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](#_Toc484275233)

[1 Introduction 4](#_Toc484275234)

[2 About ZooKeeper 5](#_Toc484275235)

[3 Deployment of a Distributed Queue 9](#_Toc484275236)

[o Implementation on a Single Node 10](#_Toc484275237)

[o Implementation on Multiple Nodes 11](#_Toc484275238)

[4 Group Membership 13](#_Toc484275239)

[o Overview 13](#_Toc484275240)

[o Modification of Curator's Group Membership Algorithm 14](#_Toc484275241)

[o Deployment of Group Membership on Multiple Nodes 16](#_Toc484275242)

[5 Evaluating the Implementation of Group Membership 20](#_Toc484275243)

[6 Conclusion and Future Work 22](#_Toc484275244)

[Acknowledgements 23](#_Toc484275245)

[References 24](#_Toc484275246)

[Appendix 25](#_Toc484275247)

# List of Figures

Figure 1: ZooKeeper's Hierarchical Namespace3 5

Figure 2: ZooKeeper Service3 6

Figure 3: Read and Write Requests3 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: Problem with the Connection State 9

Figure 8: Adding Elements to a Distributed Queue (Standalone Mode) 10

Figure 9: Verification of Added Elements (Standalone Mode) 10

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

Figure 11: Verification of Removed Elements (Standalone Mode) 10

Figure 12: Case of Empty Queue 10

Figure 13: Connecting to Different Servers in the Replicated Mode 11

Figure 14: Distributed Queue in Replicated Mode 12

Figure 15: Starting 3 ZooKeeper servers 16

Figure 16: User Interface 16

Figure 17: Creating a group 17

Figure 18: Listing the existing groups 17

Figure 19: Perception of the ZooKeeper nodes 17

Figure 20: Joining a group and listing its current members 18

Figure 21: Limitation on the Number of Members 18

Figure 22: Notification for a Leaving Member 18

Figure 23: Age Limitation Possibility 19

Figure 24: Shell Script Usage Instructions 20

Figure 25: Example File used by the Script for adding Group Members 21

Figure 26: Proof that the Script works correctly 21

# 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 a real implementation of such distributed environment, together with practical aspects like coordinating the processes among the system.

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, queues, barriers, locks etc. These recipes have the purpose to simplify the design and implementation of distributed programs.

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. This abstraction is pertinent to group membership, as it can balance the load of messages between the participants.

Subsequently, we reach to a point where the focus is set on our primary goal - implementation of a group membership abstraction, whose importance is remarkable in many distributed applications that require every process to be aware which other processes are part of the system. 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](#_References)

After modifying and deploying the group membership algorithm (with the help from the Apache Maven tool for building and running the project), we test the validity of this implementation on multiple nodes, by using a shell script which initiates arbitrary number of group members.

In the end, we conclude this semester project by introducing some ideas about future usage of the group membership abstraction.

The structure of this paper, apart from introduction and conclusion, includes four main sections. In the first part, we cover the main characteristics of the ZooKeeper coordination service. Furthermore, in the second and third sections, we implement the distributed queue and group membership abstractions respectively. Finally, in the fourth division, we include details about evaluating our implementation of group membership.

# 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](#_References_1)

Some main characteristics of ZooKeeper are:

* ***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. If a client wants to be notified about a change in some znode, he needs to set a, so called, "watch". This watch is triggered and also removed when the znode is being modified. Furthermore, every znode 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).[3](#_References_19)

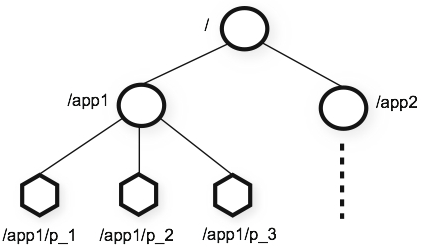


Figure 1: ZooKeeper's Hierarchical Namespace[3](#_References_15)

* ***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**.[3](#_References_20)

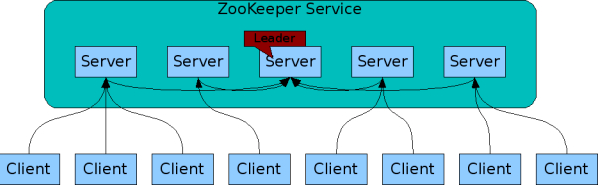


Figure 2: ZooKeeper Service[3](#_References_16)

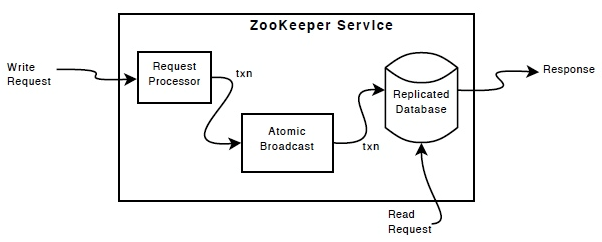


Figure 3: Read and Write Requests[3](#_References_17)

* ***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.[3](#_References_21)

* ***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.[3](#_References_3)

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*.

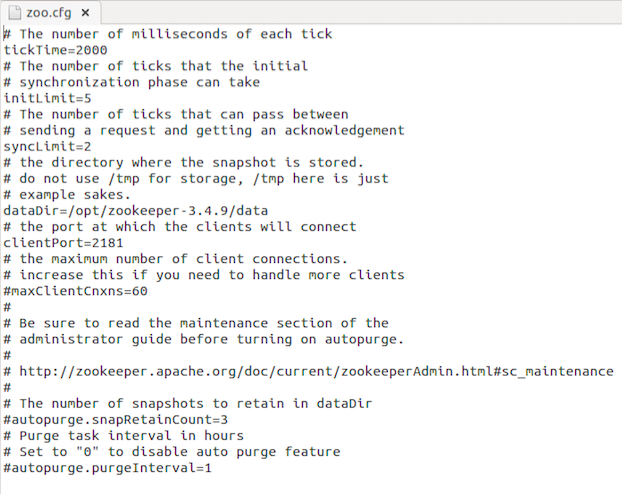
****

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](#_References_4)

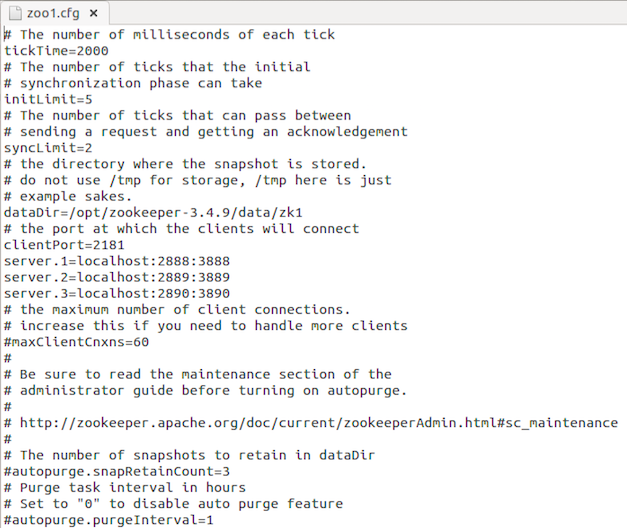


Figure 5: Configuration File for Replicated Mode

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*.

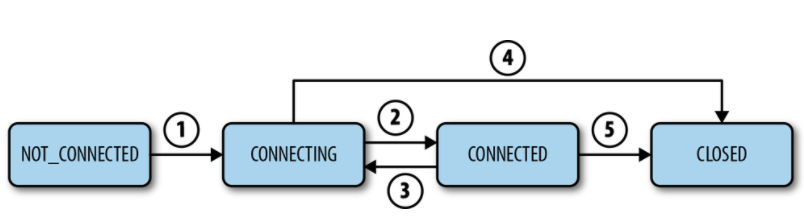
****

Figure 6: Session States and Transitions[2](#_References_12)

# Deployment of a 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 of relevance 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. However, we were still experiencing a trouble when our clients were trying to connect to the ZK servers, upon initiation through the Java program (*Figure 7*). Extremely surprising, was the fact that the command line clients had no problem in getting ZK service.

Since the state was never transferring for CONNECTING to CONNECTED, it led us to think that maybe there is something wrong with the TCP connection.

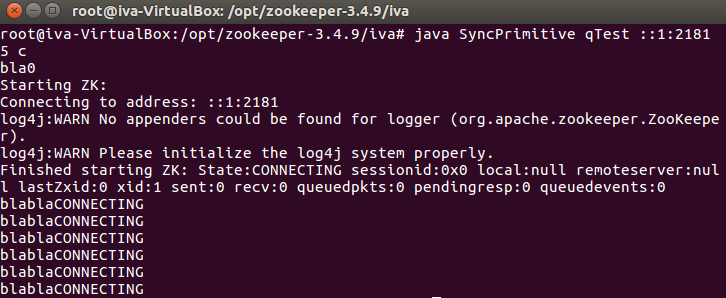


Figure 7: Problem with the Connection State

After series of troubleshooting, we came up with the idea that perhaps our Virtual Box has something to do with the wrong connection. From this reasoning, we decided to try the same Java program on a regular Linux OS and luckily our problem was solved.

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 "command line" 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.

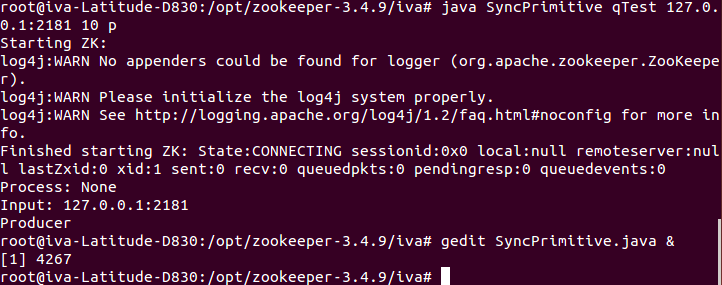
****

Figure 8: Adding Elements to a Distributed Queue (Standalone Mode)

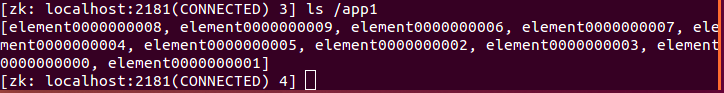


Figure 9: Verification of Added Elements (Standalone Mode)

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



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

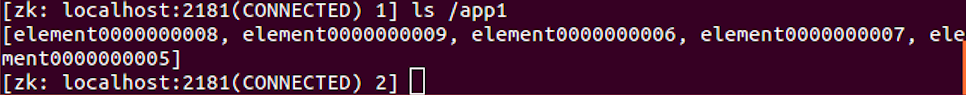


Figure 11: Verification of Removed Elements (Standalone Mode)

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

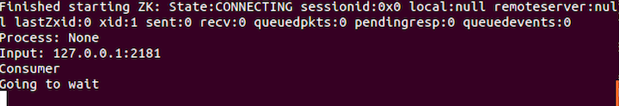


Figure 12: Case of Empty Queue

## 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 13*, 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 14*, 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.

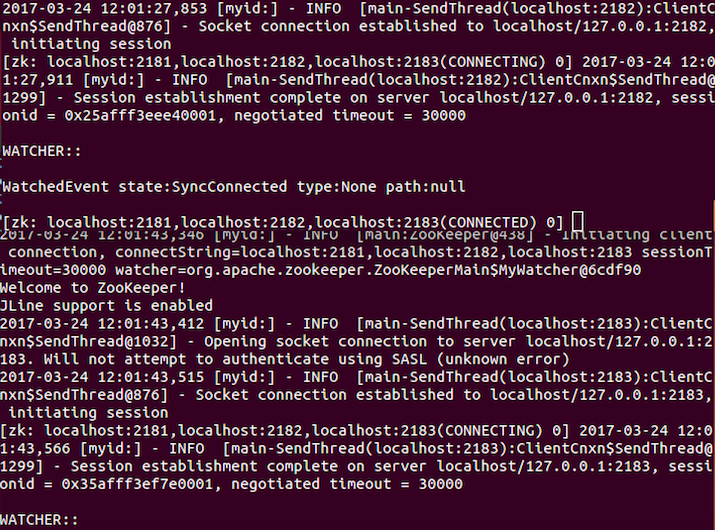


Figure 13: Connecting to Different Servers in the Replicated Mode

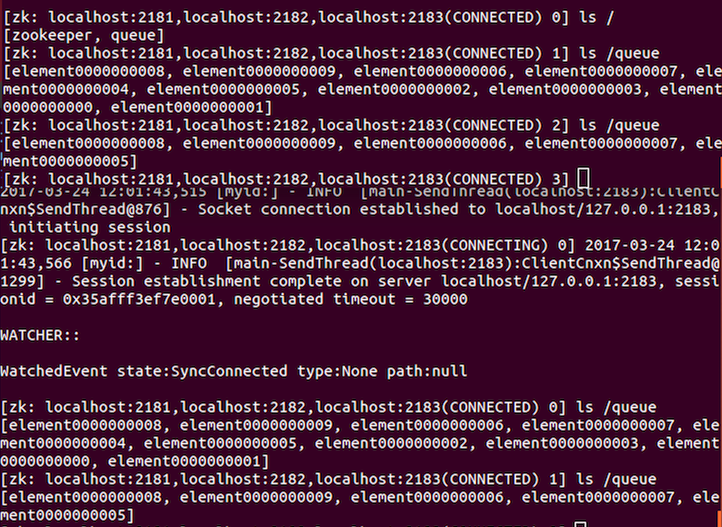


Figure 14: Distributed Queue in Replicated Mode

# Group Membership

*"A group membership (GM) abstraction provides consistent and accurate information about which processes have crashed and which processes are correct. It enables dynamic changes in the group of processes that constitute the system."*[*4*](#_References_5)

## Overview

There exist many distributed and communication oriented applications that require every process to be aware which other processes are part of the system. Providing this information is not an easy task, as the network is not always reliable and no timing assumptions (boundaries) hold. Some practical examples of this kind of applications are: multiplayer online games, virtual reality communities, aircraft real-time control systems etc.

One way of maintaining details about which processes participate, is using the perfect failure detector (distributed oracle that provides processes with suspicions about crashed processes). However, the output from this abstraction is not necessarily the same at different processes, i.e. processes may get failure notifications in different orders.[4](#_References_6)

In order to avoid that inconvenient situation, the group membership abstraction is used to coordinate the information not only about crashes, but also for joining and leaving operations, which leads to establishing a dynamic set of processes in the system.[4](#_References_7)

Providing consistent and accurate information about the members of the group is accomplished by installing group views V = (id, M) - a tuple of a unique numeric view identifier id and a set M of view member processes. A new view might be interpreted as a new composition of the system. The set of processes need to agree through consensus, which members are going to be part of the next view they are delivering.[4](#_References_8)

The following properties characterize a group membership abstraction, where the view changes are caused only by crashes (the inclusion of join and leave events requires some minor changes): [4](#_References_9)

* **Monotonicity:** If a process p installs a view V = (id, M) and subsequently installs a view V' = (id', M'), then id < id' and M ⊇ M'.
* **Uniform Agreement:** If some process installs a view V = (id, M) and another process installs some view V' = (id, M'), then M = M'.
* **Completeness:** If a process p crashes, then eventually every correct process installs a view (id, M) such that p ∉ M.
* **Accuracy:** If some process installs a view (id, M) with q ∉ M for some process q ∈ Π, then q has crashed.

## Modification of Curator's Group Membership Algorithm

Curator is a set of libraries that build on top of ZooKeeper, in order to facilitate its usage by managing the handle (connection) automatically and providing useful recipes for primitives such as locks, barriers, caches, queues and also for our primary interest in this project - group membership.[2](#_References_10)

The idea behind the GM recipe, is to have a znode for every group, and whenever someone wants to join a specific group, we append his unique identifier as a child node to that proper znode.

The algorithm for group membership, provided by Curator, made our implementation less complex, as it includes several constructors and methods like:[5](#_References_11)

* protected PersistentEphemeralNode newPersistentEphemeralNode(CuratorFramework client, String membershipPath, String thisId, byte[] payload) - creates new child ephemeral node for a given path from the provided identifier and payload
* protected PathChildrenCache newPathChildrenCache(CuratorFramework client, String membershipPath) - keeps locally cached data about every child of the given ZK path (responds to update/create/delete events)
* public GroupMember(CuratorFramework client, String membershipPath, String thisId, byte[] payload) - creates new member node in the given path, by using the above mentioned constructors for ephemeral node and cache
* public void start() **-** this method is invoked by the GroupMember object in order to initiate the new member node and to begin caching the members
* public void close() **-** this method is invoked by the GroupMember object in order to leave the group he is being part of
* public Map<String, byte[]> getCurrentMembers() **-** this method is invoked by the GroupMember object in order to get the current membership set

The first needed modification in Curator's algorithm, was replacing the PersistentEphemeralNode constructor with the new, more general PersistentNode, for which we made two versions: one for producing a persistent node (used when creating a new group) and another for producing an ephemeral node (used when creating a new member).

Another problem that we encountered was the node creation itself. Even though we were invoking the proper constructor and start method, there was no sign of existing znode. After we spent some time investigating what the problem could be, we found out that there is a race condition between the node creation and accessing it right after calling start. To solve this issue, we needed to include the method waitForInitialCreate(long timeout, TimeUnit unit) upon starting the node, as it blocks until either the node, initiated by start(), is created or the timeout elapses.[7](#_References_22)

In our algorithm, we additionally included a way to obtain a list of all available groups and keep a local cache of them in the form of HashMap. This HashMap links their real names, which start with a 128 bit UUID prefix (universally unique identifier) and the user-friendly names. This way, whenever a client wants to join a specific group, we don't need to access the ZooKeeper servers in order to discover the group's real name with the UUID, and therefore the time delays are reduced. Since the group members also need to be unique, their names start with an UUID prefix as well. However, we provided a more appropriate way of listing the members, showing only the names given by the users themselves.

What seemed not really practical in the Curator's algorithm was the need to represent the payload-data of the nodes as an array of bytes. Thus, we included proper conversion between String and byte[].

Apart from giving the users a possibility to join or leave a group, we also allow them to create a new one. When producing the new group, we construct a persistent node, because we want the group to continue existing even if the creator leaves, or his connection dies. This is not the case when a member node is being created, because we don't expect the user to continue participating in the group if his handle to ZooKeeper crashes or he willingly leaves the group. For that reason, we are using the constructor for ephemeral nodes in this situation.

When it comes to using the group membership abstraction in a distributed application, it is very convenient to receive notifications whenever a new user joins the group or an existing member decides to leave. Therefore, we included a method that "listens" to changes on a given znode, by using the PathChildrenCache class and the PathChildrenCacheListener interface, and informs every participant about any kind of modifications (CHILD\_ADDED, CHILD\_UPDATED, CHILD\_REMOVED).

As we mentioned before, every znode can be associated with data, that is usually in the byte to kilobyte range. This data, stored at each node, might cover different details like status information, configuration, location information, IP address or anything convenient for the application.[4](#_References_18) In our implementation, we decided to use the data of every group znode as a restriction about the maximum possible number of members in that particular group, and also as an indication whether the group allows minors to join. Additionally, we utilize the data of every member znode, to store the age of that participant.

In what follows, we are going to show the results from our implementation of the group membership abstraction.

## Deployment of Group Membership on Multiple Nodes

Getting our algorithm for group membership to work wasn't a simple task, as there are many Curator JAR files that need to be included with the help from Maven, a software project management and comprehension tool.[8](#_References_23)

We first decided to work in the Eclipse IDE, as it seemed less complex for adding the Maven dependencies, but later we also managed to compile and run our Java project through the command line.

Before executing our program, we need to start the ZooKeeper servers (in the same way we described before in this paper), and in this example we decided their number to be three.

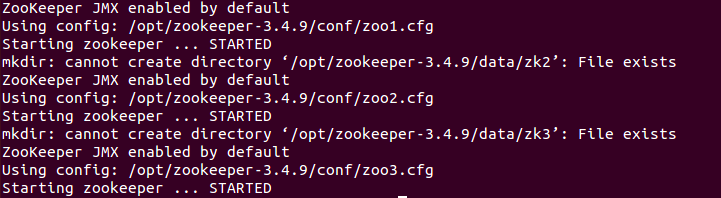


Figure 15: Starting 3 ZooKeeper servers

Furthermore, we use the command:

$ mvn -f /home/iva/workspace/GroupMember/pom.xml clean install

to build and run our Java program, and we get a user interface where three choices are available (to create or join a group, or to list the currently available groups), as shown on *Figure 16*.

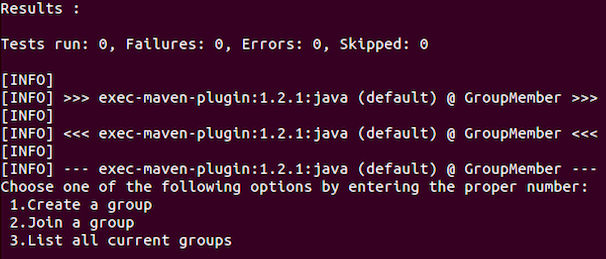


Figure 16: User Interface

On the following two pictures we are first creating a new group, named **"Tennis-Singles"** with maximum two members and no age restrictions, and then checking if it really exists by choosing the third option.

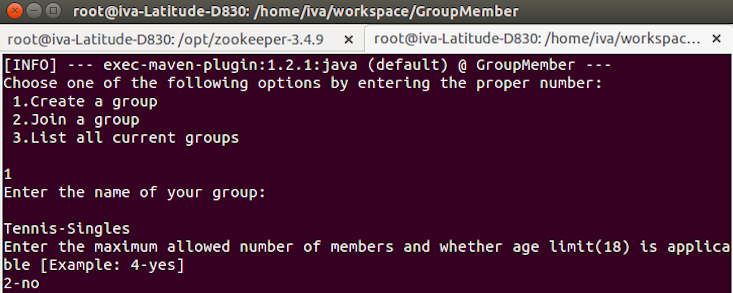


Figure 17: Creating a group



Figure 18: Listing the existing groups

If we want to observe the background hierarchy of the ZooKeeper nodes and their real names with an UUID prefix, we need to start a ZK client from command line and use the **ls / command**. This is shown on the following picture:

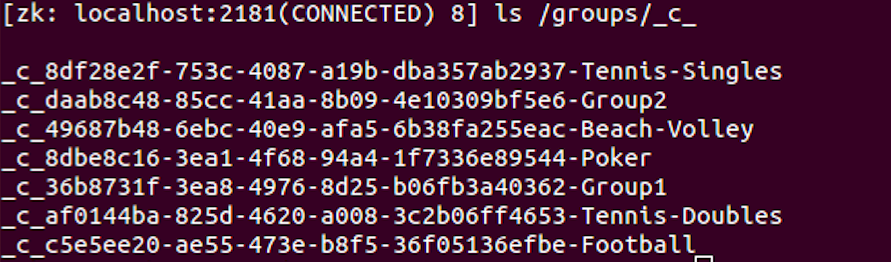


Figure 19: Perception of the ZooKeeper nodes

Now, we can move on to letting the members join the group. In this example, the users with identifiers Member1 and Member2 start participating in our "Tennis-Singles" group. They first give information about their unique ID and also age as a payload, and then they are provided with the possibility to leave the group or list the current members. As it can be seen on *Figure 20*, both users get the same membership view. It is very important to note that the users are delivering coordinated views thanks to the consensus functionality of ZK.

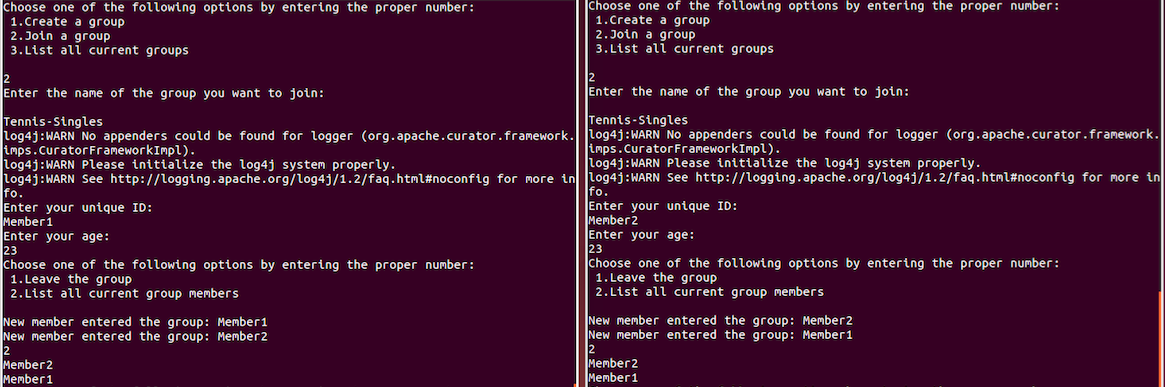


Figure 20: Joining a group and listing its current members

When a third user tries to enter this group, he receives a message that there is no available space (*Figure 21*).



Figure 21: Limitation on the Number of Members

On the following picture, we can observe how the group members receive a notification whenever some of the participants leave.

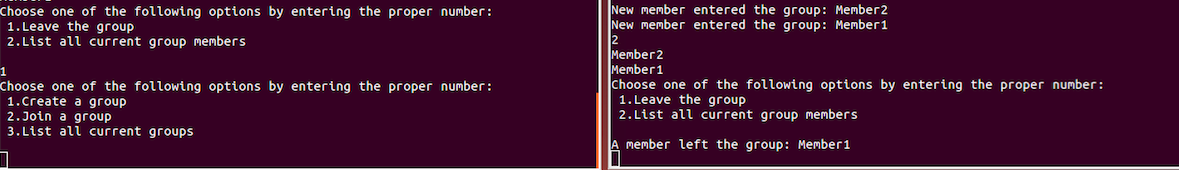


Figure 22: Notification for a Leaving Member

In order to test our age limitation possibility, we created a "Poker" group that doesn't let minors to enter it (Figure 23).

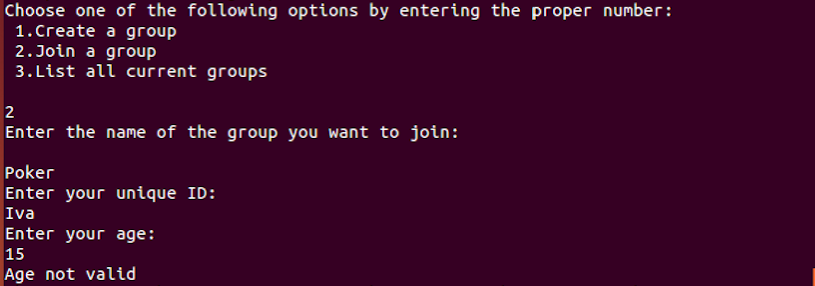


Figure 23: Age Limitation Possibility

Having reached a satisfactory implementation of the GM abstraction, we continue to evaluating it with arbitrary number of users.

# Evaluating the Implementation of Group Membership

After having deployed our algorithm, we were interested in testing the validity of this implementation of group membership with significantly higher number of users. To that end, we are introducing a shell script which can start a given number of ZooKeeper servers that will later provide service to variable number of clients.

Depending on the arguments we include when running the script, several options are possible. These usage instructions are shown on the following picture:

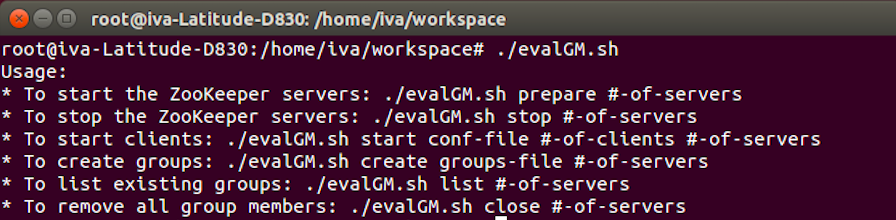


Figure 24: Shell Script Usage Instructions

The argument **"prepare"**, followed by a number, starts the ZooKeeper servers, by first producing the exact number of configuration files. The port numbers on which these servers listen for client connections, start from 2181 and increase by one for every next server. In case we want to terminate the ZooKeeper service, we need to use the argument **"stop"**, followed by the number of servers.

The main goal of this shell script was to cause many users to join their assigned group. When we execute the script with the argument **"start"**, each client reads from a textual file (provided as a second argument) what his identifier and payload is, and also the name of the group he needs to join. One example is given on *Figure 25*. The third argument is the number of clients we need to start, whereas the fourth is the number of servers that are running at the moment.

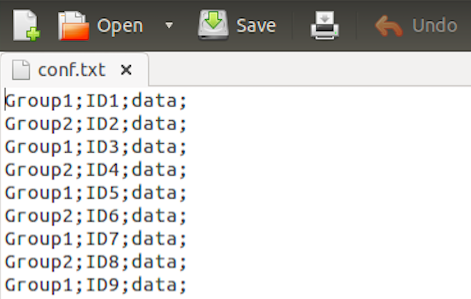


Figure 25: Example File used by the Script for adding Group Members

We also included a way to produce new groups through our script, by using the argument **"create"** and a textual file in which the new groups' names might be found. In case we need to take a look at all current groups, the argument **"list"** should be utilized, while for removing the members from every group we use **"close"**, followed by the number of running servers.

To make sure that our script is working correctly, we start a ZK client from a command line and list the content of Group1 or Group2, as according to the textual file, members were assigned to these two groups.

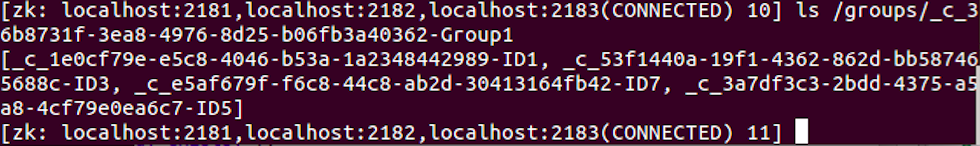


Figure 26: Proof that the Script works correctly

A better way to evaluate the validity is to improvise a real distributed environment with many users on distinct computer machines and compare their membership views. Unfortunately, due to resource limitations, we were unable to test our GM algorithm this way.

# Conclusion and Future Work

The aim of this paper was to present a real implementation of a distributed environment, which we know is prone to errors and failures of different kinds.

What makes distributed computing a nontrivial and challenging problem is the need to tolerate partial failures and also keep up with the asynchronous nature of the processes. Devising algorithms which ensure a consistent cooperation among processes, that will remain operating correctly and solve common tasks even after several crashes in the system, is not as easy task. [4](#_References_13)

In many distributed applications, several processes have to reach agreement in order to synchronize their actions and achieve a common goal. Therefore, solving the consensus problem leads to solving many other problems in the distributed computing field like: total order broadcast, atomic commit, terminating reliable broadcast. [4](#_References_14)

For our implementation of both distributed queue and group membership abstraction, we were using the ZooKeeper's consensus functionality. Significantly important assets of this coordination service, are the provided recipes which simplify the design and implementation of distributed programs.

Although we had some troubles with the connection state at the beginning of the deployment, we managed to successfully implement a distributed queue in both single and multiple node modes. The queue abstraction was of our interest because it might be used as a buffer for messages between members of a group.

The main focus of this project was set on implementing the group membership primitive and testing it with the help of the shell script we created. In spite of the fact that we observed positive results (coordinated membership views) from this deployment on a single Linux machine, we believe that the conditions are much more complicated when each member uses a separate computer, and also the network protocols add weight to this complexity.

As a future task, we consider using a more realistic distributed environment, where both clients and servers would run on distinct machines and the failure of one server-machine wouldn't mean termination of the service.

It would also be very useful to extend our algorithm to support delivery of messages which are coordinated with the installation of views - View Sinchrony ("A message is vsDelivered in the view where it is vsBroadcast").[4](#_References_24)

Another, more advanced, future work for this project would be devising a distributed application on top of our group membership implementation.

# Acknowledgements

I would like to thank Prof. Rachid Guerraoui and the LPD Laboratory for providing me with the opportunity to work on a project of my interest. Additionally, special thanks to my supervisor Dragos-Adrian Seredinschi for his guidance and help throughout the whole project.

# References

1. http://curator.apache.org/index.html
2. ZooKeeper by Benjamin Reed; Flavio Junqueira *Published by O'Reilly Media, Inc., 2013*
3. <http://zookeeper.apache.org/doc/trunk/>
4. Christian Cachin, Rachid Guerraoui, & Luıs Rodrigues, Introduction to Reliable and Secure Distributed Programming, Springer

1. <https://github.com/apache/curator/blob/022de3921a120c6f86cc6e21442327cc04b66cd2/curator-recipes/src/main/java/org/apache/curator/framework/recipes/nodes/GroupMember.java>
2. <https://www.youtube.com/watch?v=SyVZwjAxxIM>
3. <https://issues.apache.org/jira/browse/CURATOR-310>
4. <https://maven.apache.org/>

# Appendix

* <https://github.com/Iva1502/zookeeper>