Assignment 3: KVSdesign _558team5 Design Document

(1) Purpose of the Assignment/System:

The purpose of this assignment is to design a distributed key-value pair storage system. The main idea behind distributed key-value storage is that, for large and important datasets, we want a storage mechanism to fulfill main requirement as correctness (value stored with a particular key should be retrievable using same key), highly available (with very less downtime) and reasonably fast. To achieve first requirement, we can use centralized system but for second and third requirement, we structured the distributed system in a way such that if one node goes down, the remaining can still respond correctly to queries and also the data is distributed across the system with better performance such that value associated with a given key can be retrieved quickly.

(2) Assumptions:

a. The system, with each individual component of the system is called as node. This is implemented in Java using all the required packages and functionalities.
b. The communication within the system, i.e. between individual machines is carried out by means of Asynchronous Messaging. It means the system puts a message in a message queue and does not require an immediate response to continue processing.
c. The client provides the host name and port number of the system to communicate, number of nodes (m value) and Strings (acting as a key in the system). The client can find any individual node in the system and make RMI call on it, it can store and retrieve the data for any key supported by the system.
d. As soon as the client enters m value, the system starts with 2^m nodes. Example: m = 3, system has 8 nodes numbered as 0, 1, 2 …7 and both m and its own number (unique Identifier for individual node) are known to each node.
e. The actual nodes in the system are registered with a single RMI registry on port number 8005. Despite this reservation, the system implements in cases where RMI registry on any host and port.
f. When the actual node is added to the system, there is sufficient time to complete the rebalancing before any another actual node is added to the system.

(3) High-Level Design (UML Diagram):

Some of the nodes in the system are actual and others are virtual. This division is required because if the client provides the value of 'm' tremendously large then we need to create all actual nodes manually which is an efficient way to deal such kind of system. So we are asked that each of the actual nodes is responsible for 'simulating' other virtual nodes such that there is a full set of 2^m nodes. For this simulation, we have used modulo 2^m method stated in the assignment, means each actual node is responsible for simulating the nodes numbered higher than it and lower than the next actual node.
Following is the UML diagram which consists of three main interfaces and respective classes:

a. *<<Interface>> KVS Node*: This is a node which is represented as a stub in RMI registry. The AbstractKVSNode abstract class implements this interface. As the actual node and virtual node implementation have similar methods to implement, we have created an abstract class to define the shared methods. This is useful for reducing the **redundancy**.

The ActualKVSNodeImpl and VirtualKVSNodeImpl classes implements the methods of the above abstract class according to their respective requirements.

ActaulKVSNodeImpl class is an actual KVS Node and acting as a simulator of virtual node.

Important Methods:

Initially, the actual node gets registered into the registry with specific port number.

i. startUp() – This method will start the actual node with ID number, using binding to the RMI registry.
ii. simulateVirtualNode()- This method is used to simulate the Virtual Node.

VirtualKVSNodeImpl Class is used for the implementation of virtual Node. It takes the property object with configuration information and the actual node id.

b. *<<interface>> Hasher*: This deals with the hashing part.

HasherImpl is the class which implements all the methods in the above interface.

Methods:

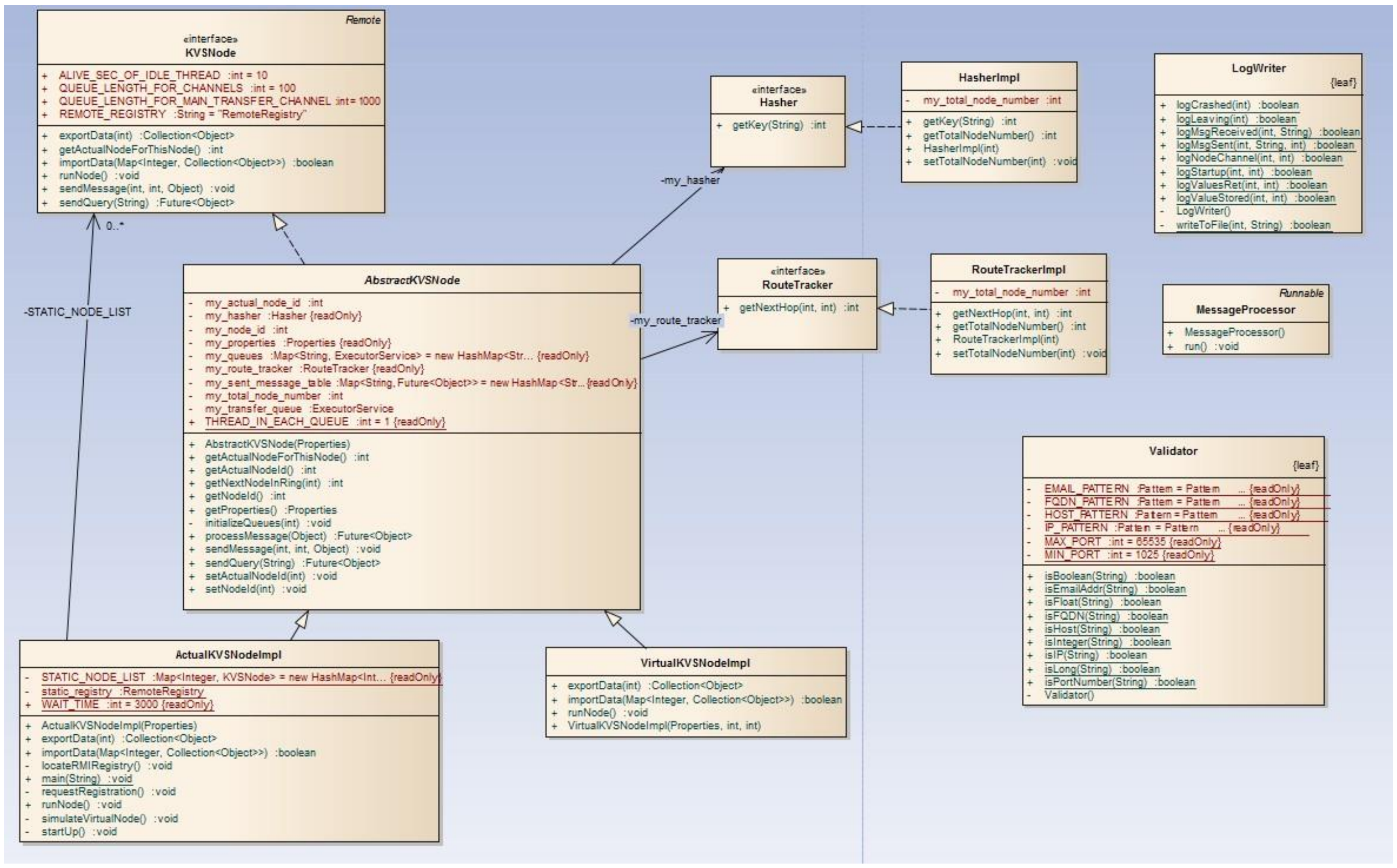
i. getKey(String, int): int- This method takes the message as a key string and total number of nodes and returns the corresponding hashed value of the key string.

c. *<<interface>> RouteTracker*: This interface provides the efficient way to communicate between the nodes.

RouteTrackerImpl class is used to calculate the next hop to pass a message to the proper node which should deal with the message. As the nodes in the system are arranged (in terms of communication channels, which will be explained in detail in the sequence diagram.) in a ring. Node i has channels to nodes i + 2^k for 0 <=k < m called as its neighbors used for finding the next hop. The idea of setting up the communication this way is that sending a message between any two nodes in a k nodes system requires at6 most 0(log k) message sends between nodes. This leads to an **efficient Communication.**

Now the initial ring network is formed using the provided number of nodes. So, every node has multiple incoming channels. The individual channel acts as a blocking queue which connects the two nodes. This channel is the combination of Port id and Node ID the message coming from. This is implemented using ThreadPools and BlockingQueue. But finally we require only one thread coming out of the node which contains the result, for this purpose, we are using TransferQueue. This whole part is given in AbtsractKVSNode class and will be implemented in later assignments.

You can find clearer diagram _558team5-kvsdesign -> Docs -> Class Model


Remote
«interface»
KVSNode
+ ALIVE_SEC_OF_IDLE_THREAD :int = 10
+ QUEUE_LENGTH_FOR_CHANNELS :int = 100
+ QUEUE_LENGTH_FOR_MAIN_TRANSFER_CHANNEL :int= 1000
+ REMOTE_REGISTRY :String = "RemoteRegistry"
+ exportData(int) :Collection<Object>
+ getActualNodeForThisNode() :int
+ importData(Map<Integer, Collection<Object>>) :boolean
+ runNode() :void
+ sendMessage(int, int, Object) :void
+ sendQuery(String) :Future<Object>
0..*
-STATIC_NODE_LIST
AbstractKVSNode
- my_actual_node_id :int
- my_hasher :Hasher {readOnly}
- my_node_id :int
- my_properties :Properties {readOnly}
- my_queues :Map<String, ExecutorService> = new HashMap<Str... {readOnly}
- my_route_tracker :RouteTracker {readOnly}
- my_sent_message_table :Map<String, Future<Object>> = new HashMap<Str... {readOnly}
- my_total_node_number :int
- my_transfer_queue :ExecutorService
+ THREAD_IN_EACH_QUEUE :int = 1 {readOnly}
+ AbstractKVSNode(Properties)
+ getActualNodeForThisNode() :int
+ getActualNodeId() :int
+ getNextNodeInRing(int) :int
+ getNodeId() :int
+ getProperties() :Properties
- initializeQueues(int) :void
+ processMessage(Object) :Future<Object>
+ sendMessage(int, int, Object) :void
+ sendQuery(String) :Future<Object>
+ setActualNodeId(int) :void
+ setNodeId(int) :void
-my_hasher
-my_route_tracker
«interface»
Hasher
+ getKey(String) :int
HasherImpl
- my_total_node_number :int
+ getKey(String) :int
+ getTotalNodeNumber() :int
+ HasherImpl(int)
+ setTotalNodeNumber(int) :void
«interface»
RouteTracker
+ getNextHop(int, int) :int
RouteTrackerImpl
- my_total_node_number :int
+ getNextHop(int, int) :int
+ getTotalNodeNumber() :int
+ RouteTrackerImpl(int)
+ setTotalNodeNumber(int) :void
LogWriter
{leaf}
+ logCrashed(int) :boolean
+ logLeaving(int) :boolean
+ logMsgReceived(int, String) :boolean
+ logMsgSent(int, String, int) :boolean
+ logNodeChannel(int, int) :boolean
+ logStartup(int, int) :boolean
+ logValuesRet(int, int) :boolean
+ logValueStored(int, int) :boolean
- LogWriter()
- writeToFile(int, String) :boolean
Runnable
MessageProcessor
+ MessageProcessor()
+ run() :void
Validator
{leaf}
- EMAIL_PATTERN :Pattern = Pattern ... {readOnly}
- FQDN_PATTERN :Pattern = Pattern ... {readOnly}
- HOST_PATTERN :Pattern = Pattern ... {readOnly}
- IP_PATTERN :Pattern = Pattern ... {readOnly}
- MAX_PORT :int = 65535 {readOnly}
- MIN_PORT :int = 1025 {readOnly}
+ isBoolean(String) :boolean
+ isEmailAddr(String) :boolean
+ isFloat(String) :boolean
+ isFQDN(String) :boolean
+ isHost(String) :boolean
+ isInteger(String) :boolean
+ isIP(String) :boolean
+ isLong(String) :boolean
+ isPortNumber(String) :boolean
- Validator()
ActualKVSNodeImpl
- STATIC_NODE_LIST :Map<Integer, KVSNode> = new HashMap<Int... {readOnly}
- static_registry :RemoteRegistry
+ WAIT_TIME :int = 3000 {readOnly}
+ ActualKVSNodeImpl(Properties)
+ exportData(int) :Collection<Object>
+ importData(Map<Integer, Collection<Object>>) :boolean
- locateRMIRegistry() :void
+ main(String) :void
- requestRegistration() :void
+ runNode() :void
- simulateVirtualNode() :void
- startUp() :void
VirtualKVSNodeImpl
+ exportData(int) :Collection<Object>
+ importData(Map<Integer, Collection<Object>>) :boolean
+ runNode() :void
+ VirtualKVSNodeImpl(Properties, int, int)

UML Diagram 1: Class Model for Distributed Kay-Value Pair System

d. Class LogWriter: This class is to write any **log message** in a file. In this way, we can keep track of the messages sent/receive, nodes added/deleted and finally writing log to a file. Every Actual node logs its activities using following methods:
   i. logStartup (int, int): boolean - startup information including its ID number and m for the system
   ii. logNodeChannel (int, int): boolean – Log the successor actual nodes, and neighbors
   iii. logMsgSent(int, String, int): booelan- Log messages sent
   logMsgReceived(int, String): booelan - Log messages received
   iv. logCrashed(int) : boolean – (to be decided)Log if/ when the node is discovered to have crashed
   v. logLeaving(int): Boolean – (to be decided)Log if/when the node simply leaves the ring.

   After adding a node and destroying a node, we are thinking to write a log of the process, but we haven't come up with a concrete idea. So these methods are to be decided.

e. MessageProcessor : This class implements Runnable, so it de-queue the message from each channel (BlockingQueue), and move it to the TransferQueue, which converges all the BlockingQueues.

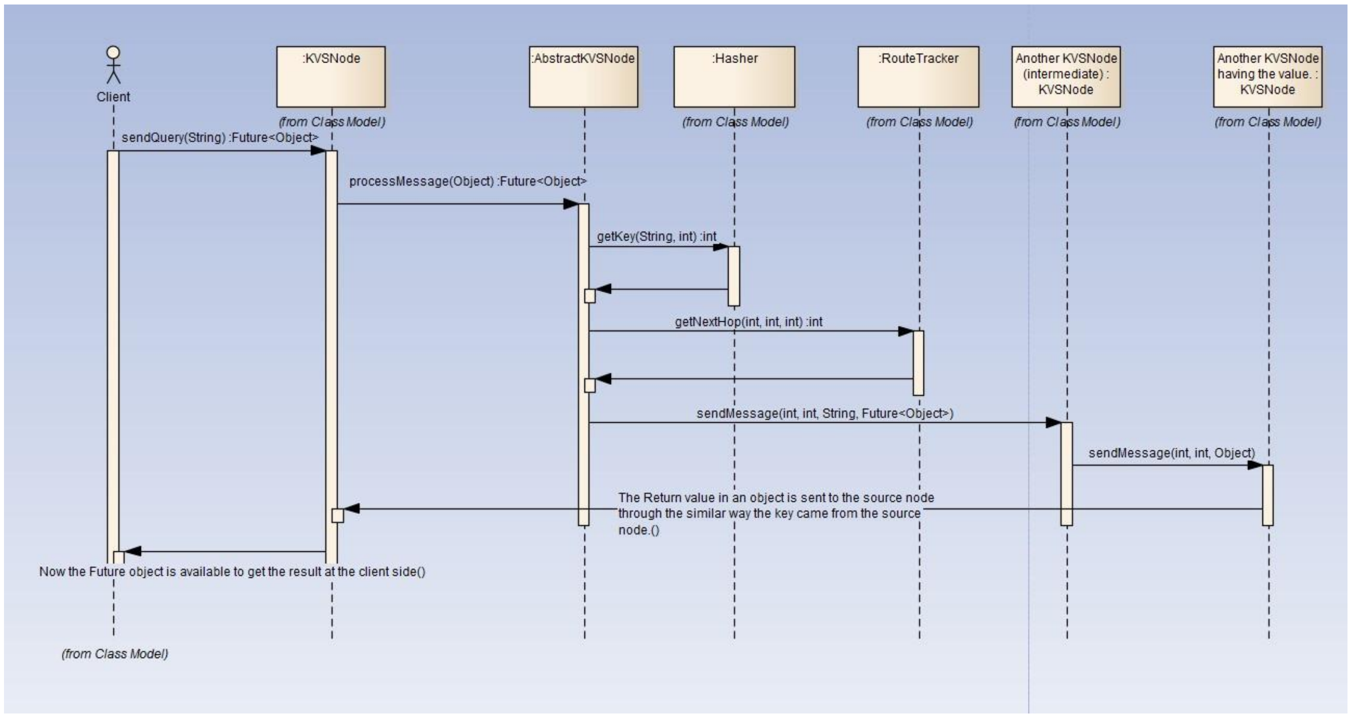
f. Class Validator: This class is used for different types of validations as valid Port, valid Host Name, and valid Integer. For this validations, we have used Pattern which internally implements Serializable.

(4) Sequence/flow Diagram:

The client sends the query to the node with the String. There are two cases for checking the value in the particular node in the ring.

The logical structure and logical processing steps of the system is shown in the figure below:

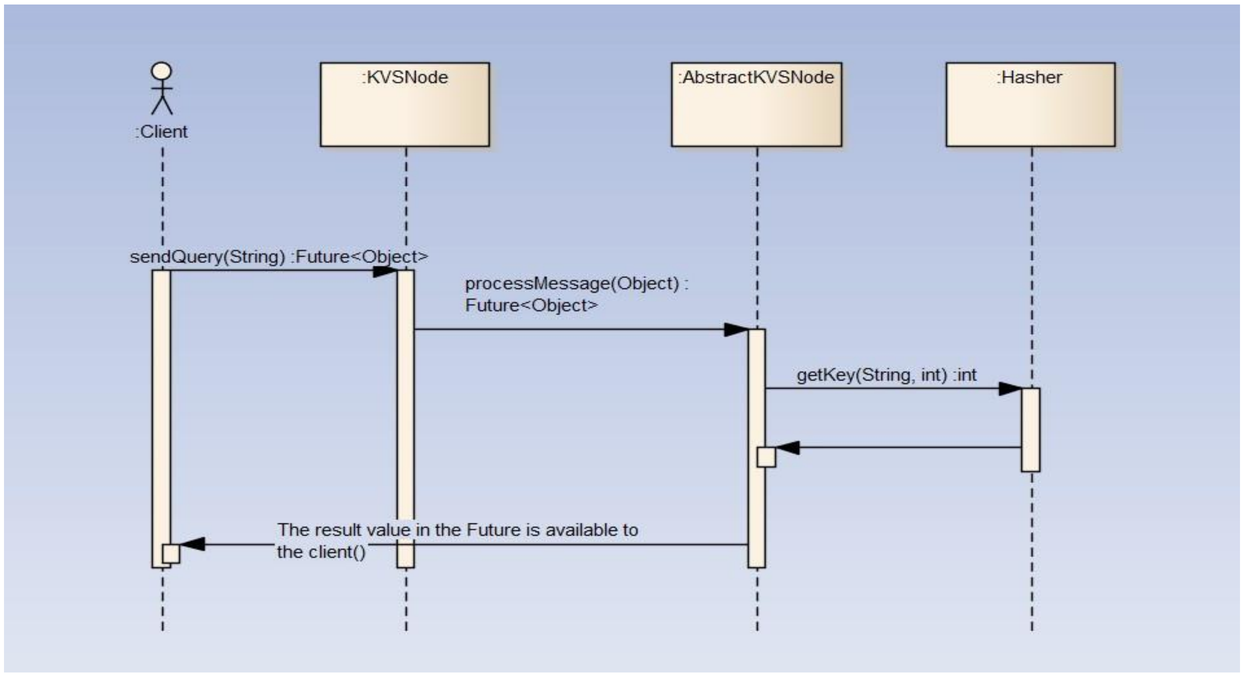
a. *If the value is not in the requested Node:*


Client
:KVSNode
:AbstractKVSNode
:Hasher
:RouteTracker
Another KVSNode (intermediate) : KVSNode
Another KVSNode having the value. : KVSNode
(from Class Model)
sendQuery(String) :Future<Object>
processMessage(Object) :Future<Object>
getKey(String, int) :int
getNextHop(int, int, int) :int
sendMessage(int, int, String, Future<Object>)
sendMessage(int, int, Object)
The Return value in an object is sent to the source node through the similar way the key came from the source node.()
Now the Future object is available to get the result at the client side()

Sequence Diagram 1: If the value is not in the requested Node

Methods: When a client request a query to a node, then it returns a Future object, but the client cannot retrieve the result until the return value comes from the destination node. The sent message is going through the shortest route using RouteTracker, then it eventually gets the destination node and gets the result. When the source node gets the result from the destination node, then it looks through the sent message map which consists of pairs of Strings and Future objects, then make the access to the result available for the clients.

b. *If the Value is found in the requested Node:*


:Client
:KVSNode
:AbstractKVSNode
:Hasher
sendQuery(String) :Future<Object>
processMessage(Object) : Future<Object>
getKey(String, int) :int
The result value in the Future is available to the client()

Sequence Diagram 2: If the value is found in requested node

Methods: It is much simpler than the previous case. It returns Future object to the client request, but it does not need to go through the ring to find the destination node.

(5) Dynamic System Changes Implementation:

Rebalancing: The rebalancing is possible by two means: Adding a Node, Leaving a Node.

a. Adding a node: A new actual node can join the system at any time when the system has less than 2^m actual nodes. exportData(int): Collection(Object) is the method invoked by a new actual node which joins the ring. A virtual node pass the data it has been storing to the new node as return value of the method.

b. Leaving a node**:** (i) The node leaving with orderly fashion: When the actual node tries to leave by its own will, then it should give all its responsible nodes' information to its neighbor actual node. This will be implemented in importData(Map): boolean method. (ii) Node destroyed by 'Crashing': (To Be Decided) when the node is destroyed by unresponsive crashing, we are thinking to use 'Backup' phenomenon, so that few actual nodes will take the backup of their neighbors. So if one goes down, we can restore that information. OR a 'snapshot' phenomenon to get the lost information.

(6) Test Cases:

Serial Number	Test Case Description	Provided Input	Expected Result	Obtained Result
1	Client Provinding wrong input as Command Arguments	Host Name: Port Number: m value: String to search:		
2	RMI registered on wrong port Number	RMI port Number: negative value / Non-number value		
3	Actual Node not started with specific ID			
4	Actual Nodes not connected to neighbors properly			
5	Log File generated/ not			

(7) Conclusion:

An interesting team assignment to design and implement a distributed key-value storage System. This is a completely new learning for all of us. We have learnt many new concepts as TransferQueue, Future, BlockingQueue and use of RMI registry in detail.

Even though we are on the design stage and it is a relevantly small size application of the distributed system, we can see it is pretty complicated system, there are lots of behind scene we have not thought about specific implementation. However, after the long consideration of the design, we come up with the idea, all the expected features which will be implemented in the end of the quarter are feasible.

References:

(1) Assignment 3 description given by Prof. Daniel Zimmerman

(2) Chord Paper: http://pdos.csail.mit.edu/papers/chord:sigcomm01/

(3) Design Document: http://www.ehow.com/how_6734245_write-software-design-document.html

(4) Design Document help online: http://blog.slickedit.com/2007/05/how-to-write-an-effective-design-document/

(5) RMI Implementation guide: Implementing Asynchronous Remote Method Invocation in Java.pdf

(6) RMI Implementation guide: Future-Based RMI: Optimizing Compositions of a Remote Method Calls on the Grid.pdf