Group: Laura Pedenaud, Jean Le Berre, Eloi Besnard, Joshua Randria

A key-value store (almost) from scratch

Link of the practical: https://github.com/otrack/cloud-computing-infrastructures/tree/master/kvstore

Objectif: de-construct a data storage service.

Implementing the key-value store

We proceed in two steps. First, we manage local calls. Then, we amend our code to handle the case where a call at the `Store` interface is accessing data stored in a remote node.

1 Local operations on the data

[Task] Complete the `ConsistentHash` class to encapsulate a strategy of data distribution based on consistent hashing.

Looping on the view.getMembers(), we complete the ring (TreeSet<Integer) and our map addresses (Map<Integer, Address>).

[Task] The `StoreImpl` class should now extend `ReceiverAdapter`. Upon receiving a new view, the KVS assigns a new strategy to the `strategy` field.

In the method init() we implement a channel using JGroup and assign a new ConsistentHash to the strategy with the argument channel.getView(). In the next steps the strategy might already have been initialized (in viewAccepted()) so to avoid redundancy we check if it is the case.

[Task] Complete the `put` and `get` methods to handle the case where the local node is in charge of storing the key. Do not forget that `put(k,v)` returns the value stored under key `k` prior to the invocation. The test `StoreTest.baseOperations` should now run with success.

In the case where the local node is in charge of storing the key, we just need to update the private Map<K,V> data of the current node. Using get(K key) we can retrieve the previous value associated with a specific key before assigning a new value with put(K key,V value).

2 Handling remote data

Our next step is to implement the management of remote calls. To this end, we use the tandem ExecutorService and Callable. A callable takes care of handling a remote call and answering back to the caller. The `ExecutorService` is a pool of threads in charge of executing the `Callable` objects.

[Task] Add a field named `workers` to the `StoreImpl` class. Initialize this field inside the `init` method and use `Executors.newCachedThreadPool()`.

See code

[Task] Add a method `send(Address dst, Command command)` to `StoreImpl`. This method pushes a command to `dst` by bundling it inside a JGroups message.

We create and send a Message containing the required information.

[Task] Create a private `CmdHandler` that implements `Callable<Void>` in `StoreImpl`. To create such a handler, we pass the address of the caller as well as the command to execute. Implement the method `receive(Message msg)`. Upon receiving a message this method retrieves the command from the payload of the message, then submits a new `CmdHandler` for this command to the `workers`.

See code for CmdHandler. In the receive method we get the address of the caller with the method msg.getSrc() and the command with msg.getObject().

[Task] Complete the `call` method in `CmdHandler` to do the computation required by the command. This method creates a `Reply` from the command and sends it back to the caller using the method `send(Address dst, Command command)`. When the command is a `Put`, the local data store is modified appropriately before replying to the caller. In the reply, the field `v` holds the current value of `k` for a `Get`, and the previous value of `k` otherwise.

We identify the instance of the command and do a similar task as it is the local node that can and has to handle (give or update) the information.

To complete the KVS implementation, it remains to modify the code of `put` and `get` inherited from the `Store` interface. In the current state, these two methods handle operations on local data only. We now modify them to also access data stored remotely.

To simplify things, we use a single `CompletableFuture` in `StoreImpl`. This future is stored in a field named `pending`.

[Task] Modify the code of `StoreImpl` to complete the management of remote calls.

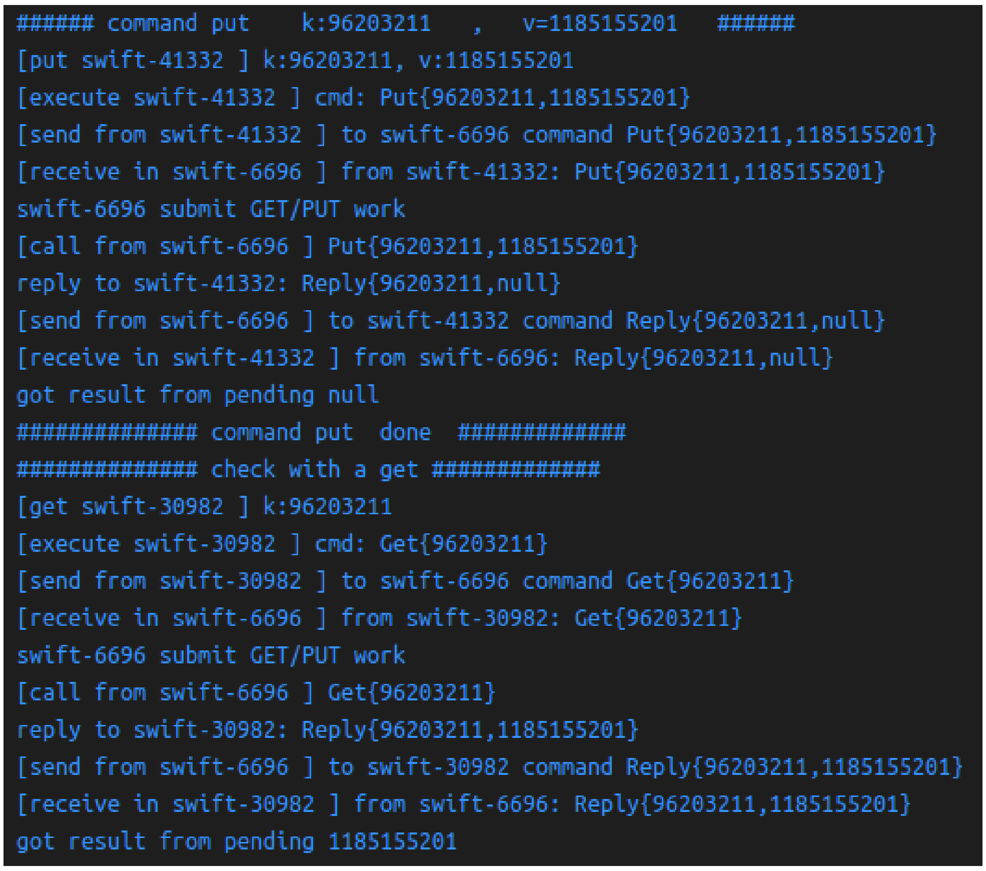
At the call of the get/put method in a node we first create the appropriate command using factory (CommandFactory) and finding the remote node containing the data with strategy.lookup(K k). Then we create pending which is a CompletableFuture before sending the command to the remote node, so we can and have to wait for the result. Meanwhile the remote node that got the request, has submitted the work to his local CmdHandler, which performs the action on the local datastore, builds a Reply and sends it back to the initial caller. Receiving a message that is an instance of Reply, the initial node calls pending.complete() that enables it to come back to the pending.get() in the originally get/put method and we can return the completed result.

[Task] Merge the management of remote and local calls into a single call method `V execute(Command cmd)`. This method should be synchronized to avoid concurrent accesses on the `pending` field.

In the get/put method we only create the command calling factory.new…Cmd(...) and return what is given by the call to the method execute() with the appropriate command as argument. Indeed the execute() method handles the rest of the previously described operations and returns the result according to the instance of the command.

[Task] Correct your code to handle the concurrent access to data across `CmdHandler`. Validate your implementation of `StoreImpl` by running the method `multipleStores` in `StoreTest`.

Here is an example of what we get in the console for a put call, followed by a get call to check the well-performed action:



```
###### command put    k:96203211   ,   v=1185155201   ######
[put swift-41332 ] k:96203211, v:1185155201
[execute swift-41332 ] cmd: Put{96203211,1185155201}
[send from swift-41332 ] to swift-6696 command Put{96203211,1185155201}
[receive in swift-6696 ] from swift-41332: Put{96203211,1185155201}
swift-6696 submit GET/PUT work
[call from swift-6696 ] Put{96203211,1185155201}
reply to swift-41332: Reply{96203211,null}
[send from swift-6696 ] to swift-41332 command Reply{96203211,null}
[receive in swift-41332 ] from swift-6696: Reply{96203211,null}
got result from pending null
############## command put  done  #############
############## check with a get #############
[get swift-30982 ] k:96203211
[execute swift-30982 ] cmd: Get{96203211}
[send from swift-30982 ] to swift-6696 command Get{96203211}
[receive in swift-6696 ] from swift-30982: Get{96203211}
swift-6696 submit GET/PUT work
[call from swift-6696 ] Get{96203211}
reply to swift-30982: Reply{96203211,1185155201}
[send from swift-6696 ] to swift-30982 command Reply{96203211,1185155201}
[receive in swift-30982 ] from swift-6696: Reply{96203211,1185155201}
got result from pending 1185155201
```

3. Data migration

In this last part of the practical, we are interested in adding a data migration mechanism to the KVS.

[Task] Propose and implement a mechanism to migrate data upon a view change (in the `viewAccepted` method). For simplicity, only the case where a node is added will be considered.

My idea for this implementation consisted of 3 steps:

- First, identify which key-value pairs had to migrate in the bucket handled by the new node, comparing the result of lookup in the old view and the new view integrating the new node.
- Second, call the put method from the node previously storing those pairs, to put them in the Map of the new remote node.
- Finally, delete locally the pairs from the node responsible for the bucket containing originally those pairs.

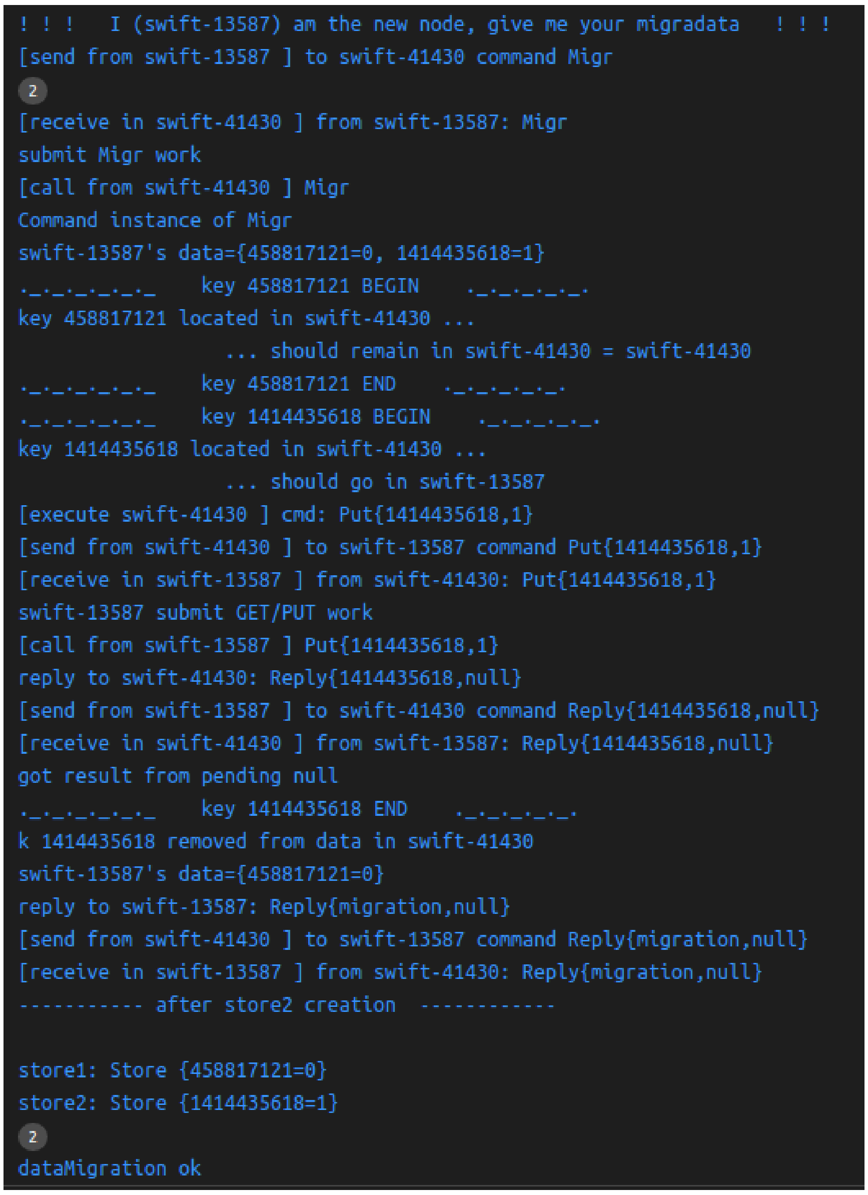
I first tried to implement those step in the viewAccepted() method, which is automatically called when there is a modification in the channel but I encountered a synchronization problem: Indeed the put request sent from an old node to the new node happened while the new node was connecting to the channel so it was not ready and functional at the moment of the request.

To bypass this problem I had to wait for the new node to be fully operational before performing the migration of the data, so I created a new method “getMigradata” that I call at the end of the init method. In this method I send to every other node member of the new view, the command “Migr” added in the CommandFactory that handles the previously described steps in the CmdHandler. I also had to save the old consistentHash in the variable oldStrategy to remember which node stores which key-value pairs.

This solution works fine for the required case of a new node being added, however it would be more convenient to find a way to wait for the new node to be fully operational before the automatic call of the viewAccepted() method.

[Task] Validate your implementation by creating a test named `dataMigration` in the `StoreTest` class.

Here is an example of what we get in the console when performing the getMigradata() method:



```
! ! !   I (swift-13587) am the new node, give me your migradata   ! ! !
[send from swift-13587 ] to swift-41430 command Migr
2
[receive in swift-41430 ] from swift-13587: Migr
submit Migr work
[call from swift-41430 ] Migr
Command instance of Migr
swift-13587's data={458817121=0, 1414435618=1}
._._._._._._    key 458817121 BEGIN    ._._._._._.
key 458817121 located in swift-41430 ...
                  ... should remain in swift-41430 = swift-41430
._._._._._._    key 458817121 END    ._._._._._.
._._._._._._    key 1414435618 BEGIN    ._._._._._.
key 1414435618 located in swift-41430 ...
                  ... should go in swift-13587
[execute swift-41430 ] cmd: Put{1414435618,1}
[send from swift-41430 ] to swift-13587 command Put{1414435618,1}
[receive in swift-13587 ] from swift-41430: Put{1414435618,1}
swift-13587 submit GET/PUT work
[call from swift-13587 ] Put{1414435618,1}
reply to swift-41430: Reply{1414435618,null}
[send from swift-13587 ] to swift-41430 command Reply{1414435618,null}
[receive in swift-41430 ] from swift-13587: Reply{1414435618,null}
got result from pending null
._._._._._._    key 1414435618 END    ._._._._._.
k 1414435618 removed from data in swift-41430
swift-13587's data={458817121=0}
reply to swift-13587: Reply{migration,null}
[send from swift-41430 ] to swift-13587 command Reply{migration,null}
[receive in swift-13587 ] from swift-41430: Reply{migration,null}
----------- after store2 creation  ------------

store1: Store {458817121=0}
store2: Store {1414435618=1}
2
dataMigration ok
```

[Task] Implement a round-robin strategy for data distribution…

Not implemented.

To conclude we implemented a Key-Value Store using a consistentHash to store the addresses of nodes responsible for buckets containing key-value pairs. The necessary communication for selection and attribution of pairs was handled using a common view between the nodes and a CmdHandler. We also implemented a solution for managing the case of migration of data.

It enables us to understand and build a classic architecture for data storage service.