a unique path (such as `/election/node_000000001` and can contain:

- small amounts of data (in bytes);
- other child znodes (like folders).

There are three main types of znodes:

1. **Persistent znode**: stays in ZooKeeper until deleted, even if the client disconnects;

2. **Ephemeral znode**: automatically deleted when the client session ends (for example if the process crashes);
3. **Sequential znode**: when created, ZooKeeper adds a unique number at the end of its name. This is useful for ordering operations, like in the leader election.

2.2.2 Watchers: How Clients React to Changes

ZooKeeper allows clients to **set watchers** on znodes. A **watch** is a notification mechanism that tells the client when something changes in the znode, such as:

- the node is deleted,
- the node's data changes,
- the node's list of children changes.

Watches are **one-time events**: once triggered, they are removed, and must be set again if needed.

This feature is very useful for building reactive systems. For example, a follower can watch the znode of the current leader, and if the leader crashes (and the znode is deleted), the follower is notified and can try to become the new leader.

2.2.3 API and Guarantees

ZooKeeper keeps its API small and simple, so it can be used easily in many programming languages. The basic operations include:

- `create(path, data)`: creates a new znode;
- `delete(path)`: deletes a znode;
- `exists(path)`: checks if a znode exists;
- `get(path)`: reads data;
- `set(path, data)`: writes data;
- `get_children(path)`: returns list of child znodes.

Although these operations are simple, they are powerful enough to build complex coordination systems.

ZooKeeper also provides the following **guarantees**, which make it reliable:

- **Sequential Consistency**: updates from a client are applied in order;
- **Atomicity**: operations either complete fully or not at all;
- **Single System Image**: all clients see the same view of the data, no matter which server they connect to;
- **Reliability**: once data is written, it won't disappear unless explicitly deleted;
- **Timeliness**: changes are propagated within a known time window.

These properties make ZooKeeper a strong foundation for building fault-tolerant distributed systems.

2.3 Leader Election

One of the most common use cases for ZooKeeper is **leader election**. This means selecting a process as the coordinator or main actor among a group of distributed processes.

ZooKeeper does not have a built-in leader election command, but we can easily implement it using **ephemeral sequential znodes** and **watchers**. Here is how the algorithm works:

- Each process (or client) creates an **ephemeral sequential znode** under a shared path, like `/election`;
- ZooKeeper assigns a unique number to each znode (for example, `node_0000000003`);
- Each client checks the list of znodes under `/election` and sorts them;
- The client with the **smallest sequence number** becomes the **leader**;
- The other clients become **followers**, and each one watches the node just before theirs;
- If the node being watched is deleted (for example, if the leader crashes), the watcher is triggered;
- The client re-runs the election process to see if it can now become the leader.

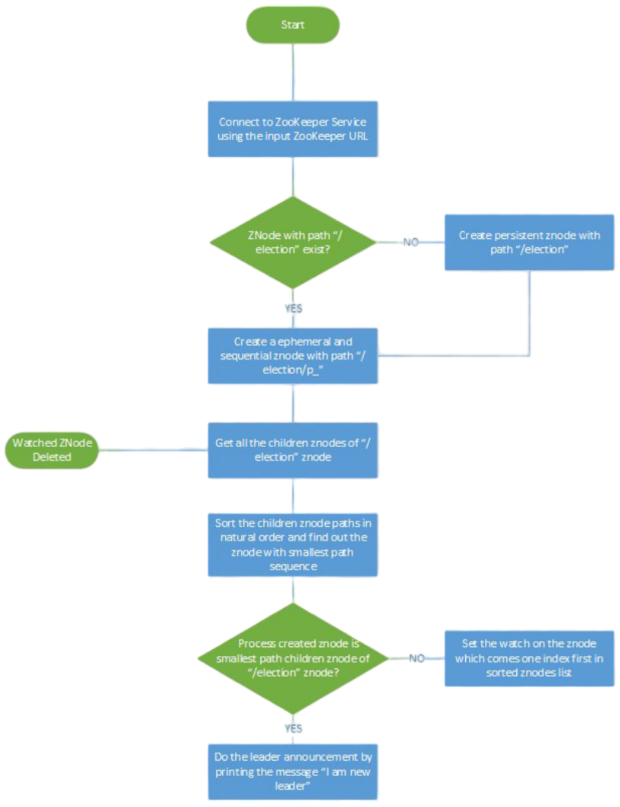

Start
Connect to ZooKeeper Service using the input ZooKeeper URL
ZNode with path "/election" exist?
NO
Create persistent znode with path "/election"
YES
Create a ephemeral and sequential znode with path "/election/p_"
Watched ZNode Deleted
Get all the children znodes of "/election" znode
Sort the children znode paths in natural order and find out the znode with smallest path sequence
Process created znode is smallest path children znode of "/election" znode?
NO
Set the watch on the znode which comes one index first in sorted znodes list
YES
Do the leader announcement by printing the message "I am new leader"

Figure 2.2: ZooKeeper's Leader Election Process

This approach ensures that there is always **one active leader**, and when it fails, a new leader is automatically elected.

This system is:

1. **Simple**: uses a small number of API calls;
2. **Fair**: any client can become leader;
3. **Reactive**: clients are notified immediately on changes;
4. **Fault-tolerant**: works correctly even when processes fail.

2.4 Algorithms Comparison

Leader election is a key problem in distributed systems, where nodes must agree on one coordinator to manage tasks and maintain consistency. Different algorithms exist, each with pros and cons depending on the system needs. Below is a brief comparison of common leader election methods, including ZooKeeper's approach.

Bully Algorithm

- **How it works**: When a node notices that the leader is not responding to requests, it sends election messages to nodes with higher IDs. If no one responds, it becomes leader; otherwise, it waits for a leader announcement.
- **Limitations**: Depends on nodes knowing IDs and reliable communication.

Ring Algorithm

- **How it works**: A token with a node's ID circulates around a logical ring. Each node adds its ID if it's higher. After a full round, the highest ID is elected leader.
- **Limitations**: Election time grows with ring size; failure of a single node can stop the process.

Paxos-based Election

- **How it works**: The nodes propose candidates and exchange messages to reach consensus on a leader.
- **Limitations**: Complex to implement; high message overhead.
- **Fault tolerance**: Designed for asynchronous networks with possible message loss.

Raft

- **How it works**: The nodes vote during the terms to elect a leader who manages log replication.
- **Fault tolerance**: Robust for network partitions and failures.

ZooKeeper Election

- **How it works**: Clients create ephemeral sequential znodes under a common path; the client with the smallest sequence number becomes leader, others watch their predecessor.
- **Fault tolerance**: Automatically handles leader failures; ensemble guarantees consistency.

Table 2.1: Comparison of Leader Election Algorithms in Distributed Systems

Algorithm	**How It Works**	**Strengths & Limitations**
Bully	Nodes with higher IDs respond to election messages; node with the highest ID becomes leader	Simple to understand; requires known node IDs and reliable communication
Ring	A token circulates around the ring; node with highest ID becomes leader	Simple token passing; vulnerable to node failure; slower as ring size grows
Paxos	Nodes exchange messages to reach consensus on a leader	Strong fault tolerance; complex implementation; high message overhead
Raft	Nodes vote to elect a leader who manages log replication	Easier to understand than Paxos; robust against network partitions and failures
ZooKeeper	Clients create ephemeral sequential znodes; node with smallest sequence number becomes leader	Simple API with robust failover; depends on ZooKeeper ensemble availability

2.5 Using Kazoo in Python

In this project, the leader election was implemented using **Kazoo** [3], a Python library that wraps the ZooKeeper API and makes it easier to use.

Kazoo supports:

- Connection management and reconnection;
- Creation of ephemeral and sequential znodes;
- Setting watchers using decorators;
- Simple methods to read/write/delete znodes.

Using Kazoo, we can write clean and understandable code that handles all the logic of the election process. Each client becomes part of the system, connects to ZooKeeper, creates its znode, and either becomes leader or watches another process.

Thanks to this mechanism, we get a distributed system that can handle failures automatically and continue to work without manual intervention.

3. System Implementation

This chapter illustrates the development of the prototype system that demonstrates leader election in a distributed environment using Python, Apache ZooKeeper, and the Kazoo library. Describes the setup of the environment, the architecture of the application, and the implementation of the core logic behind the election process. The chapter also includes an explanation of the runtime behavior observed during testing and discusses the main challenges encountered.

The goal was to build a lightweight but functional distributed system capable of electing a leader among multiple processes and ensuring automatic failover in case the leader becomes unavailable.

3.1 Prerequisites

To develop and run this project, the following software and libraries were required:

- **Java JDK** (version 11 or higher) [4]: Apache ZooKeeper is a Java-based service and requires the Java Virtual Machine (JVM) to run correctly.
- **Python Interpreter** (version 3.8 or higher) [5].
- **Kazoo Library** [6]: A high-level Python library that provides an abstraction over ZooKeeper API. It simplifies operations such as creating znodes, setting watchers, and handling connections. It can be installed using pip:

  ```
  pip install kazoo
  ```

- **Docker** [7]: Used to run ZooKeeper in a containerized environment, avoiding the need to install and configure it manually on the host machine. This ensures portability and reproducibility across different systems.

3.2 Environment Setup

Instead of installing ZooKeeper directly on the local machine, Docker was used to run an isolated ZooKeeper server using the official image available on Docker Hub [8]. This solution reduces system dependency issues and allows for quick setup between environments.

The following command was used to start the ZooKeeper service:

```
docker run -d --name zookeeper -p 2181:2181 zookeeper
```

This command pulls the ZooKeeper image (if not already available locally), starts a container in detached mode, and exposes port `2181`, which is the default ZooKeeper client port. After execution, the server can be reached at `127.0.0.1: 2181`, enabling Python clients to connect and participate in coordination tasks.

It is essential to ensure that Docker is running and that the container is active before launching the application.

3.3 Application Design

The application is designed to simulate a distributed system with multiple participants competing to become the leader. The coordination is handled by ZooKeeper, which provides primitives for distributed consensus.

Each process connects to ZooKeeper and creates an **ephemeral sequential znode** under the path `/election`. These znodes are temporary and automatically removed when the session that created them ends (for example, if the process crashes or disconnects). The sequential suffix provides a unique and ordered identifier for each node.

Leader Election Protocol

The election algorithm follows these steps:

1. Each process creates its own unique ephemeral sequential znode inside the election path;

2. The process whose znode has the smallest sequence number is elected as the **leader**;

3. All other processes become **followers** and set a **watch** on their immediate predecessor znode. The watch is a ZooKeeper feature that notifies the process when the watched node is deleted;

4. When the leader process stops (or crashes), its ephemeral znode is automatically deleted by ZooKeeper;

5. The follower watching the deleted node receives a notification and re-executes the leader election procedure to check if it can become the new leader.

This protocol ensures that only one leader exists at a time and that the system can automatically recover from failures without human intervention.

3.4 Implementation Details

The implementation uses an **object-oriented design**. The core logic is encapsulated in a class called `LeaderElection`, which handles the client connection, node creation, election, and re-election processes.

3.4.1 ZooKeeper Client Initialization

Each process creates an instance of `KazooClient` and connects to the ZooKeeper server:

```python
from kazoo.client import KazooClient

ZK_HOSTS = "127.0.0.1:2181"
ELECTION_PATH = "/election"

class LeaderElection:
    def __init__(self, zk_hosts, election_path):
        self.zk_hosts = zk_hosts
        self.election_path = election_path
        self.zk = KazooClient(hosts=self.zk_hosts)
        self.z_node_path = None
        self.is_leader = False
```

Listing 3.1: ZooKeeper Client Setup

3.4.2 Ephemeral Sequential Node Creation

After establishing the connection, the process ensures that the election path exists, creates the ephemeral sequential node, and starts the election:

```python
    def start(self):
        self.zk.start()
        self.zk.ensure_path(self.election_path)
        self.z_node_path = self.zk.create(
            f"{self.election_path}/node_",
            ephemeral=True,
            sequence=True
        )
        self.elect_leader()
```

Listing 3.2: Ephemeral Sequential Node Creation

3.4.3 Leader Election Logic

The `elect_leader()` function contains the core logic of the leader election. It first retrieves and sorts all child znodes under `/election` by their sequence number. If the client's own znode is the one with the smallest sequence number, it is elected leader and the `is_leader` flag is set accordingly.

Otherwise, the client identifies its immediate predecessor znode and sets a watch on it. The watch asynchronously monitors the predecessor node for deletion events, enabling the client to be notified when the predecessor disappears (for example, due to a crash or disconnection). Upon notification, the election process restarts, but the client reuses its existing ephemeral sequential node to avoid duplications.

```python
    def elect_leader(self):
        nodes = self.zk.get_children(self.election_path)
        nodes.sort()
        my_node = self.z_node_path.split("/")[-1]

        if my_node == nodes[0]:
            logging.info(f"{my_node} is the leader.")
            self.is_leader = True
        else:
            predecessor = nodes[nodes.index(my_node) - 1]
            logging.info(f"{my_node} is not the leader,
            watching {predecessor}.")
            self.watch_node(predecessor)
            self.is_leader = False
```

Listing 3.3: Leader Election Mechanism

3.4.4 Watch Mechanism

If the process is not the leader, it sets a Kazoo `DataWatch` on its immediate predecessor znode to be asynchronously notified if that node is deleted (e.g., due to the predecessor process crashing or disconnecting). In Kazoo, this is implemented using the `@zk.DataWatch()` decorator, which attaches a watcher callback function to the specified znode path. The callback is triggered whenever the node's data changes or the node itself disappears, enabling the process to react promptly and re-run the leader election.

```python
    def watch_node(self, predecessor):
        path = f"{self.election_path}/{predecessor}"

        @self.zk.DataWatch(path)
        def watch(data, stat, event):
            if stat is None:
                logging.info(f"Watched node {predecessor}
                disappeared. Re-electing...")
                self.elect_leader()
```

Listing 3.4: Watch Mechanism

3.4.5 Leader and Follower Tasks

The `run_election()` method is responsible for continuously reporting the client's role (leader or follower). It runs an infinite loop where it logs the status and could be extended to execute the actual distributed tasks depending on the role.

```python
def run_election(self):
    try:
        while True:
            if self.is_leader:
                logging.info(f"Process {os.getpid()} is the leader. Doing
                    leader tasks...")
            else:
                logging.info(f"Process {os.getpid()} is a follower. Waiting
                    for election...")
            time.sleep(5)
    except KeyboardInterrupt:
        logging.info("Stopping election...")
    finally:
        self.zk.stop()
        logging.info("Zookeeper client stopped.")
```

Listing 3.5: Leader and Follower Tasks

This structure clearly separates the coordination logic from the actual workload, making it easy to extend the program to implement specific leader or follower behaviors.

3.5 Execution and Results

To test the system, multiple instances of the application were executed concurrently.

At startup, the instance that created the znode with the smallest sequence number was automatically elected as the leader and performed leader tasks (displaying messages). All other instances acted as followers and set watches on their respective predecessor nodes.

When the leader instance was manually terminated (simulating a failure), ZooKeeper automatically removed its ephemeral znode and notified the followers.

The follower watching the deleted node detected the change, re-executed the leader election procedure, and became the new leader.

This behavior demonstrates the system's ability to maintain **high availability** and **automatic fault recovery** without manual intervention or system downtime.

3.5.1 Running the Application

To run the application, navigate to the project directory and execute the following command in separate terminal windows to simulate multiple distributed processes:

```
python zookeeper_election.py
```

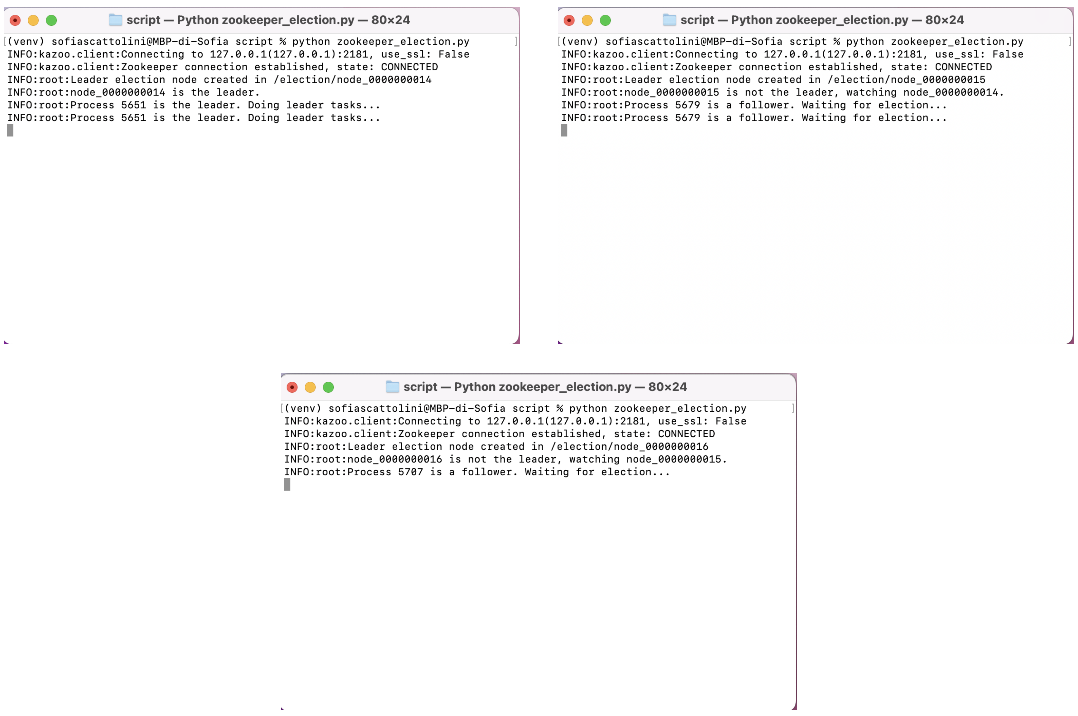
Each process will autonomously connect to the ZooKeeper server at `127.0.0.1:2181`, create its ephemeral sequential znode, and participate in the leader election protocol.

3.5.2 Observed Behavior

- One process became the leader and printed messages indicating leadership.
- Others remained in follower mode, watching their predecessors.
- When the leader was stopped, a new leader was elected without restarting the system.

Figures show:

- Initial state with one leader and two followers.
- Updated state after the original leader exited.

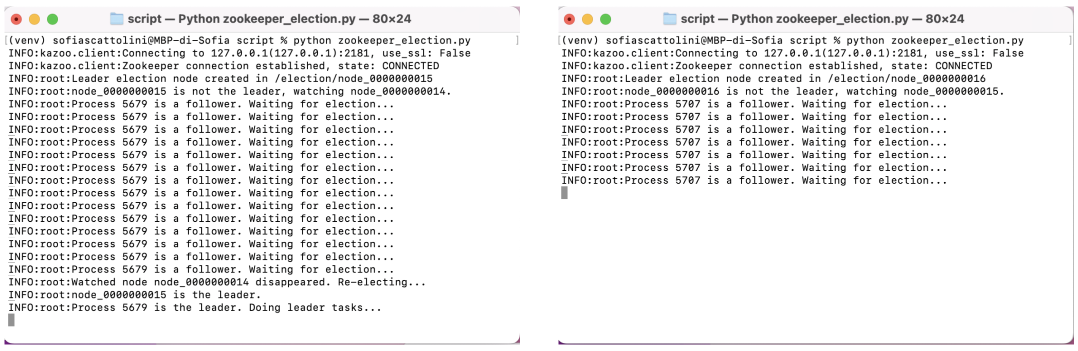

script — Python zookeeper_election.py — 80×24
script — Python zookeeper_election.py — 80×24
script — Python zookeeper_election.py — 80×24

```
(venv) sofiascattolini@MBP-di-Sofia script % python zookeeper_election.py
INFO:kazoo.client:Connecting to 127.0.0.1(127.0.0.1):2181, use_ssl: False
INFO:kazoo.client:Zookeeper connection established, state: CONNECTED
INFO:root:Leader election node created in /election/node_0000000014
INFO:root:node_0000000014 is the leader.
INFO:root:Process 5651 is the leader. Doing leader tasks...
INFO:root:Process 5651 is the leader. Doing leader tasks...
```

```
(venv) sofiascattolini@MBP-di-Sofia script % python zookeeper_election.py
INFO:kazoo.client:Connecting to 127.0.0.1(127.0.0.1):2181, use_ssl: False
INFO:kazoo.client:Zookeeper connection established, state: CONNECTED
INFO:root:Leader election node created in /election/node_0000000015
INFO:root:node_0000000015 is not the leader, watching node_0000000014.
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
```

```
(venv) sofiascattolini@MBP-di-Sofia script % python zookeeper_election.py
INFO:kazoo.client:Connecting to 127.0.0.1(127.0.0.1):2181, use_ssl: False
INFO:kazoo.client:Zookeeper connection established, state: CONNECTED
INFO:root:Leader election node created in /election/node_0000000016
INFO:root:node_0000000016 is not the leader, watching node_0000000015.
INFO:root:Process 5707 is a follower. Waiting for election...
```

Figure 3.1: Result with all 3 processes


script — Python zookeeper_election.py — 80×24
script — Python zookeeper_election.py — 80×24

```
(venv) sofiascattolini@MBP-di-Sofia script % python zookeeper_election.py
INFO:kazoo.client:Connecting to 127.0.0.1(127.0.0.1):2181, use_ssl: False
INFO:kazoo.client:Zookeeper connection established, state: CONNECTED
INFO:root:Leader election node created in /election/node_0000000015
INFO:root:node_0000000015 is not the leader, watching node_0000000014.
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Process 5679 is a follower. Waiting for election...
INFO:root:Watched node node_0000000014 disappeared. Re-electing...
INFO:root:node_0000000015 is the leader.
INFO:root:Process 5679 is the leader. Doing leader tasks...
```

```
(venv) sofiascattolini@MBP-di-Sofia script % python zookeeper_election.py
INFO:kazoo.client:Connecting to 127.0.0.1(127.0.0.1):2181, use_ssl: False
INFO:kazoo.client:Zookeeper connection established, state: CONNECTED
INFO:root:Leader election node created in /election/node_0000000016
INFO:root:node_0000000016 is not the leader, watching node_0000000015.
INFO:root:Process 5707 is a follower. Waiting for election...
INFO:root:Process 5707 is a follower. Waiting for election...
INFO:root:Process 5707 is a follower. Waiting for election...
INFO:root:Process 5707 is a follower. Waiting for election...
INFO:root:Process 5707 is a follower. Waiting for election...
INFO:root:Process 5707 is a follower. Waiting for election...
INFO:root:Process 5707 is a follower. Waiting for election...
```

Figure 3.2: Result after Deleting Process Leader

3.6 Challenges Faced

During the development of the application, the following challenges were encountered:

- Gaining a deep understanding of the behavior and properties of **ephemeral sequential znodes** in ZooKeeper, which are fundamental to the election mechanism.
- Correctly implementing and handling the **watch mechanism** to detect leader failures and promptly trigger the re-election process.
- Setting up the **Docker environment** and ensuring the Python application could successfully connect to the ZooKeeper container.

These challenges were overcome by studying the official ZooKeeper and Kazoo documentation and by performing multiple tests to verify the correct behavior of the system.

4. Conclusions

This project showed how a leader election algorithm can be implemented using Apache ZooKeeper together with the Kazoo library in Python. The final result is a simple prototype that can be used as a base to build more advanced and real distributed systems.

Even if ZooKeeper does not offer a built-in function for leader election, it offers the necessary primitives (such as ephemeral and sequential znodes) needed to implement one. With the help of watchers, the system can automatically react when the leader goes down and elect a new one without any manual action.

Kazoo made it easier to work with ZooKeeper in a Python environment, offering a clean API that follows the same logic as the original ZooKeeper commands. Thanks to this, the implementation was more readable and accessible.

This project also helped to better understand how ZooKeeper can be used not just for leader election, but also for other coordination tasks in distributed systems. Even if it is not as popular as it used to be, ZooKeeper still teaches important concepts like fault tolerance, consistency, and dynamic coordination.

Overall, this project was very useful to really understand how coordination works in distributed systems. Studying the theory is important, but hands-on implementation reveals issues and solutions that are not always obvious at first. It would be interesting in the future to extend this work and use ZooKeeper as part of a real distributed application.

References

[1] Sofia Scattolini. Zookeeper_leaderelection, 2025. URL https://github.com/Sophisss/ZooKeeper_LeaderElection.

[2] Apache Software Foundation. Apache zookeeper documentation, 2024. URL https://zookeeper.apache.org/doc/current/index.html.

[3] Kazoo Developers. Kazoo: A high-level python library that makes it easier to use apache zookeeper, 2024. URL https://kazoo.readthedocs.io/.

[4] Oracle Corporation. Java se development kit (jdk) downloads, 2025. URL https://www.oracle.com/java/technologies/downloads/.

[5] Python Software Foundation. Download python, 2025. URL https://www.python.org/downloads/.

[6] Kazoo Contributors. Kazoo — python zookeeper client documentation, 2024. URL https://kazoo.readthedocs.io/en/latest/.

[7] Inc. Docker. Docker — build, ship, and run any app, anywhere, 2025. URL https://www.docker.com/get-started.

[8] Apache Software Foundation. Zookeeper docker image, 2024. URL https://hub.docker.com/_/zookeeper.