University of Trento,Italy
Distributed System 1 Course Project, A/C 2016/2017

Distributed Key-Value Store with Data Partitioning

[Yoseph Mathewos Maloche (181406)]

August 12, 2017

Instructor : Prof. Gian Pietro Picco and Timofei Istomin

1 Introduction

This report explains about the algorithm we used and about each and every classes in detail.We tried to implement a DHT-based peer-to-peer key-value storage service. The system consists of multiple storage nodes and provides a simple user interface to upload/request data and issue management commands. It is a project with a reduced level of complexity, without the replication feature (N = W = R = 1), data versioning and support for recovery. So that, it doesn't support the mentioned features. In the following sections, we explained in detail about the algorithm and the techniques we used to implement.

2 Project description

Initially, we assumed we have six disconected nodes and one client. Then we assumed they are connected at the end of the process.The client can access any node at any time. In Here we do not have centralized administration; and each node is responsible for its own backup. Considering that When the client accesses a node for the first time, that node will be the coordinator for the new nodes in the ring. The nodes have their own ID number, port number, and IP address. The rings are arranged from the smallest to the largest ID number. The new nodes can join any time by sending a join message to the coordinator or any node in the ring joined previously. Join flag contains the IP address and the port number of the coordinator at the run time. After the nodes established the client can write on, read from or remove single node one after the other(It will support single request at a time). The implementation will be described in the next section.

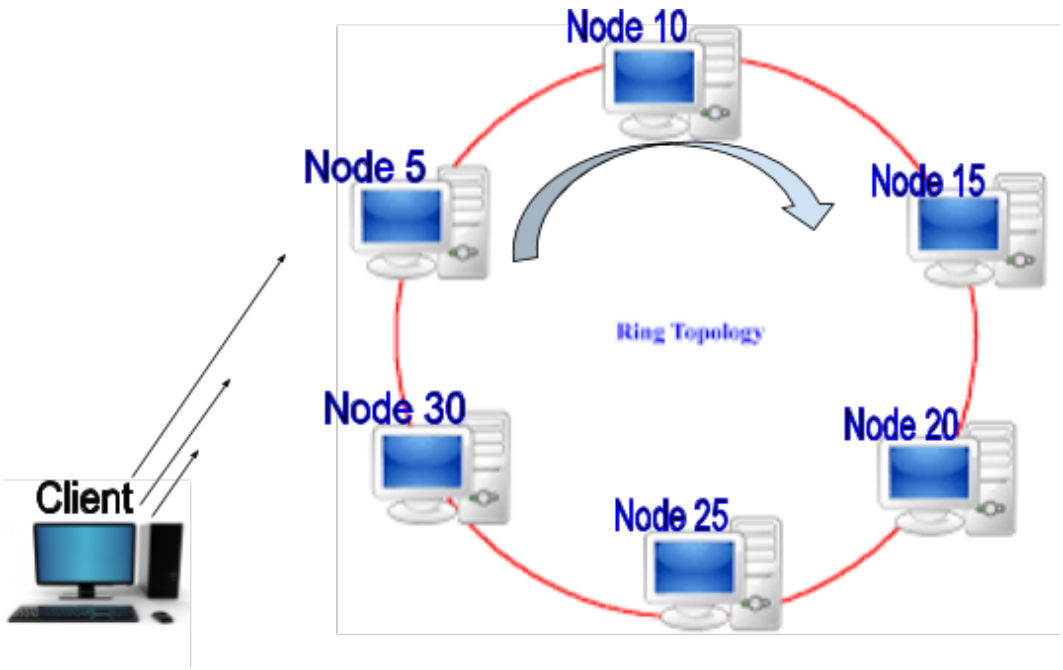

Figure 1: circular spaced nodes

3 Implementation Methods

The project implemented in Akka, with nodes being Akka actors. The protocol used for the system is Akka.remote.netty.tcp as transmission protocol which is a reliable protocol. We create class for Node and client separately. These node and client communicate via remote actor.

NodeApp Class :- This class contains a lot of classes, methods, and variable. Most of the classes inside NodeApp class are serializable. In Java, a class is serializable if and when it implements the Serializable interface, which is an interface residing in the java.io package. If a class is serializable, it means that any object of that class can be converted into a sequence of bits so that it can be written to some storage medium (like a file), or even transmitted across a network

The main method of this class does the "Join" task. First, it will check the length of the argument and if it is different from empty(0) and 3, the program will return with an error message. For example, if we use [join] [127.0.0.1] [10005] [something] this length doesn't fulfill the condition. Initially, the node can be a coordinator of a transaction or a general node. The first node called disconnected node, it doesn't need any input parameters because, the Ip address, port number and it's ID is already configured in its folder as "application.conf". It will start launching after it loads this file. At this point, the first node which is called disconnected node will stay in an idle status waiting for a new node.

Join class

This class used to allow the new nodes when joining the existing nodes. The new node can

Join the node by calling its IP address and port number. The message will be broadcasted to every node. When one of the nodes crashed again all of the nodes will be informed the crash of the node. When coordinator crashes immediately the next node will take the responsibility. Then the other nodes will continue their work without any problem. The crashed node can join anytime back to the ring. The crash detection mechanism is not requested to implement so, we won't care for the crashed nodes anymore.

ClientApp Class :- This class contains the tasks client can perform at run time. These tasks are read, write and leave operations.
Write:-The client can write any value(but 16bit unsigned) by using the command (Remote_IP, Remote_port, write command, key and the value). So in our case, the Ip is the Ip address of localhost and the other values will vary for each node. The Random 7 nodes are used$[5, 10, 15, 20, 25, 30, 40]$ with port number $[10005, 10010, 10015, 10020, 10025]$ respectively. The key and value are based on the client interest. When client writes a key with the value it should be written on the specific node since we have only one replication parameter and one write quorum. The write instruction follows update $--> $**[Key, Value]** concept.The requested node to write will display **The written data is : Key : Value** It will write the given value with it key and single requested node and the node will store in its local storage.

Read:-The client can request any node in the network with the key and the nodes will reply if the key belongs to them. The value will be accessed only from the written node by that key and the other nodes will not reply, so that timeout will happen. Read quorum follows the **get**$--> $**getValue** concept . The crash of the node will be informed only for the requested(communicated node).The reply for the client will be the read value is **"Value"**. The read requested node will display **"The key to read is Key"**
Leave:- The client can make the node go away from the ring by single "leave" command. The node which is leaving the ring will notify for the remaining nodes and the hashmap will be updated accordingly. The remaining nodes will print the existing nodes list as soon as the node/nodes leave the ring.
Local storage:-All the written files will be written to local disk storage at the same time to read history manually. Which means the local node will have history file.

4 Node and Client communication

In the client class we used Future to exchange message in between client and node. We need a way to contact a node in the Actor's System in order to request some operations. The normal way to do this in Akka is to start a new Actor System, start an actor, make the actor send a message to the designed node and wait for a response. Since this is a common use case and the standard approach is quite awkward, Akka provides the Ask Pattern to solve the problem. Akka takes care of sending a message to the target and waiting for a response.
For instance when client need to read from the node it will ask by this command and get the reply by future.

"Future<Object>request=Patterns.ask(myTargetNode,new NodeApp.ClientReadRequest(key1), timeout);"

5 Compile and Run commands

5.1 To run Node

Step1 : Change the directory to the directory the folder "MalocheGebremeskel" found, compline the java files by the following command

javac -cp $AKKA_CLASSPATH NodeApp.java ClientApp.java"

Step 2 : After you compile change the directory to one of nodes which has application.conf file and Run by using the following command

java -cp $AKKA_CLASSPATH:.:.. NodeApp

5.2 To Run Client

In the same directory to NodeApp which is already compiled on step one above Change the directory to ClientApp and Run the following commands for different cases
For Write
java -cp $AKKA_CLASSPATH:.:.. ClientApp Remote_ip Remote_port write Key Value
For read
java -cp $AKKA_CLASSPATH:.:.. ClientApp Remote_ip Remote_port read Key
For leave:
java -cp $AKKA_CLASSPATH:.:.. ClientApp Remote_ip Remote_port leave

6 Conclusion

We tried to implement Distributed Key-Value Store with Data Partitioning. We planned to implement full features of the project but we faced time constraints and lack of proficiency in java programming language. This project still needs much more additional features but, unluckily we are not able to manage more than this in our current situation