U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing 2021/2022

T09G08

Adriano Soares up201904873@fe.up.pt
Filipe Campos up201905609@fe.up.pt
Francisco Cerqueira up201905337@fe.up.pt

Index

1. Membership Service

1.1. Description

The Membership Service is responsible for the **distribution** of the cluster nodes and should be able to handle operations such as node **insertion** and **deletion** and to **find** the node responsible for a **key**.

1.2. RMI

The **Node** class implements the **MembershipService** interface that contains the **join**, **leave** and **view** methods. This interface is bound to the RMI Registry when a **Store** is invoked.

1.3. Implementation

To implement this service, we created a class **Node**, which is exclusively responsible for the implementation of this service. It implements the **join**, **leave** and **view** methods.

1.3.1 Join operation

The **join** operation starts by verifying the connection **status** of the cluster, if it's already connected it'll ignore the action. Then, after opening the TCP connection to receive the membership information messages on a **different thread**, it'll send the **JOIN** multicast message (in the main thread) **up to three times**. This connection will be opened on the port storagePort+1, in order to avoid sending unnecessary requests to the storage service. After receiving the three messages needed, it opens the TCP connection to receive the key store operations, it sends a **JOINED** multicast message (analogous to an ACK message) warning the **remaining nodes**, and updates its **status** to connected.

The node can easily determine the membership cluster through the TCP responses it receives since they contain the list of nodes currently present in the cluster.

For every log entry present in the membership information TCP response **we merge with the local membership log** by following these set of rules:

- Rule 1: If the event is already in the local log, skip it;
- Rule 2: If the event is older (verified through the membership counter) than the event for that node in the local log, skip it;
- Rule 3: If the event is new, i.e. there's no event in the local log for that member, add that event to the tail of the log;

- Rule 4: If the event is newer than the event for that node in the local log, remove the event for that member from the local log, and add the newer event at the tail of the log.

1.3.2 Leave operation

The **leave** operation, like the join message, starts by verifying the connection **status** of the cluster and ignoring the method invocation if it's already disconnected. After that initial verification, it sends a **LEAVE** multicast message, closes the TCP connection for the key-value store operations, **saves** the membership **counter** to a file and **removes all the files** that were associated with it through a different thread.

1.3.3 View method

We've also created a **view** method. It's essentially an utilitarian method that shows the node's current view of the cluster.

1.4 Periodic Membership Messages (Fault-Tolerance)

Furthermore, we've also implemented **periodic membership** multicast messages to avoid **stale information** and tolerate the miss of membership change messages (**fault-tolerance**). The necessary logic for this task is present in the **MulticastMembershipSender** class, and it runs in **parallel** to the main thread. Each node, upon receiving this message, compares the logs received with the local logs and updates its cluster view accordingly.

1.5. Missing Details

If there's currently less than three nodes in the cluster, when a new node attempts to join it won't be able to receive 3 membership information messages via TCP, to deal with this situation each of the attempts to receive messages will timeout after 1 second. If all the three attempts timed-out the joining node will assume there's not enough active nodes and join the cluster.

2. Storage Service

2.1. Description

The Storage Service is responsible for **storing** key-value pairs, providing operations such as **put**, **get** and **delete**.

2.2. Implementation

The **Store** class implements this service, extending **Node** and implementing a **KeyValueStore** interface with the **get**, **put** and **delete** methods. It overrides the **receive** method of **Node**. This function is executed whenever a **Store** joins the cluster, and its purpose is to create a **StoreOperationListener** thread that's responsible for delegating operations to **StoreOperationHandler**s (executed as threads).

Whenever a **Store** receives an operation not related to itself, for instance, a **get** operation with a key present in another store, it'll redirect that request to the key-store owner. Upon searching for the key-store owner using a **ClusterSearcher** (implemented by SearchableCluster that uses binary search), it'll create a connection between it and that node and it'll act as an **intermediary** between the client and the key owner node. If the key owner has crashed the current node will send a **LEAVE multicast message** with the node_id that went offline to remove it from the cluster.

2.2.1 Get operation

A **get** operation will result in the file transfer of a key-value-pair present in the storage of that node. If that pair **exists** the **file transfer is initiated**, otherwise, **no file is transferred**.

If the operation is **redirected** to another node that has crashed this operation will be redirected to its neighbors. If they both are down then the file transfer is **unsuccessful**.

2.2.2 Put operation

A **put** operation will result in the **insertion** of a **key-value pair** in the storage of the node. If that pair **already exists**, then the file is **overwritten**. On the other hand, if there's a **tombstone associated** with the key, the operation is **canceled**.

If the operation is **redirected** to another node that has crashed this operation will be attempted three times with 1 second timeout each, this gives enough time for the cluster to update and assign a **new key owner** that will receive the request instead of the crashed node.

2.2.3 Delete operation

When the **delete** operation is executed, the Store will **search** for the **file** with the given **key** as its name. If found, the file will be **deleted**. Regardless of whether or not the file existed previously, the store will create an empty file named **[key].tombstone**.

Upon an operation **redirect** to a crashed node, this operation will have the same behavior as the put operation.

2.2.4 Replicate operations

If **PUT** and **DELETE** messages are prefixed by the **REPLICATE** keyword, the node will not redirect the operation to the correct key owner, allowing us to replicate information in the key owner's neighbors by bypassing the redirection mechanism.

There's also an additional operation **REPLICATE DELETE_RANGE lowestKey highestKey** that deletes all the files in the interval]lowestKey, highestKey]. It's used to reduce the number of DELETE messages necessary for replication handling.

2.3. Missing Details

Key-value pairs are **transferred** upon a **membership change**; this protocol is explained in the *Replication* section of the report since these two sections are intrinsically linked.

3. Replication

3.1. Description

We've implemented data replication: when a node joins and leaves the cluster and upon the reception of a PUT/DELETE message the key-value pairs are distributed accordingly through the available nodes.

3.2. Implementation

To simplify the replication operations we created a package, up.fe.up.pt.cpd.server.replication, that contains some utility tasks for file replication and deletion.

3.2.1. Join

Upon the reception of a JOIN multicast message each node will verify if the new node is one of its neighbors through the ClusterSearcher. If that's the case we'll have two distinct nodes working together to replicate the data, the leftmost and rightmost neighbors of the new node. The rightmost node will send all its key-store own values to the new node plus the values now belonging to the new node, while removing all the redundant data from the leftmost node. Furthermore it'll instruct the node to its right to delete the range of values that used to belong to the rightmost node and now belong to the new node. The

leftmost node simply replicates its own files to the new node and removes the redundant files from the rightmost node.

3.2.2. Leave

Similarly to the mechanism used in the JOIN operation, after receiving a LEAVE message each node will verify if one of its neighbors has left the cluster. If that's the case the two neighbors of the leaving node will handle the replication of the data to restore the state and guarantee the replication factor of 3.

Considering a circular representation of the cluster, the leaving node has a neighbor on its left and one on its right. Both neighbors must replicate their own files to each other (because, upon the node leave, they became neighbors). Additionally, the rightmost neighbor now owns the files that were owned by the leaving node and, therefore, must replicate them unto its successor.

3.2.3. Put / Delete

When a put / delete operation is received, the node will locate its neighbors using the ClusterSearcher interface and create a task that will send the correct REPLICATE message, SendReplicateFileMessage and SendDeleteMessage, respectively.

3.3. Missing Details

If the cluster currently doesn't have more than 3 nodes, special care must be taken to ensure that the replication occurs as expected. In this situation the file deletion operations are not performed since the replication factor of each file cannot exceed the value of 3. Additionally we ensure that the replication requests aren't sent to the same node twice whenever the cluster only includes 2 nodes (since a node's predecessor and successor will be equal).

4. Message Format

As specified at the project requirements, our messages are **char-based** and **human readable**. Here's a list of all message formats used in our project:

Multicast:
JOIN ip_addr:port membershipCounter
LEAVE ip_addr:port membershipCounter
MEMBERSHIP ip_addr:port \n\n membershipLog

TCP:
GET key
{REPLICATE} PUT key \n\n fileBytes
{REPLICATE} DELETE key
REPLICATE DELETE_RANGE lowKey highKey
MEMBERSHIP \n\n nodeList \n membershipLog

NOTE:

- Parameters contained in { } are optional;
- The nodeList parameter is a comma separated list of nodes, represented by their ip address and port (e.g. `127.0.0.1:9002, 127.0.0.2:9002, 127.0.0.3:9002`);
- The membershipLog parameter expands upon the nodeList format, adding the value of the membershipCounter after each item (e.g. `127.0.0.1:9002 4, 127.0.0.3:9002 6, 127.0.0.4:9002 1`), the main difference between the log and the nodeList is their size, since the log must be sent at 1 second intervals it cannot exceed 32 items for performance reasons, while the nodeList doesn't have such restrictions.

5. Concurrency

Throughout our project all tasks are created using a thread-pool, the corresponding ExecutorService is initialized in the Node's constructor.

The usage of a thread-pool allows us to reduce the number of threads created by reusing them, this improves the performance slightly by reducing the need to constantly create new threads, which is an expensive operation and makes the program safer because the machine cannot be overloaded by creating an exceedling large number of threads in the presence of a high usage situation.

We did not implement the asynchronous I/O architecture.

6. Fault-tolerance

6.1. Update node membership view even if its down for a long time

In order to make sure a node can learn the current membership of the cluster, even after being down for a long time, we ensure that the node **rejoins** the cluster membership after going down. Upon joining the node will receive a complete list of nodes present in the cluster via TCP. To make sure this protocol works the node **must** leave the cluster, even if it has crashed, note that the information contained in the node is replicated on neighboring nodes therefore it can be safely recovered even during a crash.

We use two mechanisms to detect a node crash and emit a LEAVE message via multicast:

- The StoreOperationHandler detects whenever a node is unresponsive when attempting to redirect a Store operation to it;
- The MulticastListener contains a map that maps a node id to a time to live value, initialized with the value 3 x numberOfNodesInCluster; after each multicast MEMBERSHIP message the time to live of the message sender is **reset** to its original value and the time to live of the remaining nodes is decreased by 1. If the counter reaches 0 a LEAVE message is emitted for the corresponding node. In practice this allows us to monitor whether or not a node is active, with some error room in case a multicast message is missed.

Since a node can be forced to leave during a crash, whenever it becomes available again its membership counter can be outdated, to fix this issue we increase the membership counter whenever a node becomes active with an even membershipCounter value so it becomes correct.

6.2. Node doesn't have up-to-date membership view, therefore redirects store operation to wrong node

If a node doesn't have up-to-date membership view it will send a redirect request to the wrong node, but that incorrect node will redirect to the correct node as long as it has an correct view of the cluster, since the redirection requests can be **chained** together. Due to this mechanism even if multiple nodes have an incorrect view of the membership they have enough time to update their view of membership through the multicast MEMBERSHIP messages and redirect to the proper node.