University of Porto - Faculty of Engineering

Bachelor in Informatics and Computing Engineering

Parallel and Distributed Computation

Distributed and Partitioned Key-Value Store

Miguel Rodrigues (up201906042@up.pt)
Sérgio Estêvão (up201905680@up.pt)
Sofia Germer (up201907461@up.pt)

3 de Junho de 2022

Contents

Listings

1 Introduction

The aim of this project was to develop a distributed key-value persistent store for a large cluster. Distributed and Partitioned Key-Value Store is a standard in a high-performance storage system. This type of Store utilizes a distributed architecture to achieve significant performance and scalability advantages compared to traditional databases since the different data key-value items are partitioned among different cluster nodes.
This project is an approach to this Store system, done in Java programming language, loosely based on **Amazon's Dynamo**, using consistent hashing to partition the key-value pairs among the different nodes, using multicast to messages to ensure all cluster nodes are updated with the cluster's status, RMI interface to execute association and dissociation of nodes in a cluster, thread-pools to ensure concurrency in the Store's operations and a foulproof key-value system that secures the database operations.

2 Service

In this section we will summarize the services mentioned in the project's specification. For each one of these services has its respective interface that specifies the available operations regarding that specific service.

These services, are the membership service and the key-value service. The first defines a protocol, responsible for managing which nodes are active within a given cluster, i.e. a set of computers that store the same set of information in a coordinated way. On the other hand, the key-value defines how a client can interact with the cluster.

2.1 Key-Value Service

This service provides an interface that allows the user/customer to interact with a given cluster based on the following operations:

- `void put(String key, String value)`

  Adds key-value pair to the cluster.
  This operation receives a path to a file in the client's machine and stores the contents of that file inside the cluster. It returns a key, which is the hashed contents of that file, that can be used later to either retrieve those same content or delete them from the cluster.

- `File get(String key)`

  Returns a value associated to key.
  This operation receives a key, returned by a put() call, and gets the contents mapped to that key. If the key is not found in any of cluster's nodes an error is thrown.

- `void delete(String key)`

  Deletes a value associated to a key.
  This operation receives a key, returned by a put() call, and deletes the contents mapped to that key. If the key is not found in any of the cluster's nodes a warning is issued to the client.

2.2 Membership Service

This service provides an interface that allows to manage a given cluster's constitution. It has the following operations:

- `void join() throws RemoteException`

  Adds a node to a given cluster and performs the initialization of that same cluster if that is the case. The initialization of the cluster is a special procedure that occurs while there are less than 3 nodes present in the cluster.

- `void leave() throws RemoteException`

  Removes a node from a given cluster.

2.3 Service Invocation

java Store <IP_mcast_addr> <IP_mcast_port> <node_id> <Store_port>

where:

- <**IP_mcast_addr**>: Address of the IP multicast group used by the membership service
- <**IP_mcast_port**>: Port number of the IP multicast group used by the membership service
- <**node_id**>: Node's id (unique in a cluster)
- <**Store_port**>: Port number used by the storage service

3 Membership Service

The membership Service specifies the cluster membership service and respective protocol that must be provided by every cluster, handling the "join" and "leave" operations of nodes.
When a node joins the cluster, it initializes the cluster membership.

3.1 Message Format

Internally all the Messages are represented by an abstract class. Each message has a type and a body. Both join and leave messages also have a counter, that is increased by one each time the node joins/leaves the cluster.

```
1  package org.feup.cpd.store.message;
2
3  public abstract class Message {
4
5      protected final static String CRLF = "\r\n";
6
7      protected final String type;
8      protected final StringBuilder body;
9
10     protected Message(String type) {
11         this.type = type;
12         this.body = new StringBuilder();
13     }
14
15     public abstract String getContent();
16 }
```

Listing 1: Message

```
1  package org.feup.cpd.store.message;
2
3  import java.util.Queue;
4  import java.util.Set;
5
6  public class MembershipMessage extends Message {
7
8      public MembershipMessage(Set<String> view, Queue<String> events) {
9          super("MEMBERSHIP");
10
11         body.append("VIEW").append(CRLF);
12         for (String element : view)
13             body.append(element).append(CRLF);
14
15         body.append("LOGS").append(CRLF);
16         for (String event : events)
17             body.append(event).append(CRLF);
18     }
19
20     @Override
21     public String toString() {
22         return type + CRLF + body;
23     }
24
25     @Override
26     public String getContent() {
27         return body.toString();
28     }
```

```
}
```

Listing 2: Membership Message

3.2 Key-Value Message Format

```
<GET | DELETE>
<node_ap> <key>

PUT
<node_ap> <key>-<value>
```

3.3 Membership Log

A cluster node updates its view of the cluster membership and adds the respective event (either join or leave) to a membership log every time it receives a join or a leave message. Each record in this log includes only the node's id and the value of the membership counter.

Example of Membership Log:

```
content = [MEMBERSHIP, VIEW, 127.0.0.1:13000, 127.0.0.1:12000, LOGS, 127.0.0.1:12000 0]
content = [MEMBERSHIP, VIEW, 127.0.0.1:13000, 127.0.0.1:12000, LOGS, 127.0.0.1:12000 0]
content = [MEMBERSHIP, VIEW, 127.0.0.1:13000, 127.0.0.1:12000, LOGS, 127.0.0.1:12000 0]
127.0.0.1:12000 is now a part of 224.0.0.1:9000
node view = [127.0.0.1:13000, 127.0.0.1:12000]
```

3.4 RMI: Remote Method Invocation

We used the RMI as the transport protocol for membership operations. With RMI, we call a method on a local object – the proxy object, – which does not execute the method, but sends a method execution request to the remote object.
RMI is a Java API that performs remote method invocation with support for direct transfer of serialized Java classes.

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Membership extends Remote {
    void join() throws RemoteException;
    void leave() throws RemoteException;
}
```

Listing 3: Membership Interface

```java
try {
    MembershipOperation op = new MembershipOperation(pool, cluster, node);
    Membership membership = (Membership) UnicastRemoteObject.exportObject(op,
   node.getAccessPoint().getPort());
    Registry registry = LocateRegistry.createRegistry(node.getAccessPoint().
   getPort());
    registry.rebind("Membership", membership);
} catch (IOException e) {
    e.printStackTrace();
}
```

Listing 4: RMI Registry Creation

3.5 Election Algorithm

When the leader leaves the cluster we apply the election algorithm, which objective is to find the node with the most recent logs.
A more efficient way to solve this problem would be to use the **Round Robin** algorithm. This algorithm would basically iterate over all nodes using the UDP transmission protocol. Thus, a node would know if it was the most recent if it received its own logs.

```
if (node.getView().isEmpty()) {
    node.addNodeToView(node.getAccessPoint().toString());
    node.setLeader(true);
}
// Some more work happens in here...
if (node.isLeader()) {
    Thread leader = new Thread(new LeaderMulticastSender(cluster, node));
    leader.start();
}
```

Listing 5: Cluster leader election process

4 Storage Service

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

Consistent Hashing
The key-value store uses consistent hashing to partition the key-value pairs among the nodes in the cluster. This, allows to resize the hash table: change the number of buckets without remapping all the keys.

```
public String findKeyValueLocation(String key) {
    if (view.size() == 1)
        return ap.toString();
    List<String> clock = new ArrayList<>();
    for(String node : view.keySet())
        clock.add(node);
    Collections.sort(clock, compareBySHA);
    for(String node : clock) {
        if (key.compareTo(view.get(node)) < 0) {
            return node;
        }
    }
    return clock.get(0);
}
public String locateKeyValue(String key) {
    if (bucket.containsKey(key)){
        return ap.toString();
    } else {
        // Locates node within the "ring" and performs binary search
        return findKeyValueLocation(key);
    }
}
```

Listing 6: Functions for returning the node of a given key

4.1 Key-Store Message Format

```
PUT
127.0.0.1:8000 <key>-<value>

GET
127.0.0.1:8000 <key>

DELETE
127.0.0.1:8000 <key>

RETURN
127.0.0.1:8000 <task> <answer>
```

Listing 7: Key-Store Message Format

4.2 Calibration Upon Membership Changes

Upon a membership change, nodes may have to transfer keys to other nodes:
join event- 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
leave event- before leaving the cluster, i.e. multicasting the LEAVE message, the node should transfer its key-value pairs to its successor.

5 Replication

To increase availability, we could replicate key-value pairs with a factor of 3, which means that each key-value pair is stored in 3 different cluster nodes (in 3 different servers).
Before using replication we had only 1 node responsible for each file, and after its implementation we would have 3 nodes responsible for the file.
Arbitrarily, we could define a "sequence" of 3 nodes where the first node would be the node where the file would be placed normally and the second and third nodes would be the replicas.
The sockethandler structure would have to be changed in order to put in 3 nodes.
In the case of the put function, in its header we would add an attribute that would include the factor attribute: 0 in the case of being the "original", 1 and 2 in the case of being the replicas. In the case of the delete function, if the factor is 0, we would replicate the delete to the nodes with factor 1 and 2.
In the case of get, it would be the simplest function, in that it would search all nodes and return the first one it found, without the need to check the factor.

Considerations to have upon the replication:

- The actual number of copies created upon a put can be smaller than 3.
- Execution of operations can happen in different order for different replicas
- Node missing a delete operation and later try to replicate the deleted key-pair on the other replicas, upon realizing that the number of copies of the pair is lower than the replication factor

6 Fault-Tolerance

A Fault Tolerance Scenarios that may lead to failure:

- a node is down for a long time and it misses many membership events, the periodic MEMBERSHIP messages may not be enough for the node to learn the current membership of the cluster
- membership view of the node to which the client sent a request not being up-to-date and, therefore, the operation request will be sent to the wrong node

7 Concurrency

Concurrency increases performance on servers and usability on clients. Its goal is to overlap I/O with processing and take advantage of multiple cores.
In the case of our project, we used concurrency so that a node could be able to process several requests at the same time.

7.1 Thread-Pools

Our implementation of concurrency was based on thread-pools. In a Thread Pool each thread can execute more than one task, so a thread pool is an alternative to creating a new tread for each task that needs to be executed.
In Java, the *JavaExecuterService* interface represents an asynchronous execution mechanism which is capable of executing tasks concurrently in the background. There is a thread pool shared between

the UDP and TCP port listeners, which performs the decoding of the messages.

```
// Thread pool instantiating
this.pool = Executors.newCachedThreadPool();

// A new thread is instantiated to handle the incoming operation
pool.execute(new OperationDecoder(node, content));
```

Listing 8: Thread Pools

7.2 Asynchronous I/O

Event driven design decreases the number of threads because threads are only instantiated when there is data ready for reading, i.e. the thread never blocks while executing. Asynchronous I/0 means that it is not necessary to create new threads because there are blocked threads

8 Test Client

The Test Client was developed to test the key-value store and it invokes the membership operation handler (**handleMembershipOperation**) if the operation is "join" or "leave", and invokes the key value operation handler (**handleKeyValueOperation**) if the operation is "put", "delete" or "get".

To invoke the Test Client:

java TestClient <node_ap> <operation> [<opnd>]

where:

- <**node_ap**>: IP address and the name of the remote object providing the service (because we are using RMI)
- <**operation**>: Specifies the operation the node must execute (join/leave/put/get/delete)
- [<**opnd**>]: Argument of the operation, which is only used for key-value operations.
  In case of "put" it represents the file pathname with the value to add. In the case of "get" and "delete" it represents the string of hexadecimal symbols encoding the sha-256 key returned by put, as described in the next section.

9 Technical Aspects

This project was developed with **Java 11 LTS**. The only dependencies are the packages from the **Java SE packages**. As an additional note, and in order to automate our build, this project was built with the help of **Intellij**, nevertheless it is also possible to compile out project via the terminal.

10 Challanges Faced

However, we believe that some things did not go so well. Firstly, this project is of a higher complexity than it should be given the time we were given to develop it. Moreover, it took place during a very complicated period of college, where all the curricular units demanded a lot from us.
Something that also delayed code development was the fact that the three parts of the project (one for each element) were unbalanced, and clearly the first was much more complex. With the

remaining parts being dependent on the first, it became complicated to follow the code division model.
Finally, we believe that if the project documentation had been clearer and more straightforward, we would have been able to produce a much larger amount of code, since much of the time was invested in understanding what was expected of us.

11 Conclusion

Finishing this project, it is possible to say that we have learned the main concepts of distributed computing.
Distributed systems are the skeleton of real-time systems such as the Internet.
In addition, we have expanded our knowledge regarding large-scale databases of the key-value type and how this affects our daily lives that are not always perceptible to the end user.

References

[1] https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[2] https://dl.acm.org/doi/pdf/10.1145/2342356.2342360

[3] https://aws.amazon.com/builders-library/leader-election-in-distributed-systems/