University of Pisa
Scuola Superiore Sant'Anna

Course

Master Computer Science and Networking

Project Report

Enable Distributed Platform

Authors
Luca Rinaldi `lucarin91@gmail.com`

April 14, 2016

Table of Contents

1 Introduction

The aim of the project is to create a weakly consist distributed file-system by the use of gossiping, consistent hashing and vector clocks.

The communication between node exploit the Java socket mechanism so it can be execute in different ways: on a single machine with threads, on a cluster of servers or in virtual containers using *docker*.

The file-system is implemented as a map with a string key and a number or string value with the following operations:

- **add**(key, value), add only if the key is not present.
- **get**(key), get the value of the key if present.
- **update**(key, value), update the key with the new value only if the key already exists.
- **remove**(key), remove the key if present.

2 Logical Structure

The file-system is composed by two fundamental parts:

- the front-end, that provides the external access to the file-system through a Restful json API,
- the storage system its self where the data are stored and manage.

The first part doesn't have any information of the data store in the system. It only knows the servers, thanks to the gossiping protocol.

2.1 Communication system

All the internal communication are done with the UDP transport protocol, to avoid the overhead of the TCP and assuming a reliable network between the servers.

All the nodes, either the front-end and the storage one, use the gossip protocol to update the list of the servers involved in the file-system. So that each node has two services running on different ports, one for the gossip protocol and one for receiving messages from the other nodes.

When a new request arrives to a front node it is sent to a random storage node from its list and than it wait 5 second for an acknowledgement that the request is correctly served or an error, otherwise it assume that something goes wrong.

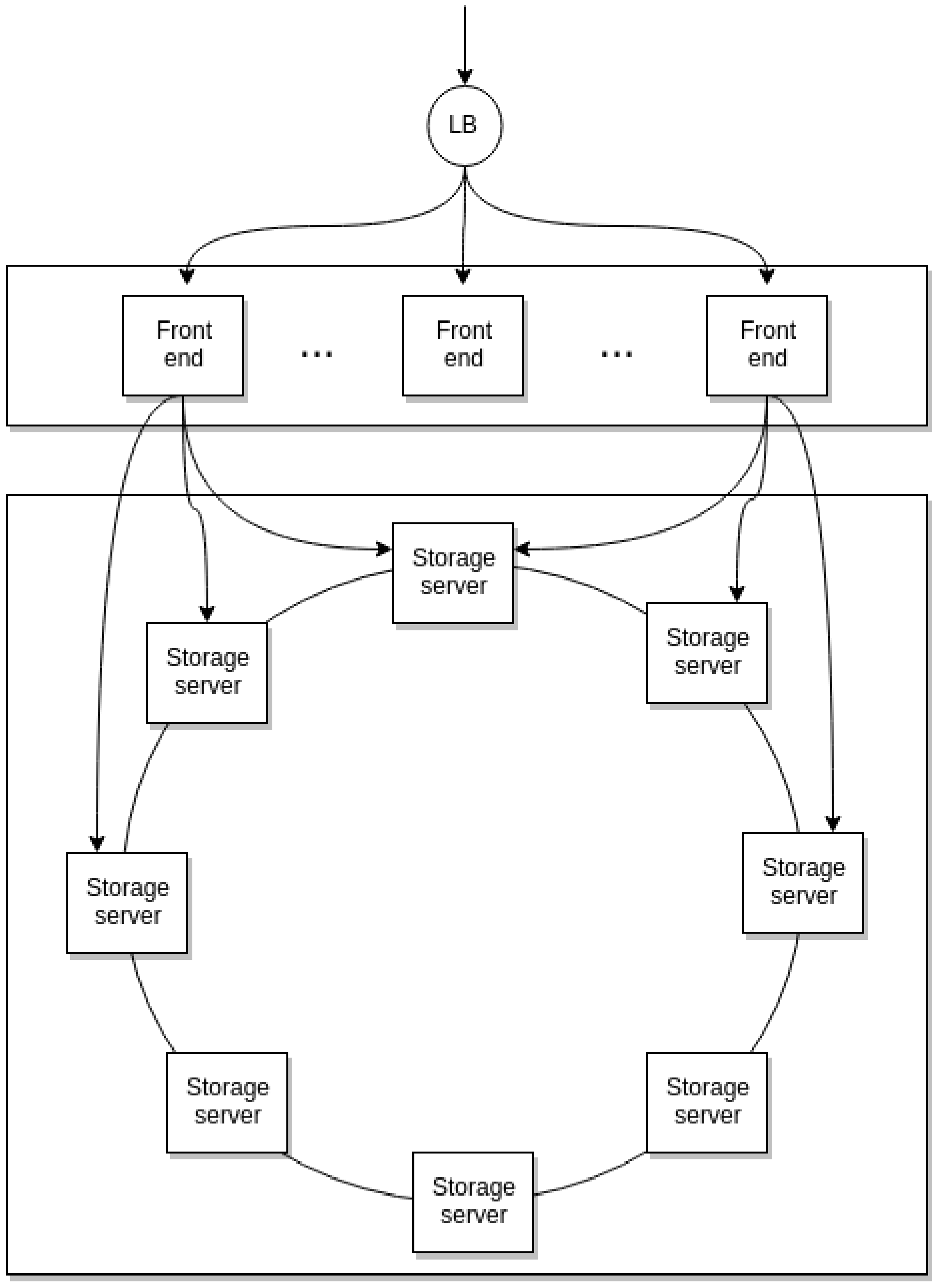


Figure 1: Project logical structure

2.2 Storage protocol

All the storage nodes use consistent hashing to assign a key value to a given server with the following strategy:

- a server is master for all the keys with lower or equal hash value.
- each key is replicated to a fixed number of next server in the consistent hash.

The system use a single master storage protocol without consensus, so the value is written or read without waiting for an acknowledgement from the backup's servers.

Each time a new server turn on it immediately became master for the keys with a lower hash and a backup server for the keys owned by the previous servers. So after their neighbors discovered it, they either send the keys that it has to manage or the keys that it has to keep for backups.

Within the data it is also added a vector clock to keep trace of with server update the value. The vector clock is implemented using a map where the key is the server id and for the value a counter, in this way all the serves that don't have a key are considered zero.

So each time a server update a value as a master it increment the counter with its id inside the object, and foreword this new vector with the key value to its backups server.

This vector clock is used every time two version of a value are founded, after some key management, to decide with is the newer. If two unconfrontable version of the value are founded the node server create value with the two different version and put the `conflict` flag to true.

At this point where a user try to get that key, it receive all the conflict version and it can decide with one it consider the correct newer version by done an update operation.

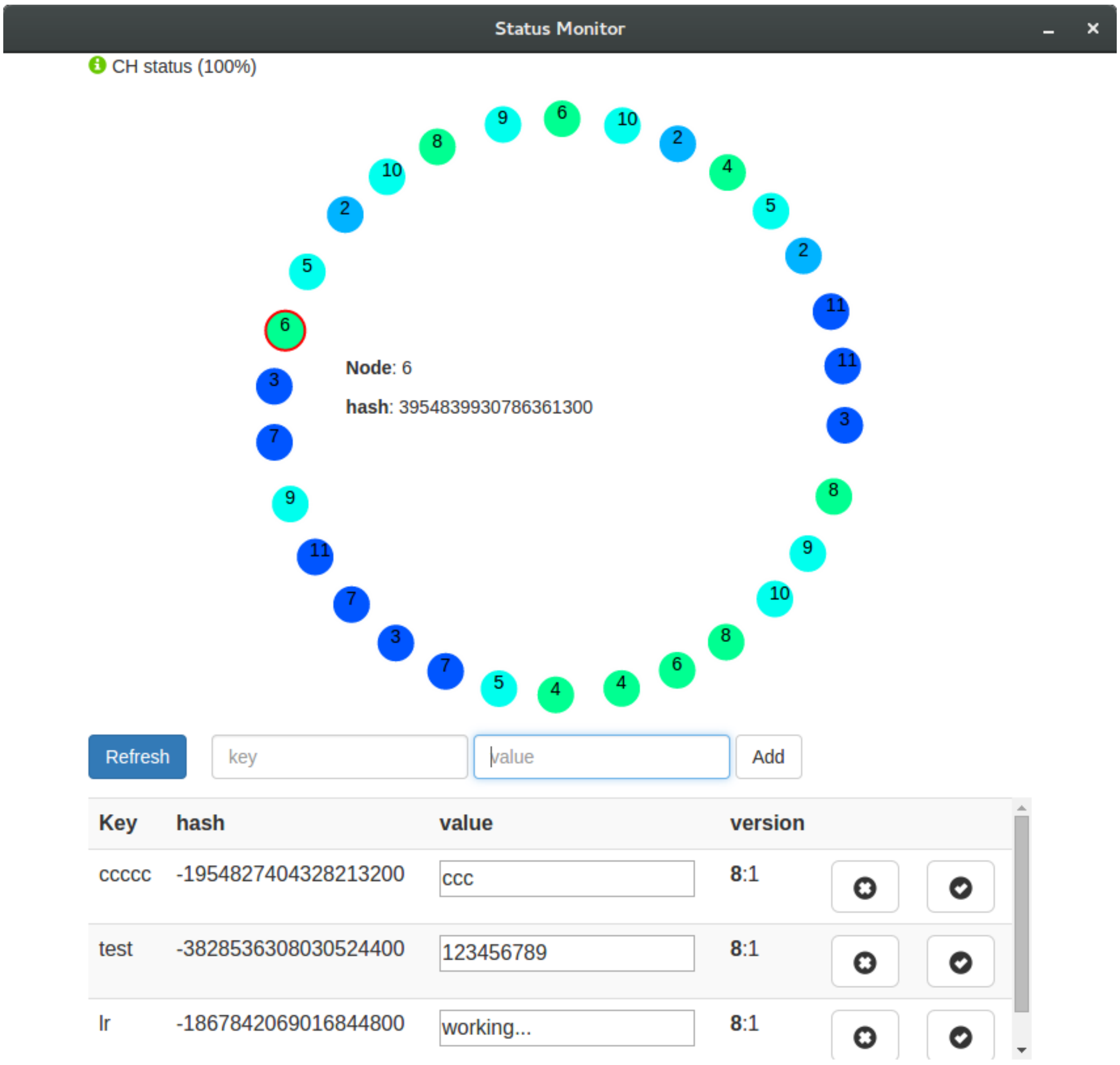
After the update the server resolve the conflict and merge all the vector clock for of all the value, in this way if at same time one of this old values are founded it will be discarded.

3 Project Structure

The system is structured in the following projects:

- **core**, it is a single storage node with all the structure and the essential algorithms.
- **api**, it is a single front-end node
- **app**, it implement a distributed file system with a single API server, where each node run on threads.

- **webapp**, this is a nodejs application that graphically show the state of the file-system.



Node: 6

hash: 3954839930786361300

Key	hash	value	version
ccccc	-1954827404328213200	ccc	8:1
test	-3828536308030524400	123456789	8:1
lr	-1867842069016844800	working...	8:1

Figure 2: screen-shoot of the MonitorWebApp

3.1 Core

The main class in the core project are:

- **Messages**, and its children class - MessageManage, MessageRequest, MessageResponse, MessageStatus - they all represent a json message exchanged between the file-system nodes.

- **ConsistentHash**, the implementation of a Consistent hash owned by each storage nodes.
- **Data**, it represent a generic key value saved inside the system.
- **Node**, a server node of the system either a front-end or a backed server.
- **StorageNode**, it is a storage node object.
- **GossipResurce**, it is the resources use by the front-end to exploit the gossip protocol.
- **PersistentStorage**, a wrapper for the MapDB library used to persistently store data on the service node.
- **VectorClock**, the implementation of the vector clock.

3.1.1 Node class

The node class represents a server of the file-system with the id, the ip and the ports of either the gossip server and the management service. It also has an protected method `send(Node n, Message msg)` method to send a message to a given node with the `DatragramSocket`.

This class is extended by the `StorageNode` and the `GossipResurce` which implements a storage node and a front-end node.

The `StorageNode` can be instantiated with the constructor `NodeService(String id, String ipAddress, int port, List<GossipMember> gossipMembers)` that initialize all the needed structure such as:

- `PersistentStorage` class
- `DatagramSocketServer` with use to receive message from the other node
- `ConsistentHash` class,
- `GossipService` to maintain the consistent hash structure

It has a thread that continually check new messages from the other nodes, and for each one it distinguish two general case: if it is a new request from another node or if it is a `MessageManage` with some manage operation sent by the master to add or modified a backup key.

3.1.2 PersistentStorage

This class implement the storage on file using a B-Tree Map implemented by the library `MapDB`. This data structure have as key the hash of the key of the value and as value a `Data<?>` object, this is because in this way is possible to retrieve in $O(log(n))$ all the keys that have the hash in certain subset.

This operation can be done with the method `getInterval(Long hash1, Long hash2)` and it used to easily understand who is the master of which keys. In this way when a new node is founded on the network thanks to the gossip protocol it is more simple to search the keys that he have to manage as master or the keys that it have to store as a backup node.

3.2 Api

The Restful API, implemented with the Spring web framework, exposes the two end point `/api` and `/status`. The first is the public entry-point to operate with the file-system. The second is a monitoring tool used by the webapp to get a snapshot of all the node in the file-system with their data structures.

The following operation can be used on the `/api` resources:

- **get** a key, `method: GET, parameter: key`
- **add** a key, `method: POST, body: {"key": "..", "value": ".."}`
- **update** a key, `method: PUT, body: {"key": "..", "value": ".."}`
- **delete** a key, `method: DEL, parameter: key`

Each of the previous operations return a json object with the `status` field that can be either `ok` or `error` and the `data` field with optional return data.

3.3 Tests

In the project core there are the following test classes:

- `ConsistentHashTest`, test the methods `add`, `get`, `getPrev`, `getNext` of the `ConsistentHash` class.
- `DataTest`, test the methods to manages multiple conflict versison of the same data in the `Data` class.
- `FSTest`, test the threaded version of the file-system with 10 nodes, it checks all the implemented operation (get, add, update, delete).
- `ModServerTest`, check if the procedure to add or remove a node of the file-system is work.
- `StorageTest`, test the `add`, `get`, `update`, `delete`, methods of the `PersistenStorage` class.
- `VectorClockTest`, test the `increment`, `update` and `compareTo` methods of the `VectorClock` class.

Given the latter test classes also a coverage analysis is done to understend if all the code is correctly tested. The first statistics is the coverage of the test in the whole project.

% Class	% Method	% Line
94.1% (16/ 17)	84.4% (135/ 160)	83.3% (509/ 611)

It is also to consider that same of those not tested method are only a constructor or the `toString` methods of the classes, so we can consider a good pervasion of the testing.

Now there is a table showing in more detail the coverage of the most important classes of the project:

Class	% Method	% Line
GossipResource	100% (2/ 2)	90% (9/ 10)
Node	94.4% (17/ 18)	92.2% (59/ 64)
StorageNode	100% (2/ 2)	92.9% (13/ 14)
ConsistentHash	100% (1/ 1)	68.2% (15/ 22)
Data	95.5% (21/ 22)	98.4% (60/ 61)
PersistentStorage	83.3% (10/ 12)	79.7% (47/ 59)
VectorClock	86.7% (13/ 15)	86.1% (31/ 36)
Message	100% (6/ 6)	100% (9/ 9)
MessageManage	100% (10/ 10)	100% (21/ 21)
MessageRequest	92.3% (12/ 13)	90.9% (30/ 33)
MessageResponse	90% (9/ 10)	81% (17/ 21)
MessageStatus	0% (0/ 8)	0% (0/ 16)

4 How to use

It is possible to use the distributed file-system in the thread version for a single machine, the multi server for a cluster of machine or using the docker container either in a single machine or in cluster.

The simplest way to use the file-system is to download the last release and run it with the only requirement of Java8. In this way the application can be used either in the thread version with the `app-<version>.jar` or in the cluster version with the storage node `core-<version>.jar` and the front node `api-<version>.jar`. In the release is also possible to find the the MonitorWebApp for Linux, MacOS and Windows.

Requirements:

- Java8
- *Nodejs/npm (optional only for the MonitorWebApp)*
- *Docker>=10 (optional only for the docker version of the file-system)*

4.1 Thread version

It can be build with:

```
./gradlew app:build
```

and run with:

```
java -jar app-<version>.jar
```

Optional parameters:

- `-N <number>` number of servers to start
- `-n <number>` number of seeds servers
- `-gport <number>` the port number used by the gossip protocol
- `-mport <number>` the port number of the management server

example:

```
java -jar -N 10 -n 2 -gport 3000 -mport 2000"
```

4.2 Single server

It can be build with:

```
./gradlew core:build api:build
```

To start a storage node run:

```
./java -jar core-<version>.jar
```

To start a front server run:

```
./java -jar api-<version>.jar
```

optional parameters:

- `-id <string>` the port of the server
- `-ip <string>` the ip of the server
- `-p <number>` the port of the server (two successive port are used)
- `-m <id>:<ip>:<port>` a seed server to start the gossip protocol (use this parameters for each seed server)
- `-h <id>:<ip>:<port>` all the server configuration as a single string

example:

```
./java -jar core-<version>.jar \
    -h server1:192.0.0.5:2000 -m server2:192.0.0.2 -m server3:192.0.0.3
```

4.3 Docker version

To build the docker image of the front node and the storage node run

```
./gradlew core:build core:docker api:build api:docker
```

Now is possible to execute a demo by run the following perl script:

```
perl start-docker.pl <number of storage node :default 5>
```

To manually run a file-system node you have to create a new docker network with the command: `bash docker network create --subnet=172.18.0.1/16 fs-net`bash then to start a two node file-system run

```
docker run -d \
    --net fs-net \
    --ip 172.18.0.1 pad-fs/core:0.1 \
    -h server1:172.18.0.1:2000

docker run -d \
    --net fs-net \
    --ip 172.18.0.2 pad-fs/core:0.1 \
    -h server2:172.18.0.2:2000  -m server1:172.18.0.1:2000
```

and a front end node with

```
docker run -d \
    -p 8080:8080 \
    --net fs-net \
    --ip 172.18.0.20 pad-fs/api:0.1 \
    -h rest:172.18.0.20:2000 -m server2:172.18.0.2:2000
```

4.4 MonitorWebApp

The webapp can be used with one of the release version for the different OS or run with the nodejs interpreter with the following command:

```
./gradlew webapp:run
```

References

Java libraries:

- edwardcapriolo/gossip,to implement the gossip protocol between the servers.
- MapDB, to implement the persistent storage.
- FasterXML/jackson, to easilly convert java class to json.
- JUnit, to the test the project.
- Spring, to implement the restful API.
- JCommander, to parse the argument given to the programs.
- Log4j, to manage the log system of the project.

Nodejs and javascript libraries:

- Angularjs, to implement the one page site of the MonitorWebApp.
- Bootstrap, for he graphics of the MonitorWebApp.
- nwjs, to transform the webapp to a native app for Mac Windows an Linux.

Build tools:

- Gradle, to build all the project and manage the dependencies.
- gradle-docker,the docker plugin for gradle.
- jitpack, to bulid java library from github.
- npm, to manage the dependency of MonitorWebApp.
- nw-builder to build MonitorWebApp for the different operation system.

Docker images:

- java a docker image with the openjdk.