Distributed and Partitioned Key-Value Store

Computação Paralela e Distribuída

André Lino dos Santos - up201907879@edu.fe.up.pt
Edgar Ferreira da Torre - up201906573@edu.fe.up.pt
João Afonso Andrade - up201905589@edu.fe.up.pt

03/06/2022

Conteúdo

1 Introduction

The main objective of this project is to develop a distributed key-value persistent store for a large cluster which means that the data items in the key-value store are partitioned among different cluster nodes.

The key-value store works similarly to a hash table where each key-value store represents a simple storage system that stores arbitrary data objects (**values**) that can be accessed with **keys**, storing both of these in a persistent storage in order to ensure persistency.

Apart from the key-value store we also have as an objective the implementation of a **test client** that will allows us to invoke any of the **membership** events (join and leave) and any of the operations on key-value pairs (put, get and delete) which will be useful to test our key-value store.

2 Implementation

2.1 Membership Service

In order to achieve this feature we need to implement a distributed membership service and protocol that strives to keep the membership information at the nodes up-to-date, even in the presence of node crashes.

2.1.1 Message Format

Whenever a node joins the cluster it has to initialize the cluster membership. In order to do this some of the cluster members will send the new member a Membership message via TCP.

We decided to create the Membership message using the following code:

```
Stringbuilder message = new StringBuilder();
  message.append(this.nodeId).append(",").append(this.ipPort).
      red↪ append(",membership:");

  // append current existing nodes
  message.append(exisitingNodes.get(0));
  for(int i = 1; i < exisitingNodes.size(); i++)
    message.append(",").append(exisitingNodes.get(i));

  message.append(":");

  message.append(operationLog.get(0).nodeId).append("-")
    .append(operationLog.get(0).membershipNumber);
```

```
for (int i = 1; i < operationLog.size(); i++)
  message.append(",").append(operationLog.get(i).nodeId)
      .append("-").append(operationLog.get(i).membershipNumber);
```

In the end of the creation of the Membership message the final output should look something like this:

$$< \textbf{nodeId} >, < \textbf{storePort} >, \textbf{membership} :< \textbf{nodeId1} >, < \textbf{nodeId2} >: \\ < \textbf{nodeId1} > - < \textbf{membershipCounternodeId1} >$$

The first two components identifies who sent the membership message, and the next component is the membership tag, whose function is to let the TCP receiver know which operation is being transmitted.

After that, in the second part, an updated date list with all the nodes in the cluster (separated by commas) sorted by its nodeId's hash, is written, in order to update the receiver node. Lastly, the operation log is prompted in the form of a list of pairs (also separated by commas) with the node identifier and the respective membership number separated by hifen.

Contrary to the previous structure, the join and leave messages have quite a simple format. A simple operation name followed by the specific node identifier is enough for the other nodes in the cluster to update their list and add or remove the intended entry. Something like this will be prompted:

$$< \textbf{nodeId} >, < \textbf{storePort} >, < \textbf{operator} >, < \textbf{operator} >, < \textbf{operator} >, < \textbf{nodeId} > \quad (1)$$

```
String data = nodeId + "," + storePort + "," + membershipCounter
    red↪ + (char)0;
        try {
            DatagramPacket packet = new DatagramPacket(data.
                red↪ getBytes(), data.length(), group, ipPort);
            socketUDP.send(packet);
            socketUDP.joinGroup(group);
        } catch (IOException e) {
            e.printStackTrace();
        }
```

2.1.2 RMI

The interface is defined in the following link: https://git.fe.up.pt/cpd/2122/t09/g02/-/blob/main/assign2/src/MembershipOperations.java

2.2 Key-Value Store

The key-value store is implemented as a distributed partitioned hash table in which each cluster node stores the key-value pairs in a bucket.

The keys are generated using SHA-256 which is a cryptographic hash function. These keys will have lenght 64 and each byte of the hash value is encoded by the two ASCII characters corresponding to the hexadecimal representation of that byte.

```
public static String getSha(String input) {
    // Static getInstance method is called with hashing SHA
    try {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] hash = md.digest(input.getBytes(StandardCharsets.
            red↪ UTF_8));

        // Conversion from hash byte array to signum
            red↪ representation
        BigInteger shaResult = new BigInteger(1, hash);

        // conversion from sha numbers to string hex chars
        StringBuilder hexStr = new StringBuilder(shaResult.
            red↪ toString(16));

        // if string doesn't have the 64 chars, fill with 0's on
            red↪ left
        while (hexStr.length() < 64)
            hexStr.insert(0, '0');

        return hexStr.toString();
    } catch (NoSuchAlgorithmException err){
        return "";
    }
}
```

The use of this cryptographic function allows us to prevent any type of hash collisions in the keys.

2.2.1 Message Format

The message format of the key:value store opearation has a standard prefix that identifies the senders' nodeId and port. After that, there is a operand that identifies the command to be used, because not all have the same structure.

Get and delete operations are presented the following way:

$$< \textbf{senderNodeId} >, < \textbf{ipPort} >, < \textbf{operation} >, < \textbf{operand} > \quad (2)$$

While the format of the put operations is presented the following way:

$$< \textbf{senderNodeId} >, < \textbf{ipPort} >, < \textbf{operation} >, [< \textbf{operand1} >, < \textbf{operand2} >] \quad (3)$$

This put operation can have multiple operands because of internal key:value transfer operations on join/leave, where multiple values are removed from a node and stored in another. This procedure is taken because sending the keys while iterating over them would consume a lot of resources and time, being more susceptible to errors and crashes.

2.2.2 Redirection to correct node

When a key is inserted there is a high chance that the node who receives the request is not the pretended one. In this case, different approaches could be taken.

Obviously, we could answer back to the client with the correct node's address and port, and generate a new request. However, we opted by an approach in which these communications are cut short. When a node finds the id of the correct one, a similar request is generated directly from one node to another and the operation is fulfilled immediately. The described procedure may be seen in the following if block:

```
if (existingNodes.indexOf(this.nodeId) != correctNodeIndex) {
    sendTCPMessage(existingNodes.get(correctNodeIndex),
        storePort, message);
    System.out.println("Redirected␣message:" + message);
    return null;
}
```

2.3 Key-value Transfer on Membership Change

In case a membership change occurs, nodes may have to transfer keys to other nodes. This can happen with:

- a join event where the successor of the joining node should transfer to the latter the keys that are smaller or equal to the id of the joining node;
- a leave event where before leaving the cluster, the node should transfer its key-value to its successor

2.3.1 Join

When a node joins the cluster a join event happens where the node who succeeds the new node in the cluster transfers all the keys that are smaller or equal to to the id of the new node. This is done with the help of the following function:

```
public void transferKeysToNew(String newNodeId){
       String newKeyHash = Utils.getSha(newNodeId);
       System.out.println("TransferKeysToNew called");

       int index = (Utils.binarySearch(existingNodes, newKeyHash)
           red↪  + 1) % existingNodes.size(); // next to new id

       if (existingNodes.get(index).equals(nodeId)) {
           File dir = new File("StoreSystem/node" + hashedId);
           File[] dirList = dir.listFiles();
           if (dirList != null) {

               boolean willSendSomething = false;
               StringBuilder sb = new StringBuilder(nodeId).append
                   red↪ (",").append(storePort).append(",").
                   red↪ append("put");

               for (File file : dirList) {
                   if (!file.getName().equals("membership") &&
                       red↪ file.getName().compareTo(newKeyHash)
                       red↪ <= 0) {

                       willSendSomething = true;
                       String value = FileSystem.readFile("
                           red↪ StoreSystem/node" + hashedId + "/"
                           red↪  + file.getName());
                       if (value == null) return;
                       sb.append(",").append(value);
                       FileSystem.deleteFile("StoreSystem/node" +
                           red↪ hashedId +"/" + file.getName());
                   }
               }

               if (willSendSomething)
```

```
                sendTCPmessage(newNodeId, storePort, sb.
                    red↪ toString());
            }
        }
    }
```

Here, firstly, we look for the index of the node that succeeds the new node using binary search. If the that node is the current node we create a new File in order to store the keys from the next node in the new node via TCP and then we delete them from the older node.

2.3.2 Leave

When a node leave the cluster a leave event happens where the corresponding node transfers its key-value to one who succeeds it. We do this using the following snippet of code:

```
        int nextIndex = (binarySearch(hashedId) + 1) %
            red↪ existingNodes.size();
        transferKeysToNext(existingNodes.get(nextIndex));
```

As we can see in the snippet above, when a node leaves, we look for the one who succeeds it by looking for the one who has the succeeding index using binary search and the *hashedId* of the leaving node. After we find it we call the function $transferKeysToNext$ that receives the new node and sends to it the key-value pairs of the leaving node via TCP.

```
public void transferKeysToNext(String ipAddr){
        File dir = new File("StoreSystem/node" + hashedId);
            File[] dirList = dir.listFiles();
            if (dirList != null) {

                boolean willSendSomething = false;
                StringBuilder sb = new StringBuilder(nodeId).append
                    red↪ (",").append(storePort).append(",").
                    red↪ append("put");

                for (File file : dirList) {
                    if (!file.getName().equals("membership")) {
                        willSendSomething = true;
                        String value = FileSystem.readFile("
                            red↪ StoreSystem/node" + hashedId + "/"
                            red↪  + file.getName());
                        if (value == null) return;
                        sb.append(",").append(value);
                        FileSystem.deleteFile("StoreSystem/node" +
                            red↪ hashedId + "/" + file.getName());
```

```
                }
            }

            if (willSendSomething)
                sendTCPmessage(ipAddr, storePort, sb.toString()
                    red↪ );
        }
    }
```

2.4 Replication

One of the vulnerabilities of the implementation mention in the above steps is that if a node by any reason goes down or becomes unavailable, makes its key-value become unreachable. This way the program becomes vulnerable to losses of information.

To prevent this and in order to increase availability we should replicate key-value pairs (3 times) which means that each key-value pair should b stored in 3 different cluster nodes. Now if any of them go down the key-value pair can be obtained from any of the other two.

The only problem with this approach is that every time a node gets deleted or joins it has to be performed in all 3 of the attributed nodes which implies constantly performing join and leave operations.

2.5 Concurrency

2.5.1 Thread-Pools

In order to allow our program to process multiple requests at the same time our implementation is based on a thread-pool which maintains multiple threads waiting for tasks to be allocated for concurrent execution.

```
new Thread(new Runnable() {
    @Override
    public void run() {
        ExecutorService executor = null;
        try {
            executor = Executors.newFixedThreadPool(5);
            while (true) {
                final Socket socket = socketTCP.accept();
                executor.execute(new Runnable() {
```

Although it comes with a few disadvantages such as not being able to control the priority and the state of each thread, using thread-pools allows us to have

a better performance, save time and prevent the need to be constantly creating new threads.

2.5.2 Asynchronous I/O

Although we didn't manage to implement this feature we did some research on it in order to correctly comprehend the impact it would have on our program.

The main problem is that every time a thread is being used to perform an I/O operation it gets blocked and waits until the I/O operation is completed to unblock and continue the program. This way of running the program can be very inefficient because while the thread waits for the operation to be completed it could be doing other processing that does not directly depend on the I/O operation being executed.

With the thread-pools we would have multiple threads working at the same time while one was blocked with an I/O operation (waiting for it to be completed). With a Asynchronous I/O we can have a single thread trigger the I/O operation and do something else useful while the operation is running, reducing significantly the number of threads.

3 Conclusion

One of the main things we took away from this project is how important distributed systems are in our daily life. When developing this project we were able to get a better understanding on the how's and why's of every tiny specification and protocol that build up the whole system structure.

The most complex aspect of the whole project was the membership implementation and generally beginning the development. Since initially there was not much information about the protocol and summing these issues with the problems regarding the TCP/UDP messages, this step proved to be quite the challenge. After addressing these situations, setting the key-value interface and their transfer between nodes was not really a big problem because the setup was already done.

After that, our main focus went to giving our program the tools to avoid failure or crash situations, which led to some code changes and refactoring that provided us more confidence in our development.

Overall, even though building up this distributed system simulation brought us many obstacles, we ended up overcoming those situations using our methods and ideas. There's a general feeling that more could have been implemented we more time, but we consider the final result a really positive work.