Institut Polytechnique de Paris

M2 Parallel and distributed systems

Logo

Description automatically generated

Cloud Infrastructures CS5004

**Key-Value Store From Scratch**

Authors:  
Jana Toljaga  
Timothée Zerbib  
Aleksandar Maksimovic  
Dimitrije Panic

Content

[1 Introduction 3](#_Toc127026941)

[2 The JGroup Library 3](#_Toc127026942)

[3 Implementing the key-value store 4](#_Toc127026943)

[4 Data Migration 8](#_Toc127026944)

[5 Conclusion 9](#_Toc127026945)

# Introduction

During this lab session, we impelemented a basic form of NoSQL data storage – a key-value store. For this we used the JGroups Library. The KVS was implemented in Java. Throughout the next sections you will see firstly the basic features of KVS and the way they were created. Afterwards, you will be introduced to the strategy that was used to distribute keys between the nodes. Lasty, the solution will be compared to a different strategy based on the number of keys needed to be migrated between the nodes.

# The JGroup Library

***[Task]****To understand the basics of JGroups, follow the online tutorial available.*

In the tutorial they implemented a SimpleChat that allows for communication between views in a JChannel. The whole code you can see in the **demo.SimpleChat** class. Here I will just include the more notable parts of the code.

Text

Description automatically generated



In the code above, you can see the way to connect to a cluster. If we want each node (here they are called addresses) to connect to a same cluster we specify the same name in **channel.connect(“ChatCluster”).** Also, because we want each node to also be able to receive messages, we extend the class **ReceiverAdapter**, and say that this node is also a receiver by putting **channel.setReceiver(this)**. The event loop is where we receive and send messages, that are read from the standard input. Also, every time a new node joins (or leaves) the cluster, all the nodes will be notified and a method **viewAccepted(newView)** will be invoked. View object contains all the addresses (nodes) that are currently in the cluster.

Now, that we know the basics we can start with implementing the KVS, using the JGroups library.

# Implementing the key-value store

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

A picture containing text, accessory, vector graphics

Description automatically generated

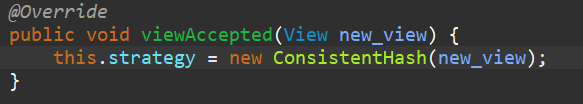
Consistent hashing works in a way that servers are assigned a position on a hash ring. Servers are responsible for keys that come before it, and after the previous server on the hash ring. This allows scalability without affecting the system.

The implementation of ConsistentHash can be seen below. We used the Strategy pattern. Each node has its own strategy object. Every time a new node joins the cluster, we create a new strategy object for that node. We use the HashMap addresses to know which position on the hash ring corresponds to which node. For the position on the hash ring we use its hash code. TreeSet is used to easily lookup which node is responsible for which key. However, we must take into account the large keys for which node with the lowest hash code is responsible for.

Text

Description automatically generated

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



The StoreImpl class now extends the ReceiverAdapter making it capable of receiving messages and view changes.

Also, every time a new view is received the method viewAccepted is called. This method assigns a strategy.

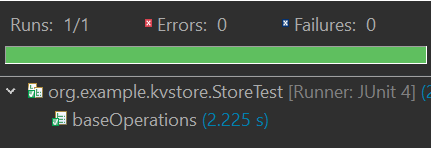
***[Task]*** *Complete the put and get methods to handle the case where the local node is in charge of storing the key. Do not forget that put(k,v) returns the value stored under key k prior the invocation. The test StoreTest.baseOperations should now run with success.*

Text

Description automatically generatedText

Description automatically generated

In the pictures above, you can see the code for get and put when we know there will be no remote calls (implemented in later tasks).



The test executes successfully.

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

*Text

Description automatically generated*

**

ExecutorService will be used to handle remote calls in a separate thread.

***[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.*

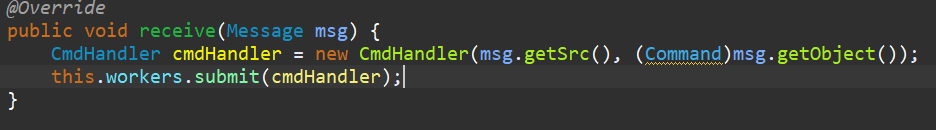
Text

Description automatically generated

This method sends the Command object to a node (address).

***[Task]*** *Create a private CmdHandler that implements Callable<Void> in StoreImpl. To create such an 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.*

**

**

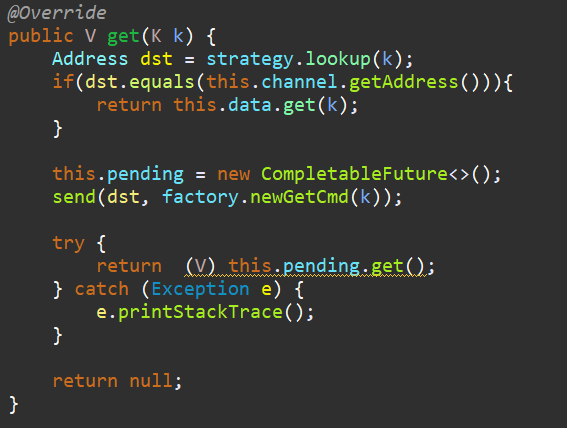
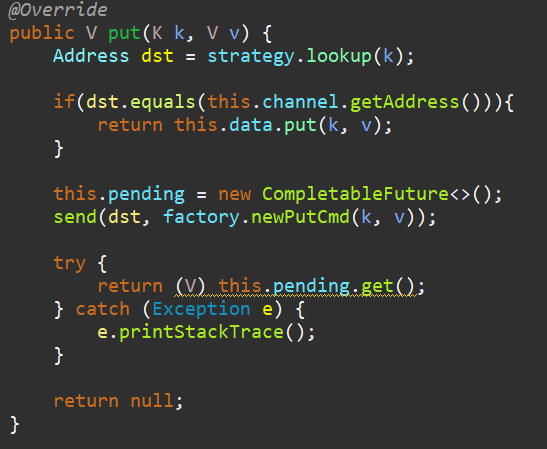
Upon receiving a message, the node will initiate a CmdHandler which will execute the command. This will be implemented in a later task.

***[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.*

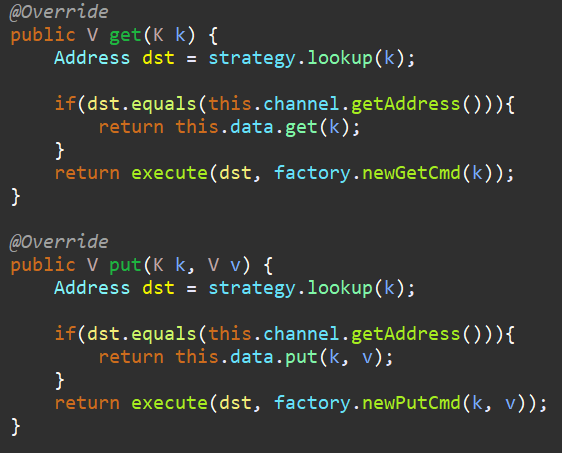
**

This method handles the Get request and the Put request. Afterwards it just sends a Reply to the sender. If the request received is a Reply it will adjust the value of the pending accordingly.

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

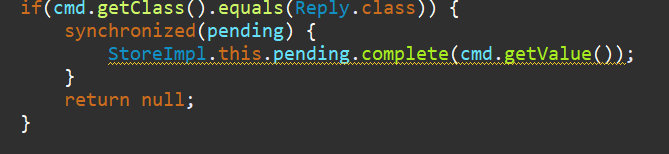
**

In the remote call scenario, the put and get methods firstly create a CompletableFuture object. Afterwards they send the command to the node responsible for that key and afterwards they wait for the result from the handler.

***[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.*

Text

Description automatically generated

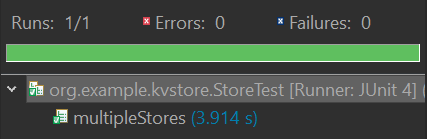


We used the execute method to make reduce the code in put and get method. However, since the this.pending.get() is blocking there is no reason to add synchronization as it will negatively impact the performance. However, we add a monitor in the call class to ensure that no one is accessing that object. Basically, that only one thread that access the pending when we are running the complete method. The adjustments for put and get are also given above.

***[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.*

To avoid concurrent access on data we used ConcurrentHashMap, instead of HashMap for the data. This is because ConcurrentHashMap is thread safe.

**

**

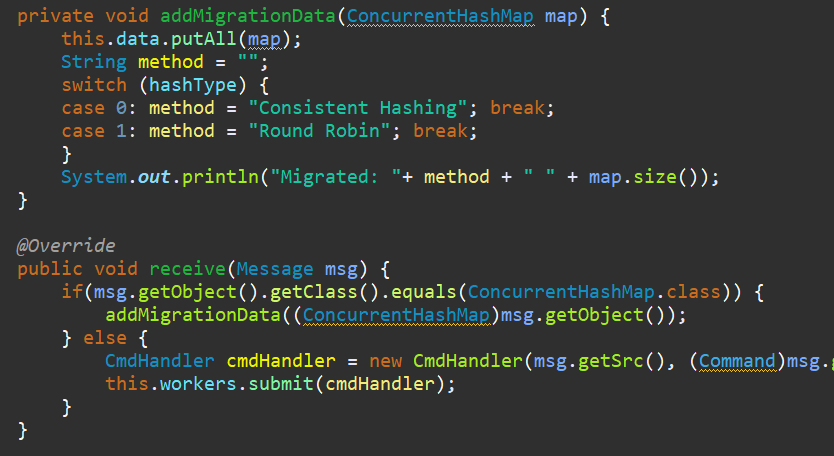
The test runs successfully.

# Data Migration

***[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.*

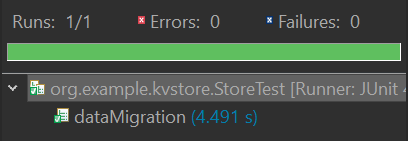


Every time the view accepted method is called we check whether there is need for some data migration. Each node check if it’s still responsible for the keys that are stored on it. If not, he collects the keys and sends it to the node that is now responsible for the data. Also, he removes the keys he is not responsible for anymore.



We add addMigrationData to the receive method. This method puts the keys in the hashmap. This is all thread safe, because no matter the time of checking their maps, in the end we are sure everyone will only have keys that they are responsible for.

*****[Task]*** *Validate your implementation by creating a test named data Migration in the StoreTest class.*



The test creates 3 nodes and puts keys onto them. Afterwards, it adds a new store (store4) and validates if all the keys that were previously created still are present.

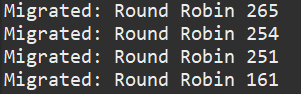
***[Task]*** *Implement a round-robbin strategy for data distribution. Create a test named strategyComparison in StoreTest. This test should assess the advantage of consistent hashing over a round-robbin strategy. (For instance, it may compare the two strategies in regard to the number of integer pairs migrated upon the addition of a node.)*

**

The round robin strategy assums that the distribution of keys is uniform. So we presume that using % addresses.size() will yield to equal distribution between the nodes.

The test insert 100 keys in a strategy with consistent hashing, and then it adds another node and prints the number of keys migrated. Afterwards, it does the same for the round robin. We can see that with consistent hashing the number of migrated keys is just 226. This is because it only has to redistribute keys between two nodes. (node that comes after the new node, and the new node). While for the round robin almost all the keys had to be redistributed. So regarding number of keys migrated consistent hashing outperforms round robin.

**

**

**

# Conclusion

During this lab session we have managed to implement a key-value store. We tested the performance of two different strategies and tested each component of the system to be sure that everything is working.