Distributed Key-Value Store with Data Partitioning and Concurrent Replication

Practical Distributed Systems

UNIVERSITÀ DEGLI STUDI
DI TRENTO
Dipartimento di Informatica e Telecomunicazioni

Trento, 17 april 2014

D. Eikelenboom, V. Ekimov

1 Introduction

This document describes the design choices that have been made while implementing a distributed key-value storage that partitions its data over the nodes. The project is part of the Distributed Systems course at the University of Trento, Italy. All requested features in the project description have been implemented, and are consecutively discussed in the following section.

2 Implementation

The project let the options open regarding the technology to use. We decided to implement the system using RPC above sockets as RPC provides a higher-level of abstraction. Choosing Java as an implementation language was a logical choice in combination with RPC and our own programming experience. Java RMI is a variant on the RPC-protocol, designed for Java which involves a way to invoke remote methods in an object-oriented way.

2.1 Partitioning

The general idea is that all nodes are organised in a ring-structure, in which any node might join and leave. The system is organised in a server-client architecture. Servers represent the nodes in the ring, where clients might put and get data or view the ring topology. Moreover, servers might be situated at different machines. Analogous to this setup, in the system, a user is able to launch the 'ServerLauncher' to join new nodes, or let leave nodes in the ring. The 'ClientLauncher' can be used for client operations. Please refer to the section 'User interface' for more details about the user interaction. A server node might reside in a number of states (see figure 1). Either, disconnected, connected or crashed. For example, a node gets connected after joining in the ring.

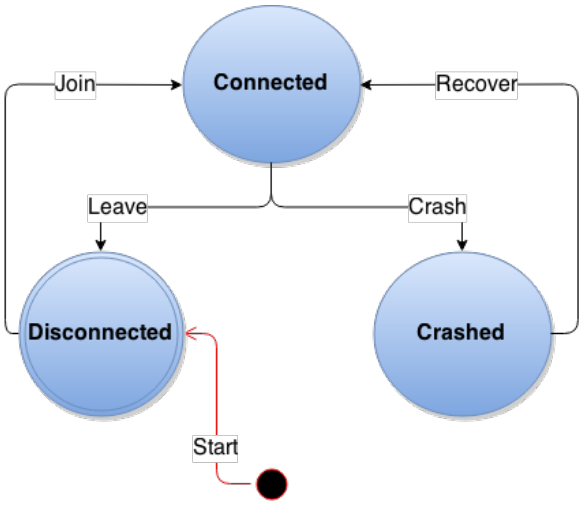

Figure 1: Server Node States

If a node wants to join, it will ask an existing node that he knows in the ring for information about all other ring members. After that it knows its place in the ring and announces its join. Data items that fall in the responsibility range of the new node are transferred from the successor node to the new node. In our implementation the method 'updateItemsAndReplicas' takes care of this procedure. The

first node is an exception in this procedure as it sets up the ring. Moreover every node performs an RMI registration on joining to be able to accept RMI requests.

A leave operation succeeds as follows. The node wanting to leave passes its items to its successor node. Then it announces its leaving to all other nodes and disconnects itself from the ring.

Storage of node contents, i.e. items and and replicas, are stored in "*.csv" storage files fore each node. Upon node creation and modification files are created and updated. Whenever a node is deleted, its contents are deleted from the storage.

When someone wants to retrieve data from the distributed storage system, it can ask any known node in the ring to deliver a specific data item. The coordinator, the asked node, will poll the network for the responsible node for the item, and in case it is not responsible for it itself, it will pass the request to its successor node. Updating operations follow a similar procedure for storing items.

2.2 Replication

A second set of project requirements issued the implementation of replication into the system in order to tolerate faults. Besides storing a data item on the responsible node itself, the items are replicated at the $N-1$ successive nodes. We implemented this replication more in detail as follows.

In addition to the item storage as described under the section 'Partitioning', for replication we added the propagation of replicas at successive nodes. In order to separate a node's own collection of items and the collection of replica's the node is responsible for, each node maintains both collections separated in the form of a Tree Map. The Tree Map is the optimal data structure for storing items as it guarantees efficient $log(n)$ time costs for get and put operations on items.

When someone wishes to retrieve an item with a specific key x, first the system will retrieve all existing items from all replicating nodes for item x. Succeeding, from all nodes x's replica is returned that has the highest version number. Here, the quorum guarantees that at least one node is aware of the most recent item version. This is controlled by assessing the read-quorum parameter R.

Updating items proceeds similar. First all current replicas of the to be updated item x are retrieved, then if the quorum condition holds ($Q = max(R, W)$), the item is updated on all replicas. In specific, the method 'updateReplicas' is assigned this task.

In distributed systems, component failure is a fact to deal with. As well in our case, nodes might crash leading to partial failure of the system. This failure must be recovered by the system without the performance of a 'join'-operation by the recovered node. The system is able to recover by asking another node for the current topology of the network. Then, when recovering the ring it updates its items and replicas from neighbouring nodes. Items are recovered from local storage if the item does not exist in the ring, or if its version number is lower. Even when items get manipulated, or if the network topology changes before a node recovers, the system is able to recover properly.

2.3 Concurrent Replication

To acquire a highly-responsive system, read and write requests are sent in parallel under replicas. At implementation level this is achieved by asynchronously updating of the replicas (excluding the item on the original node) by $N-1$ threads.

2.4 Requirements and Installation

Application requires JDK 7 or higher and Maven 3.2.x. Optionally, it is possible to configure service parameters in 'service.properties' file, which is responsible for RMI port number and replication settings. Before running execute 'mvn clean install'. Maven will produce two jar files and external properties file, which can be executed as following:

- java -jar DHT-version-server-jar-with-dependencies.jar
- java -jar DHT-version-client-jar-with-dependencies.jar

In case of running inside of IDE, use main method of classes 'ServerLauncher' and 'ClientLauncher'.

2.5 User Interface and Set-Up

The user interface allows to set up the distributed network, and do all variety of operations to manipulate its data. Through a command-line interface, one can either start the 'ServerLauncher' or 'ClientLauncher'. The first to set up nodes, the latter for viewing the actual topology of the system, and for update and get operations on data items. An overview of use cases can be found in figure 2.

For transparency, we integrated an extensive debugging-module that prints out the behaviour of the system whenever possible. As well for convenience, a number of example operations are shown on start of the sever. Remind to start a new 'server' for each node. Over the same network, these servers can be spread over multiple machines.

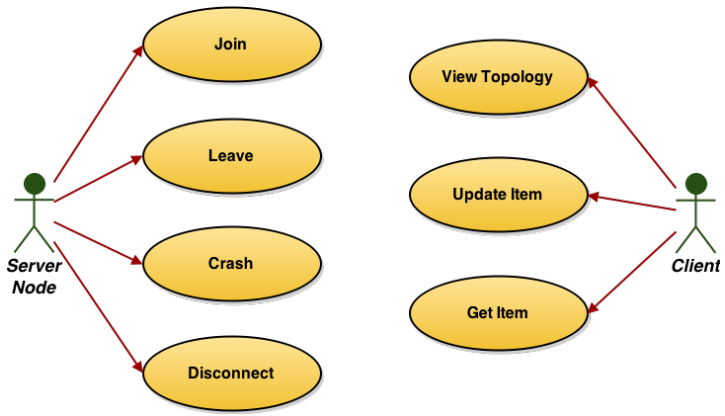

Figure 2: Use Case Diagram

2.6 Testing

In distributed systems, performing unit testing is considered difficult due to the distributed components. To remain the scope of this project on the implementation, we performed mainly tests with scenario's, i.e. we performed tests on the system by following the same pre-defined procedure of operations and checked if the outcome corresponds to our expectations.

One of the most complicated states in which this distributed system might be, is the situation in which a node crashes, and before it can recover several actions on items and the system topology occur. After recovery, the system must remain in a consistent state. For illustration, the test scenario below is

representative for this case.

Test Scenario
Requires: ip-address x, ip-adress y

Node setup

1. Start node 5, 15 at server x
2. Start node 10, 20 at server y
3. Start node 25 at server x from existing y
4. Start node 40 at server y from existing x

View topology / putting / getting item

1. Start client on machine x and [view] topology of one of the nodes
2. Put item with value ‘a value‘ and identifier 12 from machine xs client to some node of a server
3. Now client of machine y, get put item 12 from machine x.

Lets put some more items

1. update,localhost,10,17,random value
2. update,localhost,10,23,random value
3. update,localhost,10,33,random value
4. update,localhost,10,37,random value
5. update,localhost,10,47,random value (to be handled by first node)

Node joining

1. Create node 9
2. Items are passed magically to maintain system invariance
3. Create node 50
4. Items present on first node must be passed back to the new highest node 50.

Versioning

1. Create a new item with identifier 10
2. Update the same item with a random value
3. Use get-operator to check the most recent version number, this yields version >= 2

Node-leaving

1. Let node 15 and 20 leave
2. Items get passed magically

Node-crashing

1. Crash a random node, say 50
2. Now 50 is still in the ring, but not operational and items are unavailable due to the crash.

Node-recovery

1. Recover node 50
2. Items are recovered from replicas and topology is equal to before

Node crashing, updating, recovery

1. Node 25 crashes (including item 12,17,23; replicas 8,10)
2. During the crash update: 17 => new value, 23 => new value
3. Node 13 joins
4. Node 35 leaves
5. Node 25 recovers
6. Now 25 consists all nodes that it should contain, and is aware of the latest version