Project Report

# **E:\Masters - NSM\Distributed Systems\Project\distributed-systems-img.jpgDistributed Key-Value Store**

## Team Members

**Rakesh Gopinath Nirmala** [rakeshgn@kth.se](mailto:rakeshgn@kth.se)

**Aruna kumari Yedurupaka** [akye@kth.se](mailto:akye@kth.se)

## Introduction

*“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable“*

***– Leslie Lamport.***

Distributed system is a group of autonomous computers that interacts with each other to achieve a common goal. These collection of computers operate in a way that appears to its users as a single coherent system. There are different types of distributed systems – systems that work to solve a computation, physically distributed systems that store and provide access to resources etc. With the advent of recent technological advancements and increased Internet usage via socializing, data sharing and online file storages, the distributed systems have gained furthermore importance and patronage. Most of the successful establishments such as Google, Facebook, Amazon, and Netflix and so on heavy rely on these systems.

## Challenges

Distributed systems and architectures are highly complex models whose design requires a deep understanding of the entire system and a greater futuristic view. There are still lot of unanswered questions that requires in depth research and thinking. The key challenges that determines the durability of a distributed system are as follows. These are also the essential aspects that one should consider while design a distributed system.

* Transparency – Location, Replication, Resource Migration (Single coherent system)
* Openness – Easy to build, maintain and reconfigure
* Reliability – Fault-tolerance, Availability, Security and Data Consistency
* Scalability – Extendibility or Elasticity to grow/shrink
* Performance – Load balancing, adaptation to network characteristics (Bandwidth, Latency…)

## Objective

The objective of this project is to build, implement and test a simple, reliable, partitioned and distributed in-memory Key-Value (KV) store with linearizable (atomicity) operation semantics. The important aspects to be considered are Networking, Bootstrapping, Group Membership, Failure Detection, Routing, Replication and dynamic Reconfiguration.

## Scope

As per the requirement, Kompics with Java is used for the implementation. The scope of this report is to

* Explain the underlying architecture and implementation of Key-Value store.
* Describe the simulation scenarios used for testing the implementation.
* Describe how a new JOIN request is handled in the event of node failure (Partial Reconfiguration).

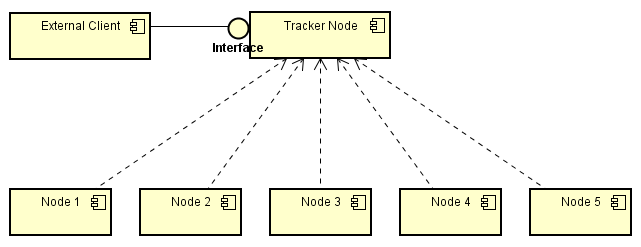
## System Architecture

The KV store in our system shall serve as a directory that stores the different countries and the list of major cities. The key will be the name of the country and the value will be a list of major cities. The user shall store a particular entry by *PUT(<country\_name>, <list\_of\_major\_cities>)* and shall fetch a particular entry using *GET(<country\_name>)*.

Our system is a partially synchronous static system that closely resembles a fail-noisy model with a replication degree of 3. We also have tried implementing an aspect of dynamic reconfiguration. We have used the following abstractions.

* Read Impose and Write Majority Algorithm for GET and PUT.
* Eventual Perfect Failure Detection Abstraction to detect Node Failures.
* Majority-based ACKNOWLEDGEMENT from all correct processes (nodes) for atomicity.
* Track failure nodes and assign failed node functionality (partition) to the new node in the event of JOIN request.

We have built a partitioned and distributed KV store as per the following architecture.



**Figure 1: Distributed KV Store Component Diagram**

**External Client:** The client is a Kompics client that is launched by the user. In its start handler, it triggers the PUT and GET requests and receives the response. In our system, the clients shall talk only with the Tracker node.

**Tracker Node:** This is the node that manages the data partitions and incorporates *failure detection, group membership grant, relaying of PUT and GET requests to the node and its replicas.* It shall also maintain the data replication list, keeps track of failed nodes and last successfully written value. Every time the client wants to perform a PUT or GET operation, it contacts the tracker node. The tracker shall, however, not perform the actual GET and PUT operation as it does not store any keys.

**Node:** The nodes are the actual entities that store the Key-Value pairs within them. They interact with and respond to the requests from the tracker node. Apart from storing the partition for which it is responsible for, each node also stores the replica of two other nodes thereby achieving required *data replication (Aspect of Availability)*.

## Key Functionalities

The following section explains the implementation of major functionalities, simulation scenarios and the output of each operation. The project was deployed on to a set of machines using the JAR file in the demo session.

#### **Group Membership**

To keep it simple and for less volume of message exchange, we have tracker node to manage the group memberships. The Tracker node and the set of nodes are started externally. Tracker shall subscribe to the JOIN\_REQ\_EVENT and shall listen to the incoming JOIN requests. When a node wishes to join the group, it sends a JOIN\_REQ message to the tracker node by triggering a Kompics event. The tracker shall then respond with a JOIN\_APPRV and also assigns it with a unique Node ID. Each joined node shall use this node ID for further communication with the tracker and vice-versa.

Tracker Node

*JOIN\_APPRV, NODE\_ID*

*JOIN\_REQ*

*JOIN\_REQ*

Node j

Node i

**Figure 2: Group Joining Illustration**

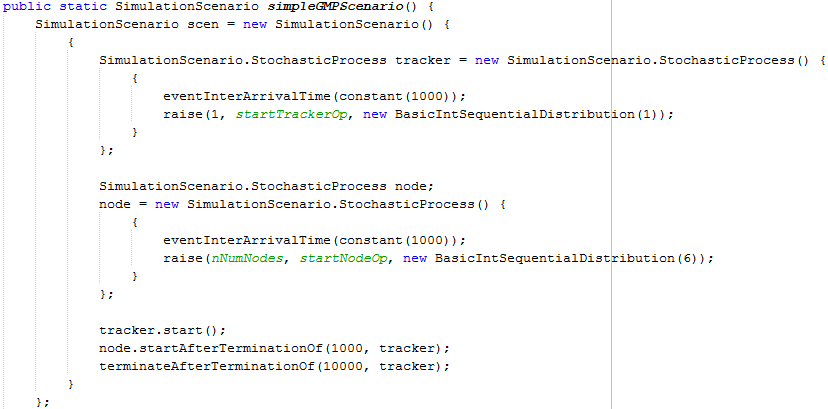
Once the limit of nodes is reached (Maximum number of nodes) new JOIN requests will be rejected. However, in the event of any node failure before the new JOIN request, the new request shall be accepted and the failed nodes responsibility shall be allocated to the new node.

**Simulation**

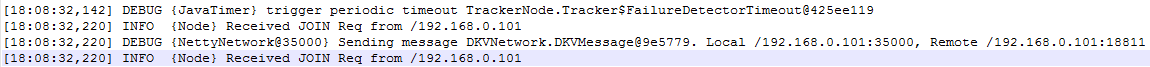
In order to test this implementation, we implemented a scenario using Kompics simulation that does the following.

* Launches the Tracker Node first.
* Launches a set of specified number of nodes which on start triggers the JOIN\_REQ to the tracker.
* Launches nodes more than the specified number to test the error scenario.

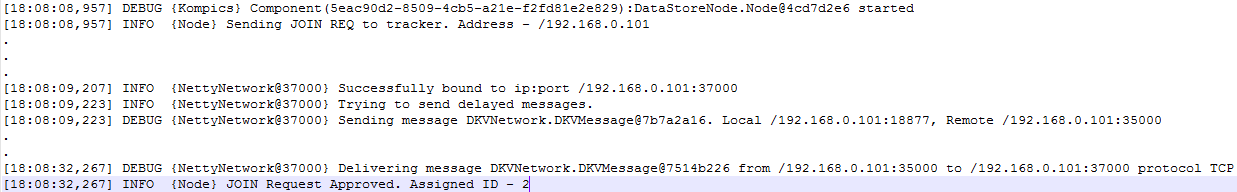
The following screenshots represent the part of scenario and the extract from output obtained during simulation. The Success and failure scenarios are tested.



**Figure 3: Group Membership Simulation Scenario**

****

**Figure 4 (a): Group Membership Simulation Output – Tracker Node**

****

**Figure 4 (b): Group Membership Simulation Output - Node**

#### **Failure Detection**

In our system, we use Eventual Perfect Failure Detector based on a timeout value that is supplied during the initial boot-up. The Failure detection component sits at the tracker node. The tracker node shall schedule a periodic timeout and then broadcasts to all the nodes a Heartbeat Request, *HB\_REQ*, message at regular time intervals (say every x seconds). Each of the subscribed nodes that are correct shall receive the heartbeat request and respond with the Heartbeat Response message – *HB\_REPLY*. In the event of not receiving the response from a particular node, the tracker shall add the Node under the SUSPECT list. In the event of not getting the response for the subsequent request (consecutive) from the suspected node, the node shall be declared as failed and it shall be logged at the tracker.

We did not want each of the nodes to broadcast heartbeat messages to every other node as it is not efficient and shall increase the congestion in the network. Also since our system is a partially synchronous and fail-noisy, we cannot determine the exact bound for the process failure and therefore we use EPFD in our system for node failure detection. In this way, any delay in the delivery of heartbeat messages shall be accounted for.

Tracker Node

*HB\_REQ*

*HB\_REQ*

*Sends HB\_REPLY*

Node i

Node j

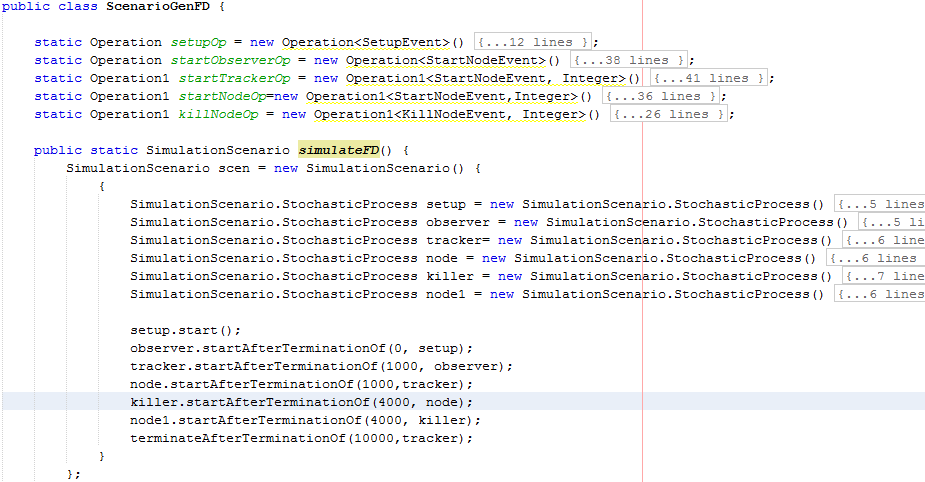
**Figure 5: Failure Detection Illustration**

**Simulation**

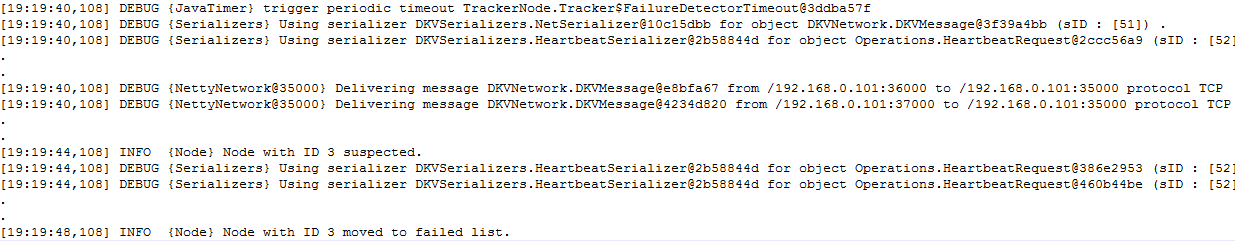
We tested the mechanism using the following scenario.

* Launched the tracker, followed by the set of nodes.
* Killed two nodes and restarted one node again.

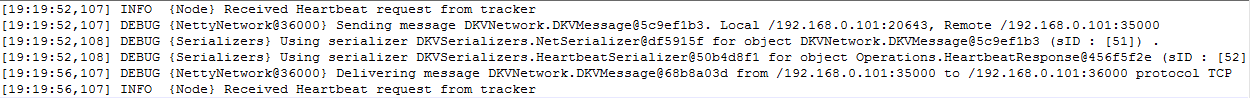
The following screenshots represent the scenario generation followed by the extract from output.



**Figure 6: Failure Detection Simulation Scenario**

****

**Figure 7 (a): EPFD Output Extract – Tracker Node**

****

**Figure 7 (b): EPFD Output Extract – Node**

#### **Data Replication**

To increase the system availability, the key-value data allocated to one node shall be replicated to two other neighboring nodes (next) to achieve better availability and prevent data loss during node failures. It is decided to keep the replication degree to 2 nodes to have minimal cost and replication overhead.

For an instance, in case of 4 nodes, replication shall be achieved in the following way.

Node 1 replicated on Nodes 2 & 3

Node 2 replicated on Nodes 3 & 4

Node 3 replicated on Nodes 4 & 1

Node 4 replicated on Nodes 1 & 2

To achieve linearizability or atomicity of operations (GET and PUT), Acknowledgements are sent back to the tracker node once the operation is successful. In the event of failure, the tracker is informed of the failure.

#### **PUT Operation**

The PUT or Store operation is initiated by the external Kompics client application. Once the PUT request reaches the tracker, it determines the hash of the key string and based on that it determines the node and its corresponding replicas to store the key-value pair. After that, the tracker shall multicast the *PUT\_REQ* to the set of three nodes (1 primary + 2 replicas) and shall *wait for the acknowledgements* from the majority of nodes (at least two in our case). The corresponding subscribed nodes, on reception of the PUT\_REQ, shall perform the actual persist operation and trigger the response indicating the operation status. Each pair of PUT Request-Reply is identified by a unique identifier for the tracker to keep track of the acknowledgements.

If the nodes fail and the majority is not achieved, then there will be an error message logged at the tracker node stating that the corresponding partition is currently down.

#### **GET Operation**

The GET or Fetch operation is again much similar to the PUT operation except that the GET requests and responses are exchanged. The requests are again triggered by the external Kompics client application. It sends the GET request to the tracker node which in turn calculates the hash of the key string and the node that has the corresponding key. It shall then multicast the GET requests to the set of three nodes (1 Primary + 2 Replicas). Then, the tracker node shall wait for the response from at least two nodes and correspondingly display the user the value of the requested key.

All the operations shall proceed seamlessly and there is a single-point of communication between the external world and the Key-Value store system.

**Simulation – PUT and GET Operations**

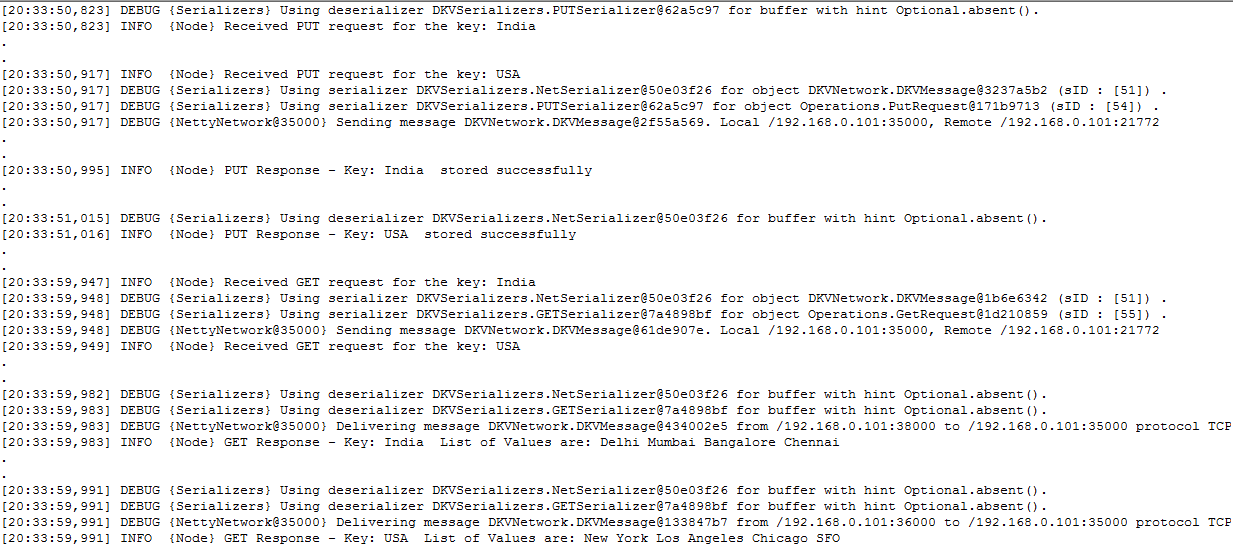
The testing was carried out by launching the tracker, followed by the set of nodes, client node to trigger to the PUT operation and GET requests. After a while two nodes are down to simulate an environment so that it shall throw an error message stating that the partition is currently down. The following screenshots should provide with the simulation scenarios, client node code and the extract of output at the Tracker Node and at the partition nodes (holding data).



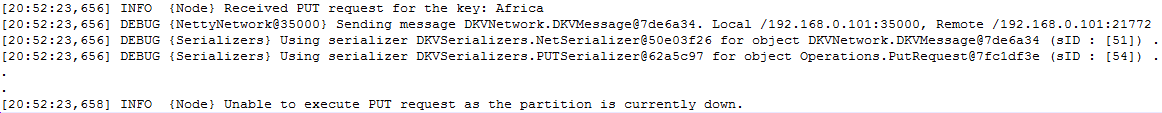
**Figure 8: Client Node – PUT/GET Simulation**



**Figure 9: Simulation Scenario – PUT/GET Simulation**



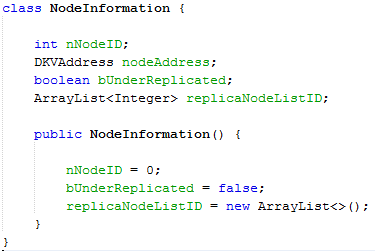
**Figure 10 (a): PUT/GET Simulation Output – Tracker Node (Success Case)**

****

**Figure 10 (b): PUT/GET Simulation Output – Tracker Node (Failure Case)**

#### **Dynamic Reconfiguration**

As part of dynamic reconfiguration, we just did a small part of it. When there is a new node JOIN request and if there are any previous failed nodes, we shall take one from the failed node list and assign that responsibility to the new node. The data corresponding to the failed node is synced with the new node from one of its replicas. This shall make the partition available for use if under-replicated (*NodeInformation* class within TrackerNode.java helps hold the information about each node). However, we have not tested the implementation.



**Figure 11: Class to store Node Info in Tracker Node**

#### **Further Improvements**

The system in its current state is not very efficient and scalable. With sufficient time, we could make the system more efficient and scalable with following changes.

* With increase in the number of nodes, have a separate tracker for each group of nodes. All the trackers can then be connected using a chord implementation for better efficiency. (Just an idea, might need to be thought through at length☺ ).
* Implement Consensus and periodic data sync between the primary node and its replicas with sync requests initiated by the tracker. This shall avoid data inconsistency in case of node failures, network glitches and recovery.

## Conclusion

This project provided a useful insight into the practical implementations and implications of the distributed systems. We had a chance to learn about the various NoSQL databases and their architectures such as Apache Cassandra, MongoDB, Redis, Cats and CaracalDB using Kompics.

## References

[1] <http://kompics.sics.se/> - Kompics Tutorials.

[2] <http://blog.fourthbit.com/2015/04/12/building-a-distributed-fault-tolerant-key-value-store> - A good read for the design of KV store for a beginner.

[3] <https://docs.oracle.com/cd/E17952_01/refman-5.1-en/partitioning-types.html> - Various partition types in Java.

[4] <http://www.aosabook.org/en/nosql.html> - Article about NoSQL databases.