University of Pisa
SSSUP Sant'Anna

Course

Distributed Enabling Platforms

PAD-FS

A distributed persistent data storage

Project Report

Luca Rinaldi 9987321

April 15, 2016

Table of Contents

1 Introduction

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

The communication between nodes exploits the Java socket mechanism, thus it can be executed 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 key value map of type $\langle string, string \cup number \rangle$ with the following operations:

- **add**(key, value) that adds the pair only if the key is not present;
- **get**(key) returning the value of the key if present;
- **update**(key, value) that updates the key with the new value only if the key already exists;
- **remove**(key) that removes 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 itself where the data are stored and managed.

The former component does not keep any track of the data stored in the system. It only knows the servers, thanks to the gossiping protocol.

2.1 Communication system

All the internal communication relies upon the UDP transport protocol to avoid the overhead of the TCP and assuming a reliable network between the servers exists.

All the nodes, either the front-end and the storage one, exploit gossiping to update the list of the servers involved in the file system. According to that, each node runs two services on different ports: one in charge of the gossiping protocol, the other for receiving messages from the other nodes.

When a new request arrives to a front-end node it is sent to a random storage node from its list and then it waits 5 seconds for an acknowledgement that the request has been

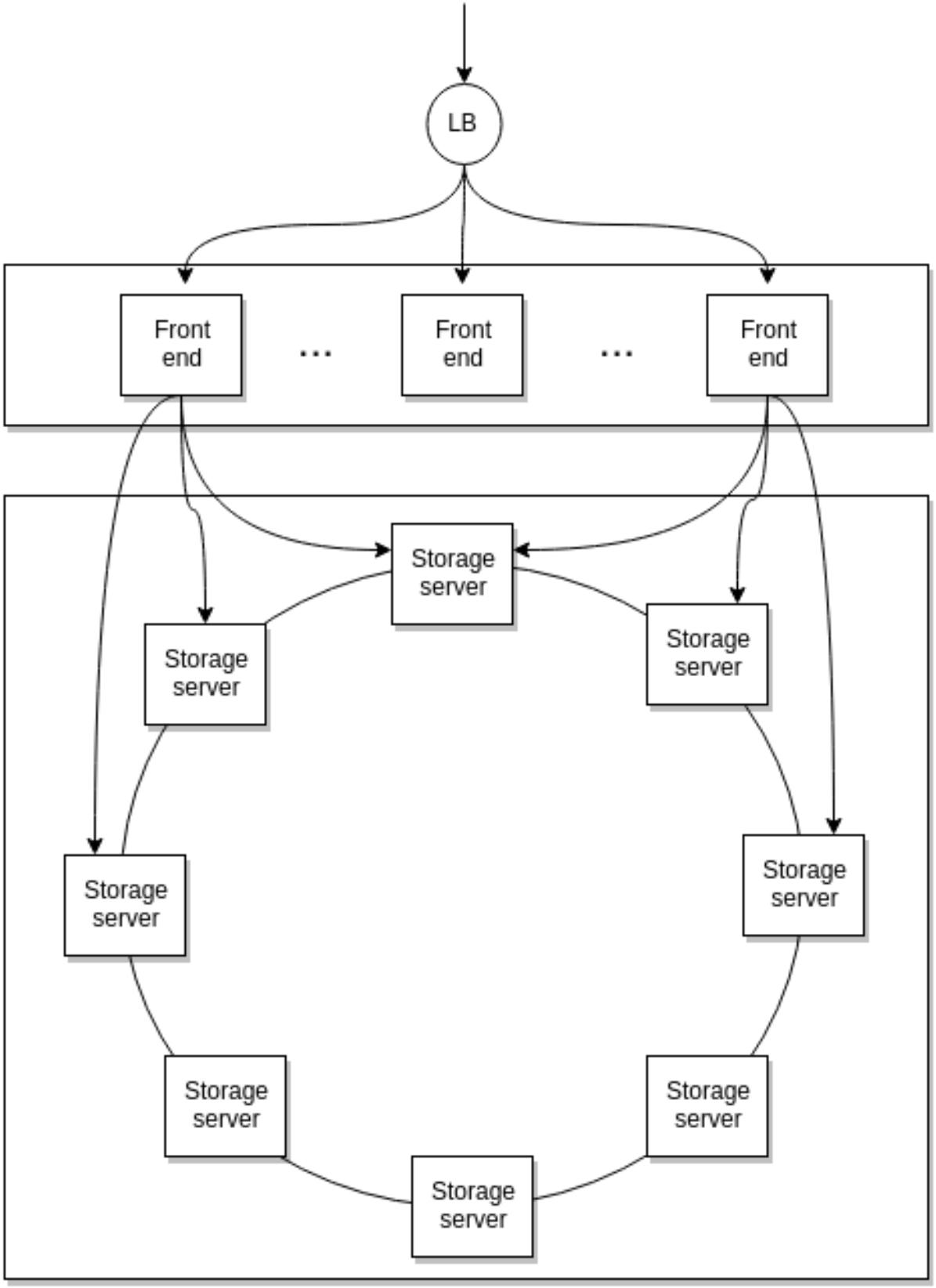

Figure 1: Project logical structure

correctly served or not. If no message is received, it is assumed that something went wrong.

2.2 Storage protocol

All the storage nodes use consistent hashing to assign a key-value pair 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 subsequence servers in the consistent hash ring;

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

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

Within the data it is also added a vector clock to keep track of which server updated the value. The vector clock is implemented using a map where the key is the server id and the value is a counter; the servers that are not present in the map are considered with a 0 counter.

So each time a server updates a value as a master it increments the counter with its id inside the object, and forwards this new vector with the key-value to its backup servers.

This vector clock is used every time two version of a value must be compared to decide which is the most recent. If two uncomparable versions of a value are found, the server node creates a value with the two different versions and sets the `conflict` flag.

At this point when a user attempts to get a key, he/she receives all the conflicting versions and can decide which one is the correct version by performing an update operation.

After the update the server resolves the conflict and merges all the vector clocks together. For so, subsequent incoming old values for a key 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 implements a distributed file system with a single API server, where each node run on threads.
- **webapp**, this is a nodejs application that graphically shows the state of the file system.

Figure 2: screen-shoot of the MonitorWebApp

3.1 Core

The main classes in the core project are:

- `Messages`, and its children classes – `MessageManage`, `MessageRequest<T>`, `MessageResponse<T>`, `MessageStatus` – that represent JSON messages exchanged between the file system nodes.
- `ConsistentHash<T>`, the implementation of a Consistent Hashing owned by each storage nodes.
- `Data<T>`, represent a generic key-value stored 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.
- `GossipResource`, 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 both the gossip server and the management service. It also has a `protected` method `send(Node n, Message msg)` method to send a message to a given node through the `DatagramSocket`.

This class is extended by the `StorageNode` and the `GossipResource` 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 initializes all the needed structures, i.e.:

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

It has a thread that continuously checks new messages from the other nodes, and for each one it distinguishes two general cases: either it is a new request or a `MessageManage` with some operation sent by the master to add or modify a backup key.

3.1.2 PersistentStorage

This class implements the storage on file using a B-Tree Map $\langle Long, Data \rangle$ – implemented by the library `MapDB` – this is because it is possible to retrieve in $O(\log n)$ all the keys that have the hash in a 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 found on the network thanks to the gossip protocol, it is simpler to search the keys that it has to manage as master or the keys that it has 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 MonitorWebApp 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 JOSN 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`, tests the methods `add`, `get`, `getPrev`, `getNext` of the `ConsistentHash` class.
- `DataTest`, tests the methods to manage multiple conflicting versions of the same data in the `Data` class.
- `FSTest`, tests the threaded version of the file system with 10 nodes, it checks all the implemented operations (get, add, update, delete).
- `ModServerTest`, checks if the procedure to add or remove a node of the file system works.
- `StorageTest`, tests the `add`, `get`, `update`, `delete`, methods of the `PersistenStorage` class.
- `VectorClockTest`, tests the `increment`, `update` and `compareTo` methods of the `VectorClock` class.

Given the latter test classes also a coverage test analysis is done to understand 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 User Guide

It is possible to use the distributed file system in the multi-threaded version for a single machine, the multi- server for a cluster of machines 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 multi-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 built 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 built 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 running 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

```
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 web-app can be used with one of the released versions for the different OS's or run with the Node.js 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 easily 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.
- NW.js, 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 plug-in for Gradle.
- JitPack, to build 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.