Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing
2nd Project Report

Henrique Sousa, Mateus Silva, Melissa Silva

Teacher Jorge Barbosa

1 Membership Service

Since the goal of the project is the development of a distributed and persistent key-value store for a large cluster, it's required that every node knows every other node inside the cluster. To accomplish this, the nodes run a distributed membership service and protocol.

The Membership Service supports two distinct operations: *join()*, which adds a node to the cluster, and *leave()*, which removes a node from the cluster.

Every time a node chooses to join or leave the cluster, a specific message detailing which operations it would like to do must be sent.

1.1 Message Structure

1.1.1 Join & Leave

Each message has four arguments:

- The operation declaration (either *join* or *leave*);
- An identifier of the node who will perform the operation;
- The membership counter;
- The local port of the TCP socket.

The membership counter value helps determine which operation occurred – an even value means the node just **joined** the cluster and that it **left** the cluster otherwise.

1.1.2 Membership

These messages are sent in response to the *join* message the cluster nodes receive by an outside node, so their structure is different from the one used for *join* and *leave* and one of the several types of messages made using the *Message* class, each type having both a header and a body.

The header only includes the message's subject, *MEMBERSHIP*. The body includes information from the *MembershipData* class: the 32 most recent membership events in the membership logs and a list of the current members in the cluster. This information is translated into a string which is then sent.

1.2 Membership Log

Upon receiving a *join* or *leave* message, the cluster node adds an entry in the membership log specifying the last action.

Each entry includes the operation value (*join* or *leave*), the node's *id* and the value of the membership counter.

```
private void logJoin(String id, int counter)
{
    Log log = new Log(Ops.JOIN, id, counter);
    this.replaceLog(log, counter);
```

Note: Code edited to fit formatting (explicit class invocation).

1.3 Message Handling

The transfer of membership information is done using TCP.

A new member sends the request to *join* message using this protocol and the cluster members send information back in the same way. These don't, however, send it right away, opting instead to wait

a **random** time length before doing so.

A connection between a new node and the cluster members is broken after reception of 3 *MEMBERSHIP* messages.

In case this trio of information never arrives, the new node repeats transmission of the *join* message for, at max, two more times, making it a total of **three attempts** at getting a response from the cluster.

1.4 Stale Membership Information

Outdated nodes will not provide confirmation to a joining node.

A node calculates if they are outdated or not — this happens when a node hasn't received a membership update for 1.5 times the periodicity of the membership update *multicast* messages.

1.5 Initialization + Extra Feature

The very first node joining the cluster will not be able to receive any membership message. Thus, for any node that does not receive the required number of confirmations to join the cluster, initialization is made as if it is the only node in the cluster.

Beyond being a neat solution for the cluster creator, it also addresses problems when a node was not able to receive the required confirmations: the node will still join the cluster and, as membership updates arrive, the node will eventually reach the current state of the cluster.

Additionally, to create a more controlled environment, we provide another operation to the test client: a *create* operation that allows instant creation of the cluster without any confirmation hindrances.

1.6 Fault Tolerance

1.6.1 Clusters Under 3 Members

When a node joins a cluster with less than 3 members, there will not be 3 confirmations: *MembershipData* is analyzed for each of the confirmations received.

If the number of nodes indicated by the confirmations is under 3, then the joining node will adjust the needed number of confirmations.

Even though with just 3 required confirmations this problem is reduced, for larger amounts, this solution provides an adaptable way to have a seamless joining procedure.

1.6.2 Membership Updates

To ensure that all the nodes are away of the updated cluster status, we adopted a *brute-force* approach.

What we mean by this is that every node will periodically share their status to the whole network. Due to the incremental nature of the logs, the status is a monotonic function that will only vary towards the most up to date state.

Missing some of the membership update messages is not critical since the node will be able to use other nodes to update its status.

2 Storage Service

A normal execution will have the following steps:

1. In case of a *put*:
   (a) *Client* sends a PUT message to a node with the name of the file and the file path;
   (b) *Node* hashes the filename, obtains the node the operation must be sent to and sends a CLUSTER_PUT to that node.
   (c) The other node receives the request and processes it, storing the file in memory and obtaining the storage key. It then does two things:
      i. First, it replies to the first node with a PUT_RESPONSE;
      ii. Next, it informs its neighbors of the operation with a PUT_INFORM;
   (d) Finally, the first node that was contacted receives the PUT_RESPONSE. From the request number, it is able to identify the socket of the client where the request came from and send the hash back to them.

The process for other operations is analogous. **Page 5** includes the message structure used for all operations.

2.1 Fault Tolerance

2.1.1 *Get* Operation to Crashed Node

It might happen that a *get* request cannot be fulfilled.

This can happen for two reasons:

1. The node has crashed before the node requesting a file has had time to register the change;
2. The node was able to establish a TCP connection, but the node crashed in the meantime.

To guarantee that requests under these situations are fulfilled, we implemented a timeout in the request: if there is a timeout, the node will ask the neighbors of the crashed node for the needed information. Since keys are replicated, they will be able to provide it.

3 Replication

Replication is possible in only two circumstances: when a key-value pair is updated, or when a node leaves or joins the cluster.

Considering the existence of replication and multiple ownership of a key-value, we defined the original owner as the node whose hash is closest to the one of the key-value - this is the node described in **Chapter 4** of the project description.

3.1 Key-Value Update

Key-value pairs are updated whenever a *put* or *delete* operation happens; however, there might be a difference whenever a node receives either operation.

The node that receives the operation might be the original owner, in which case it will update its neighbors.

Otherwise, the node receiving the operation is one of the neighbors receiving the update.

```
// Is the node the original node?
Boolean act =
!(op.equals(RequestHandler.PUT_INFORM)
|| op.equals(RequestHandler.DELETE_INFORM)
|| op.equals(RequestHandler.GET_INFORM));
//Do the operation
mem.del(hash);
//If is the original node then inform
    neighbors
if (act) {
    message.setHeader(0,
        RequestHandler.DELETE_INFORM);
    this.informNeighbours(message);// Inform
        prev
}
```

Note: == was used instead of *.equals()* for formatting purposes.

4 Membership Updates

Any change within the cluster causes replication of keys. In case of a *join*, each node who is already in the cluster evaluates if the joining node is its neighbor.

In that case, it will replicate its original keys [1] to the newly joined node.

There will be no attempt at removing replicas that are now outdated.

```
if (nodeData ==
    membershipData.getPrev(data.getAddress()))
{
    this.nodeReplicateOwnKeysTo(data);
}
else if (nodeData ==
    membershipData.getSucessor(data.getAddress()))
{
    this.nodeReplicateOwnKeysTo(data);
}
```

Note: == was used instead of *.equals()* for formatting purposes.

In case of a *leave*, each node in the cluster will evaluate if the leaving node is its neighbor. In that case, then it will replicate its original keys to the **second neighbor** of the leaving node, hence *"replacing"* (in practice) the leaving node.

```
NodeData prev =
    membershipData.getPrev(leavingAddress);
NodeData suc =
    membershipData.getSucessor(leavingAddress);
if (nodeData.equals(prev)) {
    // If we are its predecessor
    // Send to its sucessor
```

[1] As in keys originally owned by the node.

```
    this.nodeReplicateOwnKeysTo(suc);
} else if (nodeData.equals(suc)) {
    // If we are its sucessor
    // Send to its predecessor
    this.nodeReplicateOwnKeysTo(prev);
}
```

Additionally, the leaving node will also replicate its original keys to **three nodes**: their new original owner and its two neighbors.

```
NodeData prev =
    membershipData
    .getPrev(nodeData.getAddress());
NodeData suc =
    membershipData
    .getSucessor(nodeData.getAddress());
NodeData nextSuc =
    membershipData
    .getSucessor(suc.getAddress());
this.nodeReplicateOwnKeysTo(prev);
this.nodeReplicateOwnKeysTo(suc);
this.nodeReplicateOwnKeysTo(nextSuc);
```

Note: Line breaks added for formatting purposes.

5 Concurrency

5.1 *Thread Pools*

Overall, we have a single *thread pool* to which tasks are added accordingly. At the time of writing, it uses six total threads.

There is a thread assigned to each of the two open sockets (one UDP and one TCP) to handle the reception of messages. The threads will constantly listen on the respective socket, waiting for a packet or message. Upon receiving either one, it will dedicate a new thread on the pool to process the received information.

We chose this approach since it resembles having a single dispatcher where handling of any outside connection does not warrant delays, since the thread is always listening for new information: for example, when the TCP socket is open, we'll prepare to receive information on it first.

```
Socket socket = null;
try {
socket =
    this.connectionMembershipProtocolSocket
        .accept();
} catch (IOException e) {
    e.printStackTrace();
}
```

```
InputStream input = null;

try {
    input = socket.getInputStream();
} catch (IOException e) {
    e.printStackTrace();
}

BufferedReader reader = new
    BufferedReader(new
    InputStreamReader(input));
```

Note: Line breaks added for formatting purposes.

Then, the reader will be permanently trying to assemble a valid message, the structure of which has been described previously.

```
while (true) {
   Message message = new Message(reader);
}
```

If it succeeds, the handling of the message will be handed to the *membershipProtocol* which will create another thread.

```
this.node.threadPool.submit(() -> {
     this.processMessage(message);
});
```

The process is the same for the three threads handling the reception of information through a socket.

6 Fault Tolerance

6.1 Message Sent to Wrong Destination

Every operation sent through TCP features the node that the message was (originally) sent to. If the receiving node notices that they aren't the indicated destination, the message will be redirected to the right one.

7 Conclusion

This was a very challenging project, especially in all the testing made for it - we invested some time into figuring out the best way to test our code and automate this process for efficiency purposes.

We did have a chance to apply concepts learned in theory classes, specifically, the class lectured by Prof. Fahed Jubair.

Operation	Header	Message
Get (from client)	GET	*Hash*
Get (relayed by node)	CLUSTER_GET, DESTINATION_NODE, REQUEST_NUM	*Sending node, Hash*
Get response (response to *get*)	CLUSTER_GET_RESPONSE, NODE_STRING, REQUEST_NUM	*File*
Put (from client)	PUT	*Name, File*
Put (relayed by node)	PUT, DESTINATION_NODE,	*Sending node, Name, File*
Put inform (key update)	PUT_INFORM, DESTINATION_NODE, REQUEST_NUM	*Sending node, Name, File*
Put response (response to *put*)	PUT_RESPONSE, SENDING_NODE, REQUEST_NUM	*Hash*
Delete (from client)	DELETE	*Hash*
Delete (relayed by node)	CLUSTER_DELETE, DESTINATION_NODE	*Sending node, Hash*
Delete inform (key update)	DELETE_INFORM, DESTINATION_NODE	*Sending node, Hash*