Membership Service

Joining and leaving the cluster

To join a cluster a node will first join its multicast group and then send a **JoinLeaveMessage** with its *id*, *port,* and *membershipCounter* to the multicast group. The nodes already present in the cluster will know whether this message indicates the node is joining or leaving the cluster based on whether the *membershipCounter* modulus of two is zero or one respectively. Each node stores its *membershipCounter* in a file to ensure this information survives node crashes. In case the node is joining the cluster it will wait to receive three **MembershipMessages** from the cluster, there is a timeout of three seconds to receive the three messages upon which a node will resend the **JoinLeaveMessage**. A joining node will send the **JoinLeaveMessage** up to three times.

When a node receives **JoinLeaveMessage** indicating a node is joining the cluster it will create a **MembershipEvent** and update its membership log and send a **MembershipMessage** to the joining node if it hasn't already handled the **JoinLeaveMessage**.

Being the first node to join a cluster

If a node does not receive any **MembershipMessages** in response to its **JoinLeaveMessage** it will assume that it is the first node in the cluster.

Determining membership from membership messages

Upon receiving a **MembershipMessage** a node will update its membership information. Each **MembershipMessage** includes a list of up to the thirty-two most recent **MembershipEvents** and a set of the nodes in the cluster known by the node that sent the message. A **MembershipEvent** instance stores a node's *id* and its *membershipCounter*.

For each **MembershipEvent** in the **MembershipMessage** the receiving node will update its membership log if that **MembershipEvent** is more recent than the one in its membership log. If the *membership log* has been updated with a **MembershipEvent** the *cluster nodes* will also be updated accordingly.

Updating the membership log

The membership log is implemented as a list of **MembershipEvents**. To update the membership log each node uses the method *addMembershipEvent()*, this method receives a **MembershipEvent**.

If the log has a **MembershipEvent** of the same node with a lower *membershipCounter* it will remove that one from its log and add the new one at the end of the list, if the log does not include a **MembershipEvent** by that node it will be added to the end of the list. Otherwise, the **MembershipEvent** will be ignored. This ensures that the last **MembershipEvents** in the membership log are always the most recent from the node's perspective. The method *addMembershipEvent()* returns *true* if the membership log has been updated and *false* otherwise.

Sending a periodic membership message

To ensure a **MembershipMessage** is sent to the group approximately once every second, each node will send a **MembershipMessage** to the group every *ClusterNodesSize* seconds, where *ClusterNodesSize* is the number of nodes in the cluster.

Preventing stale memberships

Each time the method *addMembershipEvent()* is called a class field called *lastMembershipUpdateTime* will be updated with the current time. Before sending a **MembershipMessage,** either in response to a **JoinLeaveMessage** or in the periodic ping, a node will check if its membership event is stale. It will do this by calling the *isStaleNode()* method, this method returns true if there is more than one node in the cluster and the current node's *lastMembershipUpdateTime* was more than 3 * *ClusterNodesSize* seconds before the current time, where *ClusterNodesSize* is the number of nodes in the cluster.

Message format

All message objects are instances of subclasses of an abstract **Message** class that implements the **Java.io.Serializable** interface. This is done so we can implement *fromBytes()* and *toBytes()*, based on ByteArray and Object input and output streams, that read and convert a **Message** object from and to byte arrays that can be sent through sockets. This way we don't have to worry about parsing **Message** objects from strings.

Each subclass of the **Message** class is a different type of message that can be sent (**JoinLeaveMessage**, **MembershipMessage**, **GetMessage**, etc.).

The **Message** class has *id* and *port* fields that store the id and port of the node that sent the message. Each subclass has additional fields to store information relative to its function. To ensure human readability each message class implements the *toString()* method.

Key-value Store

The keys are created by using a hash function, which uses SHA-256, by taking the bytes in the file presented and creating a string of characters, and then converted to the big-endian order.

The values are an object of the class **Value**, which has two fields: one for the original file's name, used later for type 'get' operations to create and save the file similarly to the original, and another field to store the original file's bytes.

Storage

The keys are stored in the same folder as the membership logs and the membership counter, each having its name being the key and its content an object of the class **Value**.

In order to determine the correct node in which to store the key, there's a value calculated from the key ranging from 0.00 to 359.99, akin to an angle in a circumference. The same angle will then be calculated for a node and the closest one with a higher or equal angle than the one of the key will be selected using a binary search algorithm.

Membership changes

Join

Upon a node joining, it needs to be able to load its keys, which happens in two different ways: keys that had already been stored in the file system and keys stored in another node that, given the new node, would now shift ownership.

Checking for already existing keys happens by simply checking the amount of files on the folder in which we store a node's data, and if there are more than the base ones, simply loading its names to the **SortedSet** which contains all the keys belonging to a certain node.

In regards to getting them from another node, by using binary search on the angle calculated by the id of the new node, the information of the successor node is calculated and an empty **JoinKeyTransferMessage** is sent.

That message will then be received by the successor node and it will calculate which of its keys should belong to the previous node instead, placing them in the message received and sending it back towards the new node, which will then add them to itself.

Leave

Upon leaving, a node will calculate its successor node, creating a **LeaveKeyTransferMessage**, on which it will load all of its keys and values into a **HashMap**, and send it, leaving the successor node to simply add those news keys and values to itself.

Operations

Any operation, 'put', 'get', or 'delete', can be called for any key on any node, since it's node, by the key given, will then find the node it belongs to and relay the necessary information using TCP

Put

The 'put' operation is called on the **TestClient** by providing it with a filename. From that filename, the file is opened and the hash key is created from the bytes within, and after it the object of the class **Value** will too.

Afterwards, using TCP, a **PutMessage** is sent to a node, containing the information of the key-value itself: the key and the Value.

Once the information reaches the correct node, there's an attempt to create a new file, which will fail and send an error message 'File already existed' back to the **TestClient** if there is already a file with the same key, as long as that same file isn't a tombstone.

Barring that complication, the file will be created and the value stored in it, sending a confirmation message back: 'Put was successful'.

Get

The 'get' operation is called by simply providing it with a key, which, before anything else, will be tested to check if it's in a correct format, giving an error message 'Hash is not in correct format' if that's not the case.

If everything's correct, using TCP, a **GetMessage** is sent to a node and, containing the key, and, upon reaching the correct node, it tries to retrieve the value store, always sending back a response in the form of an object that's an instance of the class **Value**.

Should a file with the key provided simply not exist, the node will send back a **NullValue**, eliciting the **TestClient** to print the error message: 'File doesn't exist'.

In the case of a file that has been ordered deleted, and thus replaced by its tombstone, the node will send back a **TombstoneValue**, which will then prompt the error message: 'File has been deleted'.

If everything goes well, then a normal **Value** will be received by the **TestClient**, and a file with the original name and extension, pre-appended with the string 'received_', will be created with the contents of the original and a successful message will be printed: 'Get file successful', followed by the new file name.

Delete

The 'delete' operation functions similarly to the 'get' operation in terms of sending a message to the node using TCP, however it receives a string message as a response, which will then be printed by the **TestClient**.

If a file with the key provided doesn't exist, then a message 'File doesn't exist', will be sent back as a response.

Had the file already been deleted, having been replaced by a tombstone, the **TestClient** will then receive the message: 'File already deleted'.

If everything goes well, the file itself won't be removed from the system, but it's value will be replaced by a **TombstoneValue**, and there will be a successful message sent back: 'Delete was successful'.

Replication

Replication was implemented as a way to mitigate the damages of a possible node failure as the fact that the information is shared between more than one node leaves a higher possibility of success whenever a node crashes.

It’s implemented with a factor of 3, which means each key-value is stored on 3 nodes, the one which it rightfully belongs plus its two successors.

In order to handle possible message failures, when a node is deleted, instead of the file itself being deleted, its contents are replaced by a **TombstoneValue**.

To ensure that there are always 3 nodes with the key-value information, each node that contains a key, in which the value hasn’t been replaced by a **TombstoneValue**, will periodically send a **CheckReplicationMessage** to the other 2 nodes in which the key should be stored.

Whenever this message reaches a node, one of three things can happen:

- Either the node will have the key and everything is okay.
- The node has the key in a different state. For example, the value in the receiving node has been replaced by a **TombstoneValue**, which means the node which is checking for replication hasn't received a **DeleteMessage**, which means the receiving node will send a message to correct it.
- Or lastly, the node doesn’t have the key and will then get the information to correct that.

Implication on Membership

Whenever a node joins, it will get keys from its successor node and, due to replication, it might be getting them not from an original key owner, but from one of its successors.

Due to this possibility, not only is the key transfer done normally, but the key is actually removed from the file system, instead of the regular delete.

When a node leaves, its keys are also removed from the file system, and sent to the successor node. However, that successor could have already repeated keys, due to, for example, being the second successor for a certain key, while the node who was deleted was the first.

In that scenario, the replication factor will be of 2 before the following replication check, which will notice the fault and correct it.

Since the replication check works by calculating the nodes which the key should be stored in, as long as a node has the correct information regarding the current cluster state, a node joining or leaving won’t create any issues.

Implication on Storage Services

Just like it was previously mentioned, the ‘delete’ operation instead of deleting the actual files, will simply replace their contents by a **TombstoneValue**, and will send a **DeleteSuccessorMessage** to its two successor nodes that share the key.

In regards to ‘put’, it will do the same thing as before, while sending a **PutSuccessorMessage** to the two successors, so they can get the key.

The ‘get’ operation will be the most different, as instead of a check for the correct node which the key belongs to and seeing if it’s the same or send a message if it’s not, it will now check if the current node is one of the three that contains the key and, only if not, it will get the information from another.

Concurrency

Thread pools

Each node has up to twenty-two threads running concurrently, one thread is responsible for sending periodic **MembershipMessages** to the cluster, one to check the replication of values, one to listen to UDP messages and one to listen to TCP messages. Of the remaining eighteen threads nine are for handling messages received by UDP and nine for handling messages received by TCP.

Asynchronous I/O

Each time a node receives a message, either by UDP or TCP, it will submit a handler to be executed by a thread pool (this handler will also be responsible for responding to the message if needed) and then be ready to receive another message. This ensures that the node is always capable of receiving and sending messages without blocking.