SPM Project: Micro Macro Data Flow

Luca Rinaldi

22 February 2016

Contents

1 Introduction

The aim of the project is to create weak consistency distributed file-system by the use of gossiping, consistent hashing and vectors clock.

The communication between node exploit the Java socket mechanism in such a way it can execute in different ways: on a single machine with the use of 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 provide the external access to the file-system through a Restful json API, and the storage system its self where the data are stored and manage. The first part doesn't have any information of the data present on the system, it only know the server present in the FS.

2.1 Communication system

All the internal communication are done with the java `DatagramSocket`, that use the UDP transport packet. All the node either the front end one and the storage one use the gossip protocol to update the list of the server involved in the file-system. So each node have two service running on it with two different port one for the gossip protocol and the other to receive messages from the other nodes. When a new request arrive to a front node is randomly extract a node from its list and send the request to it and wait for an acknowledgement that the request is correctly served.

2.2 Storage protocol

All the storage nodes use consistent hashing to assign a key value datum to a given server with the following way strategy: - a server is master of all the data with lower or equal hash value. - each data is replicated to a fixed number of next server in the consistent hash circle.

The system use a single master storage protocol without consensus, so the data are write or read without wait for an acknowledgement the server that have the backup version of the data.

Each time a new server arrive it immediately became master for the key with a grater hash and a backup server for the keys owned by the previous servers. So after the neighbors discover this new server, they either send the keys that it have to manage and also the key that it have to backup for a previous server.

Within the data it is also added a vector clock to keep trace of the update of the datum. The vector clock is implemented with a map with as key the server iid and as value a long counter, where all serves no present in the map are consider 0. Each time a server update a datum as a master it increment the counter with its id inside the object.

This vector clock of a datum is used every time two version of it are founded, after some key management. If two unconfrontable version of the data are founded the node server keep the one in with the sum of all the counter is greater, this s a simple heuristic base on the fact that it is more probable that a datum that is updated more times is newer.

3 Project Structure

The system is structured in the following projects:

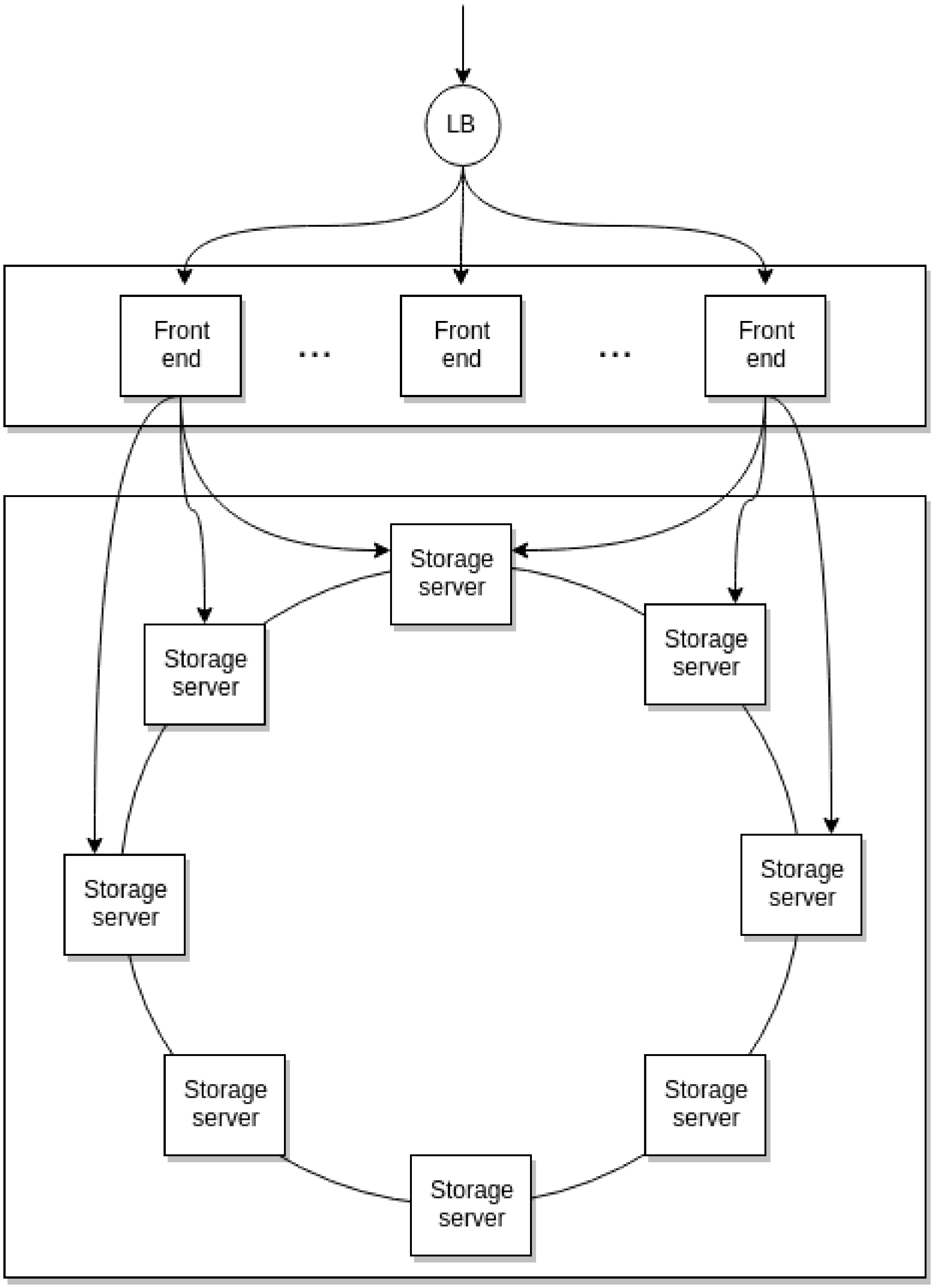
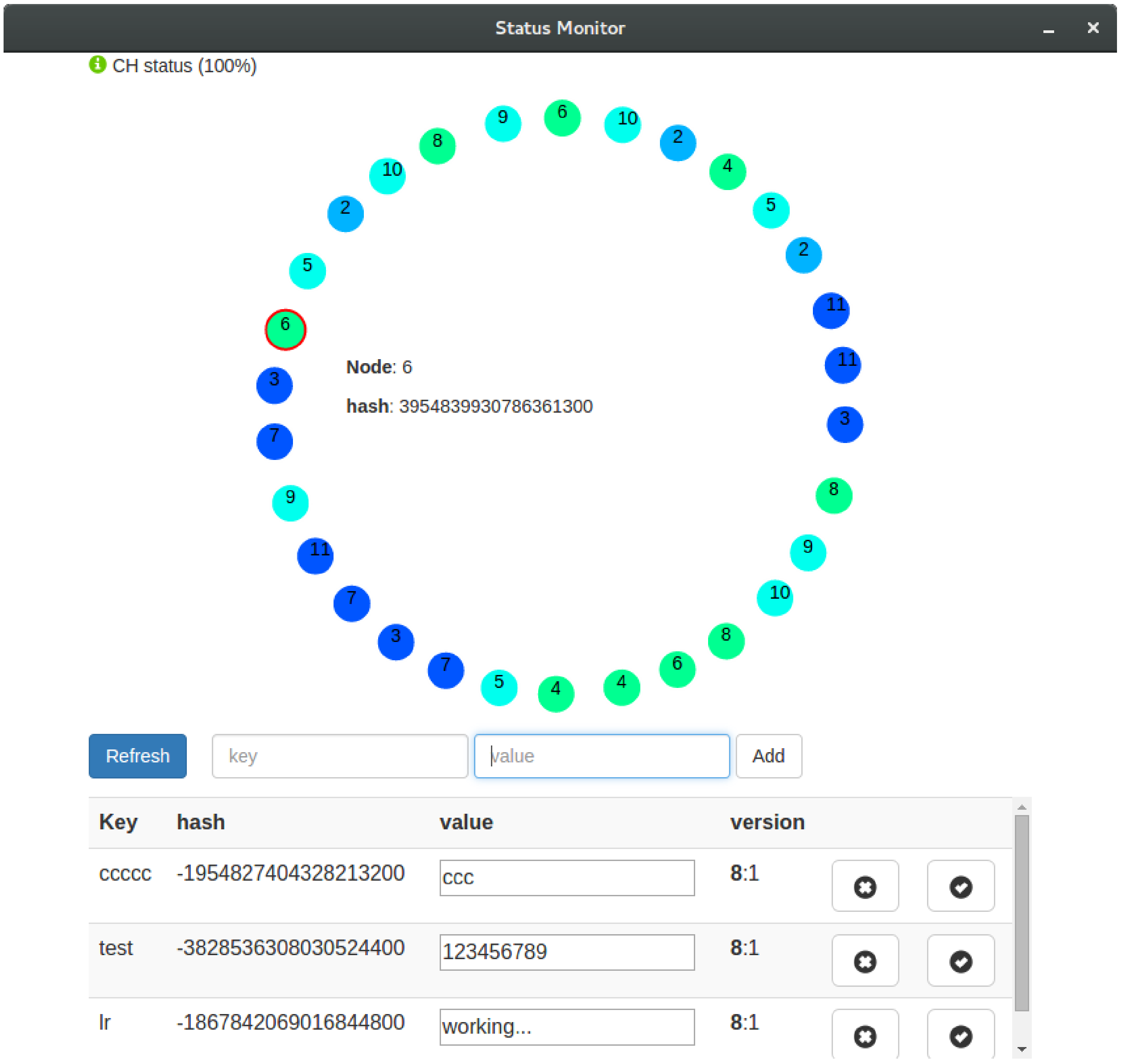


Figure 1: Project logical structure

- **core**, it represent a single storage node with all the structure and the essential algorithms to work.
- **api**, this is a single front-end node
- **app**, it implement a distributed file system with a single API server, where each node run as a thread.
- **webapp**, this is a nodejs application that graphically show the state of the FS.



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 his children class - MessageManage, MessageRequest, MessageResponse, MessageStatus - they all represent a json message exchanged between the FS nodes.

- **ConsistentHash**, the implementation of a Consistent hash owned by each storage nodes.
- **Data**, it represent a generic datum saved inside the system.
- **Node**, a generic server node of the system either a front-end or a backend server.
- **NodeService**, it is a storage node object.
- **PersistentStorage**, a wrapper for the MapDB library used to persistently store data on the service node
- **VectorClock**, the implementation of the vector clock included in each data

3.1.1 Node class

The node class represent a server of the file-system with the id the ip and the port of either the gossip server and the management service. It also expose a `send(Message msg)` method to send to it a new message with the DatragramSocket.

This class is extended by the NodeService and the GossipResurce witch implement a storage node and a front-end node.

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

- `PersistentStorage` class
- `DatagramSocketServer` with use to receive message from the other node
- `ConsistentHash` class,
- `GossipService` to mantein the CH structured

Each NodeService use two port to work, one for the GossipServcer and the other to wait for messages from the other nodes. If it is not specified the service take two consecutive port.

3.2 Api

The rest 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 that get a snapshot of all the node in the file-system including 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`

4 How use it

4.1 Requirements

- **java8**
- nodejs/npm (optional only for the management tools)
- docker>=10 (optional only for the docker version of the file-system)

4.2 Thread version

```
./gradlew app:run
```

if you want also to start also the management tools run

```
./graadlew webapp:run app:run
```

optional parameters (you can pass it to gradle by `-Dexec.args="<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:

```
./gradlew webapp:run app:run -Dexec.args="-N 10 -n 2 -gport 3000 -mport 2000"
```

4.3 Single server

to start a storage server run

```
./gradlew core:run
```

to start a front server run

```
./gradlew api:run
```

optional parameters (you can pass it to gradle by `-Dexec.args="<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:

```
./gradlew core:run -Dexec.args="-h server1:192.0.0.5:2000 -m server2:192.0.0.2 -m server3:192.0.0.3"
```

4.4 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:

```
docker network create --subnet=172.18.0.1/16 fs-net
```

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 server1:172.1
```

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 ser
```

now you can also start the management app with:

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

5 References

5.1 Java libraries:

- edwardcapriolo/gossip (https://github.com/edwardcapriolo/gossip)
- MapDB (http://www.mapdb.org/)
- FasterXML/jackson (https://github.com/FasterXML/jackson)
- JUnit (http://junit.org/)
- Spring (https://spring.io/)
- JCommander(http://jcommander.org/)

5.2 nodejs and javascript libraries:

- Angularjs (https://angularjs.org/)
- Bootstrap (http://getbootstrap.com/)
- jquery (https://jquery.com/)
- nwjs (http://nwjs.io/)

5.3 build tools:

- Gradle (https://gradle.org/)
- gradle-docker (https://github.com/Transmode/gradle-docker)
- npm (https://www.npmjs.com/)
- nw-builder (https://github.com/nwjs/nw-builder)

5.4 docker images:

- java (https://hub.docker.com/_/java/)