École Polytechnique Fédérale de Lausanne

LPD Semester Project

Professor: Rachid Guerraoui

**Implementation of Group Membership**

Author: Supervisor:

Iva Najdenova Dragos-Adrian Seredinschi



Contents

[List of Figures 3](#_Toc480546793)

[1 Introduction 4](#_Toc480546794)

[2 About ZooKeeper 5](#_Toc480546795)

[3 Deployment of Distributed Queue 9](#_Toc480546796)

[o Implementation on a Single Node 9](#_Toc480546797)

[o Implementation on Multiple Nodes 10](#_Toc480546798)

[4 Modification of Curator's Group Membership Algorithm 12](#_Toc480546799)

[5 Deployment of Group Membership on Multiple Nodes 12](#_Toc480546800)

[6 Conclusion and Future Work 12](#_Toc480546801)

[7 References 12](#_Toc480546802)

# List of Figures

Figure 1: ZooKeeper's Hierarchical Namespace 5

Figure 2: ZooKeeper Service 6

Figure 3: Read and Write Requests 6

Figure 4: Configuration File for Standalone Mode 7

Figure 5: Configuration File for Replicated Mode 7

Figure 6: Session States and Transitions2 8

Figure 7: Adding Elements to a Distributed Queue (Standalone Mode) 9

Figure 8: Verification of Added Elements (Standalone Mode) 9

Figure 9: Removing Elements from the Distributed Queue (Standalone Mode) 10

Figure 10: Verification of Removed Elements (Standalone Mode) 10

Figure 11: In case of Empty Queue 10

Figure 12: Connecting to Different Servers in the Replicated Mode 11

Figure 13: Distributed Queue in Replicated Mode 11

# Introduction

One of the fields in computer science, which have recently gained enormous popularity is distributed computing. What makes distributed programming extremely challenging and different from the standard, centralized environment, is the need of coordination among many loosely-coupled components that cooperate. The questions arising from this area, are great in number, diverse in context, and most of the time influenced by surrounding factors like networking failures and delays. Owning to that, in this project we emphasize the practical aspects, together with a real implementation of such distributed environment.

To be more specific, we use the Apache ZooKeeper coordination service for distributed applications, which has bindings in both Java and C programming languages. ZooKeeper facilitates the development of applications by providing many useful functions and recipes like: name service, configuration, queues, barriers, locks etc.

Starting point of this project was installing the ZooKeeper software on a Linux system, and testing its basic functionalities. Furthermore, in this paper, we are going to discuss how to deploy one of the recipes - distributed queue, firstly on a single node and later on several nodes.

Subsequently, we reach to a point where the focus is set on our primary goal - implementation of a group membership abstraction. To that end, we use a Java/JVM client library for ZooKeeper, named Apache Curator. It provides a highlevel API framework and utilities, in order to make ZooKeeper more reliable and ease its usage.[[1]](#footnote-1)

After modifying and deploying the group membership algorithm, we test this implementation on multiple nodes. Moreover, we use the Apache Maven tool for building the project.

We conclude this paper with few remarks about the distributed computing field, which is prone to errors and failures of different kind and finally, we introduce some ideas about future usage of the group membership abstraction.

# About ZooKeeper

The need for cooperation among processes is the main cause of problems that have to be solved in the distributed computing field. In order to simplify this task of building robust distributed systems, Apache ZooKeeper has been designed around some core concepts and primitives. Its main objective is to provide an interface that is easy to understand and use by the developer, but even with this coordination service, the development of distributed applications is not trivial.[[2]](#footnote-2)

Some main characteristics of ZooKeeper are[[3]](#footnote-3):

* ***Simplicity****:*

The coordination of distributed processes is done through a shared hierarchal namespace, which consists of data registers - called **znodes** (similar to files and directories). Each znode might be associated with data and can also have children nodes. Furthermore, it contains an Access Control List (ACL) for restricting the users. It also exists a notion of ephemeral node, which disappears once the session is over. ZooKeeper can achieve high throughput and low latency because the data is kept in-memory (unlike a typical storage system).

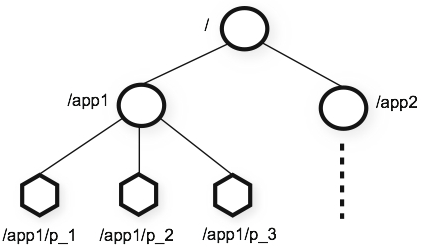


Figure 1: ZooKeeper's Hierarchical Namespace

* ***Replication****:*

Each server that constitutes the ZooKeeper service must maintain its own copy of the in-memory image of the state and be aware of the presence of every other server. This way of keeping replicas helps in case of unexpected termination of the connection, so that the client could easily transfer to a different server. Through the TCP connection, the client can send requests, get responses and watch events, or send heart beats. Every read request is dealt with the local replica of the corresponding server, whereas the write requests are forwarded to a single server, which is called **leader**.

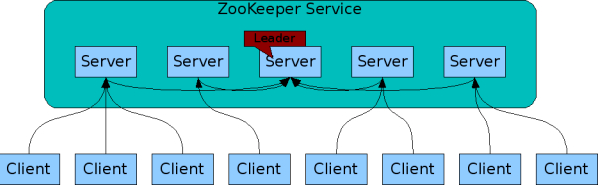


Figure 2: ZooKeeper Service

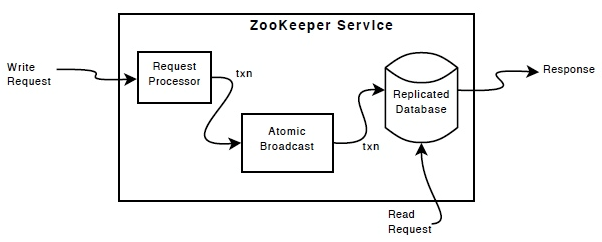


Figure 3: Read and Write Requests

* ***Order****:*

ZooKeeper keeps track of the order of all transactions, by putting a timestamp on each update, which can later be used for implementing higher-level abstractions, such as synchronization primitives.

* ***Guarantees****:*
  + *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 observes the same view of the service regardless of the server that it connects to.
  + *Reliability* - Once an update has been applied, it will persist from that time forward until a client overwrites the update.
  + *Timeliness* - The clients view of the system is guaranteed to be up-to-date within a certain time bound.[[4]](#footnote-4)

After getting familiar with the fundamental concepts of Apache ZooKeeper, follows the step of downloading the distribution from the official Apache website and setting up the proper configuration.

For the standalone mode, we needed to make changes in the **conf/zoo.cfg** file and provide correct values for:

* **tickTime** - used to do heartbeats and the minimum session timeout is twice this value
* **dataDir** - the location where the in-memory database snapshots, as well as the transaction log are stored
* **cleintPort** - the port which will be used to listen for client connections

The final outlook of the zoo.cfg configuration file is shown on the Figure 4.

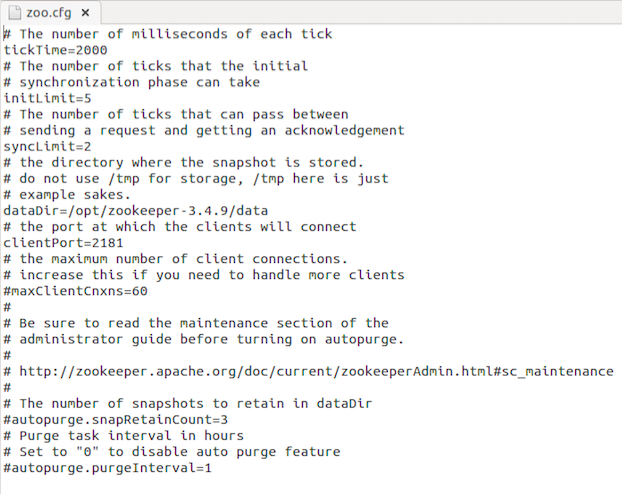
****

Figure 4: Configuration File for Standalone Mode

When it comes to running ZooKeeper in replicated mode, we first need to set up data directories for each server, where we'll keep the particular configuration file together with a, so called, **myid** file from which the server gathers information about his ID. What should be done next, is to create a separate configuration file for each server, where we will include the IP addresses of the servers together with the TCP port numbers used for quorum communication and leader election. An example of such configuration file for replicated mode is given on the following figure. 2

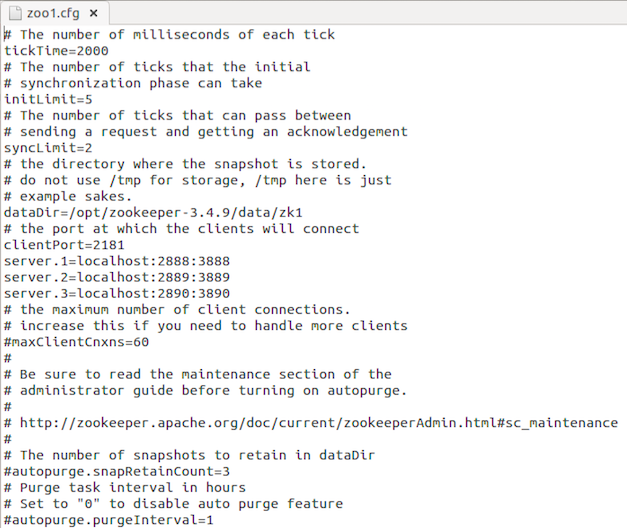


Figure 5: Configuration File for Replicated Mode

Once we have successfully set the ZooKeeper environment up, we can move on to starting the servers and clients by using the proper command line instructions. The service becomes available only if the majority of the servers work correctly. When a client tries to establish a session with one of the servers, there are several possible states: NOT\_CONNECTED, CONNECTING, CONNECTED and CLOSED. We can observe the probable transitions between these states on Figure 6.

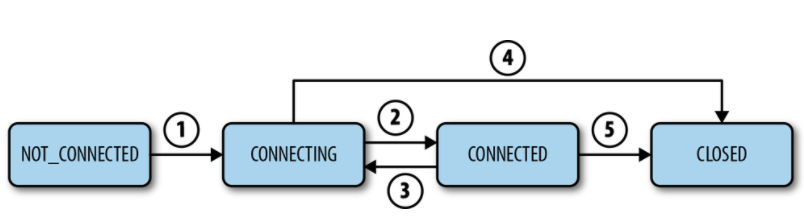
****

Figure 6: Session States and Transitions2

# Deployment of Distributed Queue

After verifying that the ZooKeeper service is running correctly, we are ready to go for an implementation of one of the provided recipes, which by choice was distributed queue. As it is a very common data structure, a generic code already exists and only few corrections were needed.

The idea behind the realization of this abstraction, is to designate a znode to hold the queue and then append children nodes with valid sequence numbers each time a client creates an element for the queue. Moreover, if some client decides to consume an element from the queue, then the znode with lowest timestamp will be deleted.

In order to accurately compile and run the Java program, it was needed to export several .jar files into the ZooKeeper CLASSPATH. Here we show the results from the deployment, for both standalone and replicated mode.

## Implementation on a Single Node

Before executing the distributed queue recipe, it is needed to successfully start the ZooKeeper server and it is also recommendable to launch one client in order to observe the changes made to the queue znode.

Firstly, in our example, we produce ten elements and check if they are genuinely created.

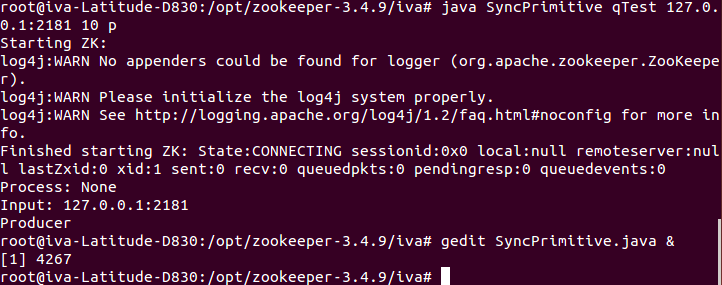
****

Figure 7: Adding Elements to a Distributed Queue (Standalone Mode)

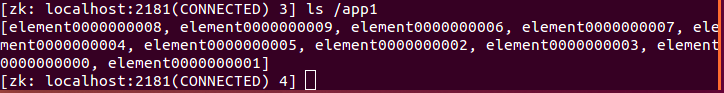


Figure 8: Verification of Added Elements (Standalone Mode)

Then, we show how some of these queue elements are being removed after invoking the same program, but this time as a consumer.



Figure 9: Removing Elements from the Distributed Queue (Standalone Mode)

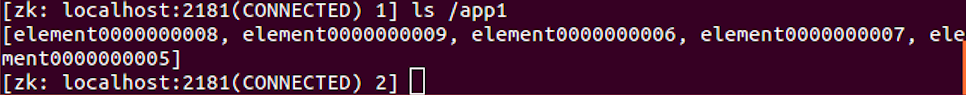


Figure 10: Verification of Removed Elements (Standalone Mode)

In case we try to remove elements from an empty queue, the consumer will have to wait until another client adds elements.

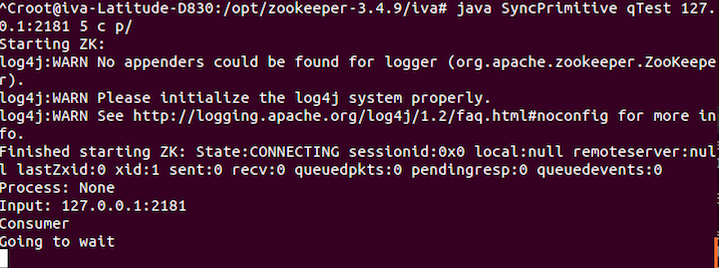


Figure 11: In case of Empty Queue

## Implementation on Multiple Nodes

The first step of deploying a distributed queue in replicated mode, is identical as in the standalone case: correctly starting the ZooKeeper servers that are going to be used to provide service.

When a client tries to establish a connection, there exists no rule about which server he is going to be assigned to. If we try to start and then terminate the session several times, we can see that we are not connecting to the same server every time. On Figure 12, it is possible to notice how two clients are establishing connections to different servers, yet they are still part of the same ZooKeeper service and have an equivalent view of the znode topology.

Later, in order to check how this environment with multiple servers and clients works, we ran the same example of adding ten elements to the distributed queue and then removing five of them. It can be spotted from the Figure 13, that regardless of the server we are connected to, the state of the queue is identical, and even if two clients try to consume elements at the same time, the consensus abstraction which is implemented by ZooKeeper, will not allow any inconsistency.

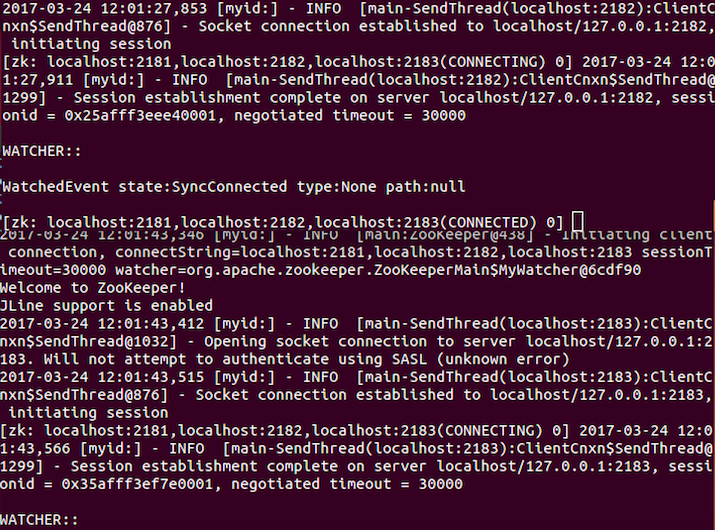


Figure 12: Connecting to Different Servers in the Replicated Mode

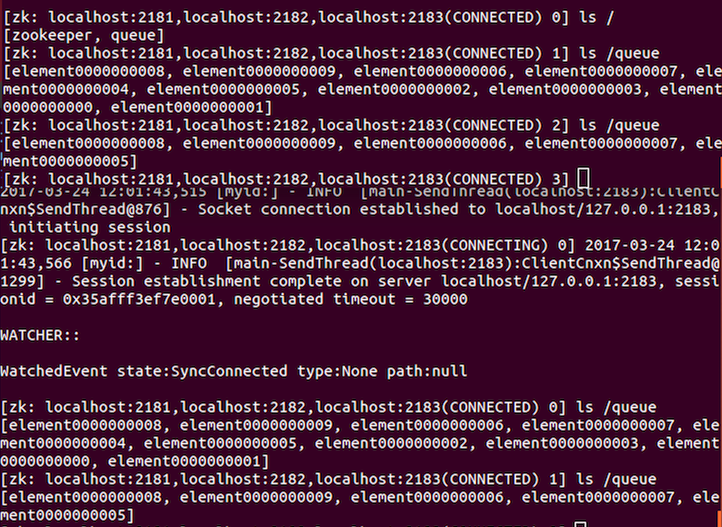


Figure 13: Distributed Queue in Replicated Mode

# Modification of Curator's Group Membership Algorithm

# Deployment of Group Membership on Multiple Nodes

# Conclusion and Future Work

# References

ZooKeeper by Benjamin Reed; Flavio Junqueira *Published by O'Reilly Media, Inc., 2013*

1. http://curator.apache.org/index.html [↑](#footnote-ref-1)
2. ZooKeeper by Benjamin Reed; Flavio Junqueira *Published by O'Reilly Media, Inc., 2013* [↑](#footnote-ref-2)
3. http://zookeeper.apache.org/doc/trunk/zookeeperOver.html [↑](#footnote-ref-3)
4. http://zookeeper.apache.org/doc/trunk/zookeeperOver.html [↑](#footnote-ref-4)