U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed and Partitioned Key-Value Store

LEIC - Computação Paralela e Distribuída 3º Ano - 2º Semestre

Turma 5
up201906834 - Óscar Gonçalo Martins Esteves
up201905396 - Pedro Magalhães Moreira Nunes
up201904979 - Luís Carlos Barros Viega

Distributed and Partitioned Key-Value Store

In this project, we developed a distributed key-value persistent store for a large cluster. The objective is to create a reliable means of storage at a large scale. It stores arbitrary data objects that are accessed by means of a key, similar to a hashing table. These values are stored in persistent storage and they are partitioned among various cluster nodes, and the design is somewhat based on Amazon's Dynamo, utilizing consistent hashing to partition the key-value pairs among the different nodes.

Membership Service

Leave and Join

Work with any amount of nodes, and when a node enters, it sends a message via IP multicast in the following format:

(the format is the same for join and leave)

join

<port> (port where node is accepting communications from other nodes)

<membership count> (even for joining and odd for leaving)

A node that receives this message will wait a random time up to 400ms and send a Membership message via TCP with its view of the cluster. A joining node will only accept up to 3 of these messages to prevent information overdose.

Starting the cluster

When a node is the first node of the cluster, it will send a join message but it won't receive any response. It tries to send the join message 3 times. If after 3 times the node does not receive any response it assumes he is the first one and initializes the cluster.

Periodic Membership Log

To avoid stale info, every second, the Leader Node sends a version of his Membership log via IP multicast. The other nodes trust this information because the leader is one of the oldest nodes and has up to date information.

Log Merging

With avoidance of stale info: Must include message format and description (how and why) of missing details (including fault-tolerance). Our log merging function is called “mixLogs()” , this function iterates over the received log and , using an HashMap<NodeId,MembershipCounter> of its own log, verifies if there are lines remaining , adding them, and verifying if the membership values of all nodes are lower than they are supposed to be, increasing them to the correct value if needed.

```
protected StringBuilder mixLogs(BufferedReader receivedLog) throws IOException {
    BufferedReader currentLog = StoreData.getMembershipLogBuff(StoreData.nodePort);
    HashMap<Integer,Integer> currentLogMap = StoreData.LogToMap(currentLog);
    StringBuilder newLog = new StringBuilder();
    String line = receivedLog.readLine();
    while (line != null && !line.equals("")) {
        String[] lineArray = line.split( regex: ";");
        String[] id = lineArray[0].split( regex: ":");
        int port = Integer.parseInt(id[1]);
        int membershipCount = Integer.parseInt(lineArray[1]);
        line = receivedLog.readLine();
        if(!currentLogMap.containsKey(port)) {
            newLog.append(StoreData.getLogLine(StoreData.nodeId,port, String.valueOf(membershipCount)));
            continue;
        }
        if(currentLogMap.get(port) > membershipCount ) {
            newLog.append(StoreData.getLogLine(StoreData.nodeId,port, String.valueOf(currentLogMap.get(port))));
        } else {
            newLog.append(StoreData.getLogLine(StoreData.nodeId,port, String.valueOf(membershipCount)));
        }
    }
    receivedLog.close();
    currentLog.close();
    return newLog;
}
```

Storage Service

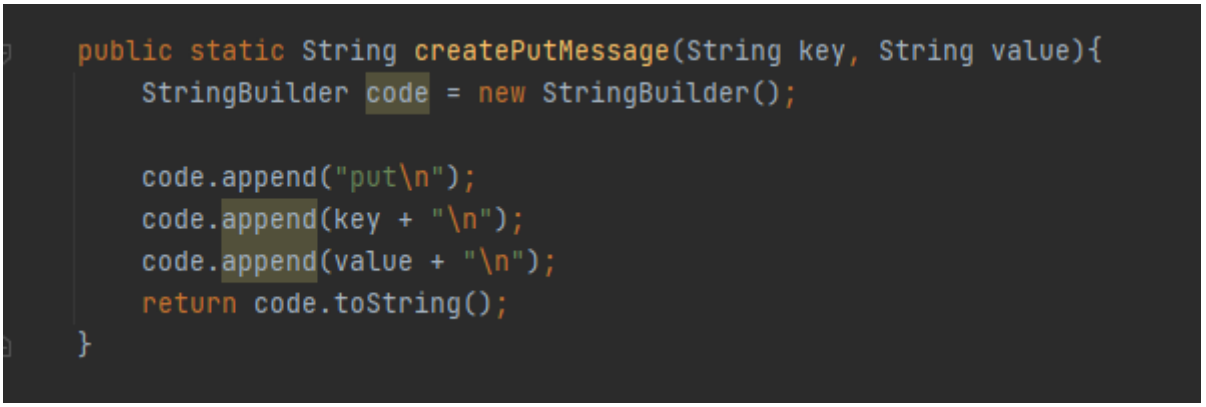
Message Format

The first line tells us which command we're running (put, delete, get, putreplica)

The second line tells us the key of the value we're dealing with. Those two consist of our header

The body is only used for put and putreplica. It holds the content we're saving

The following image is an example of a function we use to create a put message.



```
public static String createPutMessage(String key, String value){
    StringBuilder code = new StringBuilder();

    code.append("put\n");
    code.append(key + "\n");
    code.append(value + "\n");
    return code.toString();
}
```

Put

Put was successfully implemented according to the guide.

Delete

Delete was successfully implemented according to the guide.

Get

Get was successfully implemented according to the guide.

Putreplica

A put which will be only called on valid nodes, making the replication process more easier to handle. All the preprocessing is done before calling it. This command is only called by the store.

Hashing and Binary Search

A TreeMap was chosen since it holds a similar behavior to the one described in the guide.

There was also a function that allowed us to choose the node according to the guide - ceilingEntry(). (*The ceilingEntry(K key) method is used to return a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.).*

CeilingEntry outputs a valid node. Besides put, if no node is found, the methods stop. If a valid node isn't found on put and if there are nodes in the treemap, then the first node is used.

If the node where get and/or delete are called don't own the key for the file then ceilingEntry() is called and we parse the get/delete command to it. If there isn't any valid node then we conclude that the key is invalid and stop.

Features not implemented

Pair transfer and fault tolerance were not implemented. Thus if a node leaves or crashes all its content is lost, besides if it's saved with the replication implementation.

Replication

Our replication works in the following way: the node where we are storing our file has a log with the known nodes. We read that file and create a tree map. While creating it we read the number of known nodes. If the number of known nodes is under three, that value is taken as the replication factor. If not, three is taken as the replication factor.

During put we find all the nodes that are valid for a given key. With that we mean returned by the ceilingEntry() function. If no node is found, replication will not be taken regardless of the node factor. That happens because our get function will only be able to find nodes that are valid. Then putreplica is called, and since we already know it's a valid node it automatically inserts it in the node store.

If a node crashes, then we should be able to find the contents of said node in the rest of the cluster if replication was done. Since the replication is done only in values that are returned by ceilingEntry() our get function should be able to find all the correct nodes.

We weren’t able to implement the delete function perfectly at the time of writing this report. It deletes the file, but we didn’t add the replication factor to the function, so it never really knows how many nodes have said file stored.

Fault Tolerance

- Membership Counters as well as any Membership Log parts will always be stored in non-volatile memory.

- If a node is down for a long time it won’t be a problem for our Log merging algorithm as the node will receive the view of the cluster from a Leader Node.

Concurrency

Thread-Pools

Thread-pools were implemented and are used in the Store class.

This allows nodes to be able to process more than one request at a time.

```java
public class TCPServer implements Runnable{

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected InetAddress node_id = null;
    protected boolean      isStopped    = false;
    protected Thread       runningThread= null;
    protected ExecutorService threadPool = Executors.newFixedThreadPool( nThreads: 10);
    protected Boolean testClient;

    public TCPServer(int port, InetAddress node_id, Boolean testClient){
        this.node_id = node_id;
        this.serverPort = port;
        this.testClient = testClient;
    }

    public void run(){
        synchronized(this){
            this.runningThread = Thread.currentThread();
        }
        openServerSocket();
        while(! isStopped()){...}
        this.threadPool.shutdown();
        System.out.println("Server Stopped.") ;
    }
```

Implementation Details

When a server receives a message it will invoke a MessageHandler to decide what to do with that message. The message handler itself is an abstract class. It can either be a MessageHandlerTestClient or a MessageHandlerNodes, each of them will deal with messages from Nodes or TestClient.

```java
public abstract class MessageHandler {
    Socket clientSocket;
    PrintWriter writer; // if you want to answer to the received message write here
    InetAddress clientIp; // useful in debugging to know who you are talking to
    String id;
    int port;
    int replicationFactor;
    NodeStore store;
```

```java
public MessageHandler(Socket clientSocket) throws IOException {
    this.clientSocket = clientSocket;
    this.writer = new PrintWriter(clientSocket.getOutputStream(), autoFlush: true);
    this.clientIp = clientSocket.getInetAddress();
    this.id = String.valueOf(this.clientSocket.getInetAddress());
    this.port = clientSocket.getLocalPort();
    this.nodes = StoreData.nodes;
    this.store = new NodeStore(Utils.sha256(Integer.toString(this.port)));
    this.replicationFactor = 3;
    this.handleMessage();
}

protected abstract void handleMessage() throws IOException;
```

Conclusions

Seeing as an outage may have, in certain types of platforms, massive consequences, reliability is a very important requirement when building storage systems. Utilizing consistent-hashing as well as replication and thread-pooling, the result is a highly available means of data storage.