A distributed file-system: PADfs

Distributed-enabling-platform course (prof. N. Tonellotto)

Student: Andrea Tesei

Master degree in Computer Science and Networking
University of Pisa and SSSUP Sant'Anna

-a.a 2015/2016-

Table of Contents

1. Introduction

The *PADfs* is a distributed persistent data storage, which can store all type of data as key-value pairs (filename, content). It's a ***flat*** file-system: so there isn't any hierarchical structure and all data inside the file-system must have a different name.
This project was developed in Java (version 8), use Maven as project management software and all source code can be found at: https://github.com/andrea-tesei/PAD-Repository/tree/master/PADfs.

It is distributed: so it deals with all (actually a part of) issues of distributed programming. The design choices were made taking to account all the distributed system theory seen in the lectures of this course together with all external references.
With respect to the CAP theorem, this file-system is a CA application; the *consistency* model adopted is the *monotonic-write* one; the *availability* is guaranteed by replication of each *node* of the file-system. However I will describe in detail all this things and other design choices in section 3.

What about missing parts? This file-system doesn't have any quorum system implemented, which actually can be a good idea for future works on this project since with this choice maybe the temporary failures handling can be performed in a more secure way; another simplification is the replication factor equal to one: this is the biggest limitation of this file-system, since you can imagine that if these two *nodes* (the master and the slave/replica) dies, you lost a part of the files until this *nodes* comes up; it doesn't have any prevention and detection system for permanent failures, which actually can happen in a distributed system, but covers only temporary failure with replication (prevention/handling) and gossiping (detection).

The rest of this report will describe in details all the above things. In the second section, the architecture of the file system and the structure of the source code are presented and explained. In the third section, I will expose all my design choices with a little explanation, and I will explain even all the simplification adopted where it is relevant. In the fourth section, I will present the client's interfaces that can be used to access the file system running. In the fifth section, few lines on how to run the PADfs *node* and the relative client. In the sixth section, the tests made are presented and discovered issues during test phase will be explained. In the seventh section, a couple of ideas for future works on this project are listed.

2. Architecture

We start from an high level view of our distributed system, composed by *PADfs nodes* and a *Client* which interacts with them via a command-line REST-based interface:

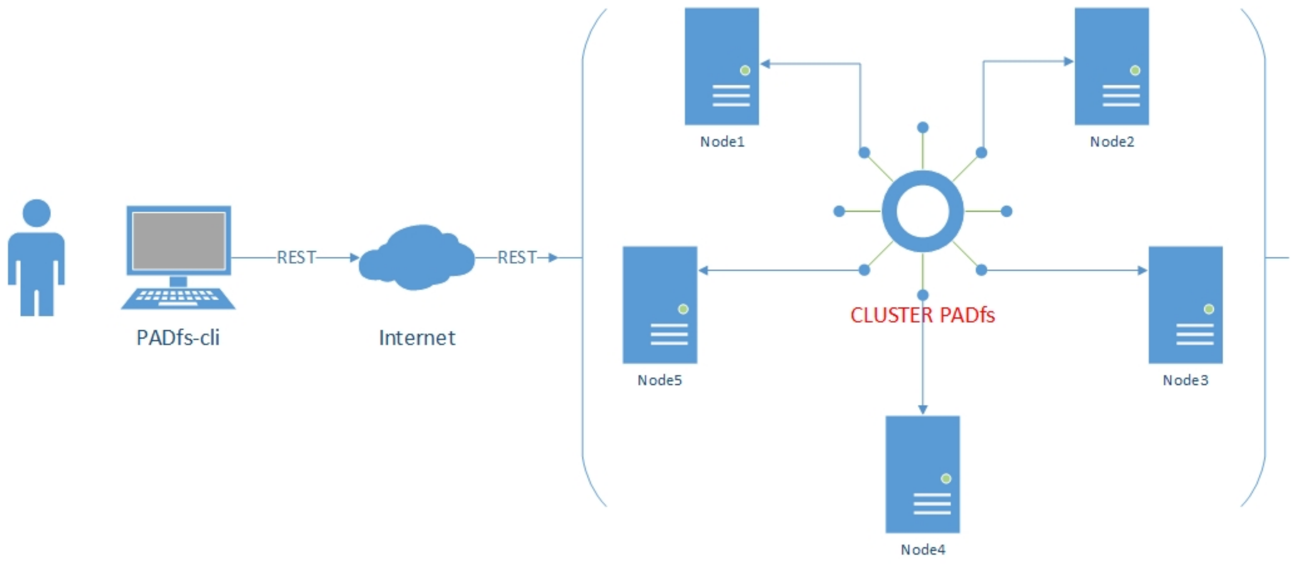

Figure n.1: High level view of the PADfs system. Each *client* can contact each *node* for command executions. The *nodes* communicate each other within an UDP socket: this channel is use for internal operation.

In the figure n.1 is represented a network of five PADfs *nodes* and one client.
Each *node* is equipped with a set of RESTful web services exposed to the client, and an UDP socket interface, useful for all communication between *nodes.* The circle-like organization of the *nodes* is imposed by the structure of Consistent Hashing, as I will explain in more details in the next section.
In the figure, we can see all communication links between the "actors" of this "game": each client can send a request to one of the active *node* of the system, even if he request for a file which the contacted *node* isn't responsible for (active replication explained in section 3).
The operation exposed to the clients are five (REST-based):

- GET: for download a file from the file system;
- PUT: for put in the storage new file, or override/update an existing one;
- LIST: for retrieving the list of files currently on the file system;
- DELETE: for delete a file from the file system;
- CONFLICT_RESOLUTION: for resolve conflict with files.

In the next figure, we can see the internal structure of each *node*:

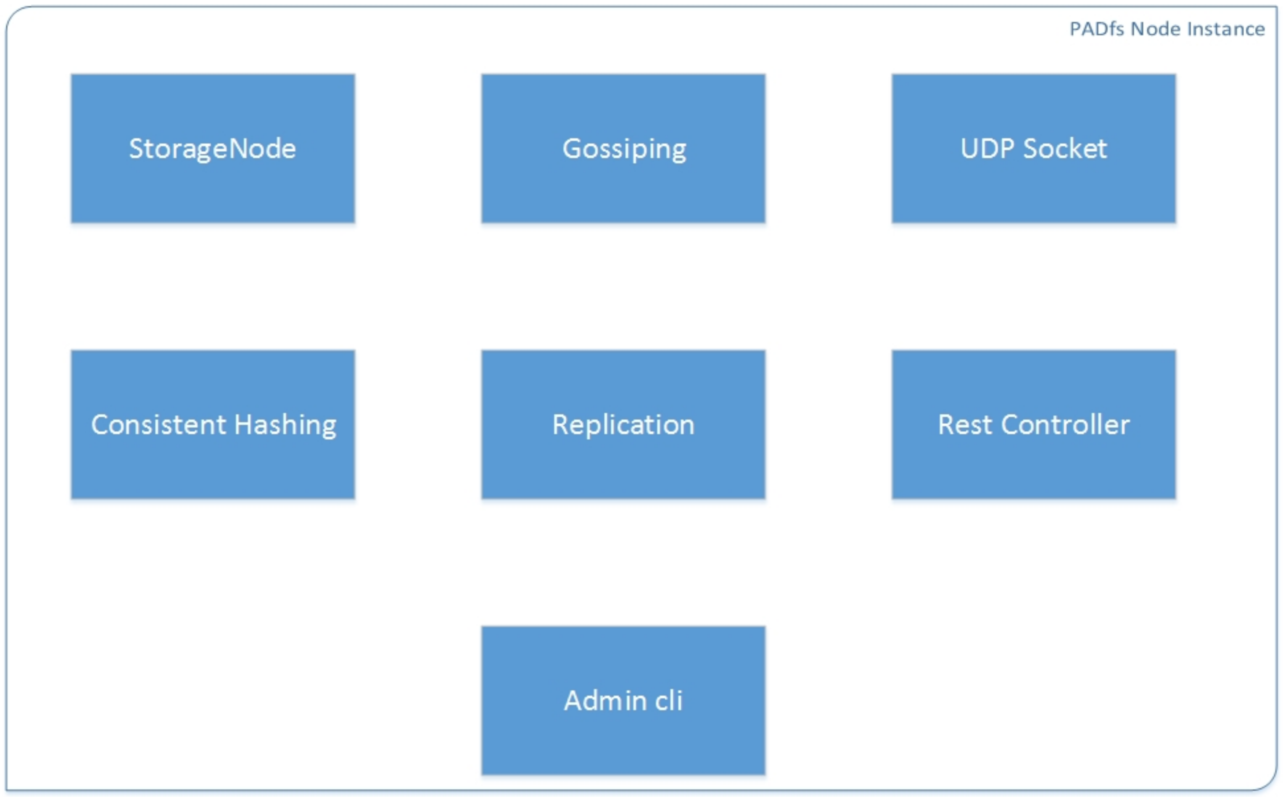

Figura n.2: the internal view of each *node*.

As we can see, each *PADfs node* is implemented by the *StorageNode* class, which is composed by several components/features:

- **Gossiping**: for new *node* discovery and failure detection;
- **Consistent Hashing**: which acts as a load balancer;
- **UDP Socket**: used in inter-*node* communication and composed by a sender and a receiver thread: the former is called by *StorageNode* whenever a send message is required; the latter interacts whit *StorageNode* thread via listener whenever a new message arrives and has to be processed;
- **Versioning**: which is implemented with VectorClock (see section 3) and is used for versioning the file inside the file system and to discover conflict inside them;
- **Replication**: each *node* is responsible for the replication of its files in a different *node* (see next section for the details of implementation);
- **Rest controller**: which exposes the methods above for the client;
- **Client**: a little command interface inside each *node* helps the Administrator to know about the cluster status and file to *node* mappings.

So each *node* is responsible for a set of files put by the clients, for their replication and update in *replica's node*, and it has either to serve each request arrives from a client (via REST controller) or from another *node* (via UDP socket). The **Consistent Hashing** component help to spread the load on the set of active *nodes*: thanks to **Gossiping** component, each *node* is able to construct its view of the network and knows within few seconds either whenever a fail occurs in some *node* or when a new *node* comes up. As said, those arguments will be discussed in more detail later.

2.1 Source code structure

Now we can focus on the structure of the source code. It is composed by seven packages included in the folder PADfs/src/main/java:

- **it.cnr.isti.pad.fs.entry**: is the package containing the entry point of the program which starts the REST controller and the StorageNode instance (App.java);
- **it.cnr.isti.pad.fs.event:** this package contain the implementation of the events triggered when new message arrives (OnMessageReceivedListener.java), when the response asynchronous handler finishes to process a response (OnResponseHandlerFinished.java) and last but not least when some message has to be send outside (OnSendMessageListener.java);
- **it.cnr.isti.pad.fs.gossip:** is a wrapper of the external Gossiping library from Edwuard Capriolo's github repo;
- **it.cnr.isti.pad.fs.runnables:** this package contains the Runnables of the set of thread: first the one which send message on-demand (StorageSenderThread.java); then the one which behave like a daemon and wait for new incoming messages (StorageReceiverThread.java); the thread which await response message asynchronously (StorageResponseAsyncHandler.java); and finally the one which is used as shutdown hook (StorageShutdownThread.java);
- **it.cnr.isti.pad.fs.udpsocket:** contains the set of classes needed for the client and the server udpsocket, used to communicate with the other *nodes.* Both of server and client sockets implements the interface IUDPSocket.java which contains the signature of the common methods (send, receive and closeConnection); then there are UDPServer.java and UDPClientsHandler.java: the former is a server as you can imagine, the latter wrap a set of SocketRemoteInfo, each one representing a remote *node*, and contains all the information to send a message to another *node*; then we have the various class defining the structure of the messages, the various command, return code and type of messages (Message.java

contains the inner classes Type, Command and ReturnCode; StorageMessage.java is the real format of each message);

- **it.cnr.isti.pad.fs.udpsocket.impl:** here there is the implementation of the sender and the receiver thread for the UDP socket (StorageReceiverThreadImpl.java StorageSenderThreadImpl.java);
- **it.cnr.isti.pad.fs.storage:** this package contains the core of this application. There is the StorageNode.java class which we have discussed above and the Data.java class which models a file in the file system;
- **it.cnr.isti.pad.fs.api:** which contains the implementation of the web services exposed to the clients (ApiControllers.java)

The logger chosen is log4j, which properties can be found in PADfs/src/main/resources, and give to the Administrator two log files: *padfs.log*, including all log about StorageNode operations, received and sent messages ecc; *gossiping.log*, which print all messages from the Gossiping library.

2.2 Message format

The format of the messages is defined in the StorageMessage.java class but it is variable with the type of the message considered:

- **Host**: is the host of the sender *node;*
- **Type**: can be REQUEST (type = 0) or RESPONSE (type = 1);
- **Command**: can be one of these commands:
    - GET/PUT/LIST/DELETE;
    - PUT_BACKUP: used to send whole backup copy to a *node*;
    - UPDATE_BACKUP: used to update single backup copy;
    - DELETE_BACKUP: used to delete single backup copy;
    - ERASE_BACKUP: used to erase the whole backup copy in the receiver *node;*
    - CONFLICT_RESOLUTION: used in conflict situation to resolve it;
- **Idrequest**: the identifier of the current request (selected by the *node* which starts communication with a REQUEST);
- **Returncode**: it can be one of these rc:
    - OK/ERROR;
    - NOT_EXISTS: the requested file does not exists (for GET and DELETE);
    - CONFLICT_EXISTS: this is used in conflict situation (response for a GET request);
- **Output**: is an array of files (with or without content) used in case of LIST and GET response when conflicts exist;
- **Data**: contains the file with its metadata. It is used in all single file command like PUT, GET(response), UPDATE_BACKUP, DELETE_BACKUP;

- **Filename**: the name of the requested file;
- **Backup**: used in PUT_BACKUP message.

3. Design choices and Implementation

For what concerns the design choices, as said, this system is a CA one: ***consistency*** and ***availability*** are guaranteed with different mechanism, which I will present later on.
Let's just list all the topic choices:

- **Consistency**: the consistency model adopted is the *monotonic write* one, which is a Client-centric consistency model guaranteeing ordering on write operation by a process on a certain data item. When a PUT command arrives to a *node* which is responsible for the given file, it updates the content of the file, increment its version, and when all operation on this file are completed successfully, it returns the "OK" message to the client; if an error occurs, it just sends back an error message to the client;
- **Replication**: as said before, the *data* replication factor is k=1 for each file. Each *node* is responsible of the update of its backup file in replica *node*: when a PUT arrives, it has to complete the local store and sends back to the client the "OK" message; then it sends a message to its replica *node* for this file and wait for response without inform the client of the result of this operation. Each *node* is able to manage its set of data and the relative backup copies: the clients aren't aware of this. The choice of the replica *node* is driven by the *ConsistentHashing* structure: for each file, the next *node* in the circular array is chosen and the set of backup files sent to it.
  This is actually an *active* replication, since each client can retrieve each file requesting it to one of the active *nodes* in the system: if this *node* is responsible of the file, it will prepare a message containing the content of it; else it will sends to the right *node* this request, waiting for result to give back to the client;
- **Availability**: the availability of the whole file system is guaranteed by the replication implementation, which tolerate temporary failures;
- **Partitioning and load balancing**: for this kind of problem I use the implementation of ConsistentHashing from the Gomathi's repository (https://github.com/gomathi/ConsistentHashing latest version).
  The implementation provides also the use of virtual buckets to spread the load factor of each *node*: this is set to the $\log_2 N$, where N is the starting number of *nodes* in the system. This protocol put all bucket (a.k.a. *nodes*) and all members (a.k.a. files) in a circular array with a hash function (in this project I use SHA1 hash function): this guarantees the balancing of the

load factor within the *nodes* of the system. The set of files for a given *node* are the ones stored between it and its *predecessor* in the array. Starting from this implementation, I extended it with several methods as: given one *node*, retrieve previous and next *node* in the network; retrieve which files has to be send to the replica *node* or to a new *node* spawned; retrieve the set of virtual buckets for a given *node*;

- **Versioning**: to resolve this problem I used the VectorClock implementation from the Linkedin's Voldemort project (version seen during lectures), extended only with two methods for JSON conversion. The VectorClock structure is a pair <id:n> where id is a short number representing the id of the *node* and n is the relative "progress" done for the relative file in the id's *node*;
- **Conflict resolution**: the conflict resolution problem is delegated to the clients: when a conflict on some files is detected, the two or more versions are saved locally to the *node* which has detected the conflict. When the client request the file later on, the responsible *node* gives back all versions saved and wait for a decision from the client;
- **Gossiping**: I use the implementation of Gossiping protocol from the Edward Capriolo's repo (https://github.com/edwardcapriolo/gossip/) (version 0.0.4) extended only to parse also the hostname of the given *node* in order to retrieve the correct IP address in the code;
- **API**: for developing the REST controller I used Spring framework (version 1.3.5 RELEASE).

3.1 Implementation of the client commands

I will present now the behaviour (actually the implementation protocol) of the operation exposed to the client: GET, PUT, LIST, DELETE. We discuss all this arguments considering a general scenario in which we have five *nodes* active in the system (so the number of virtual *nodes* for each of them is equal to 2).

GET

This operation can be requested to each *node* active in the system. We now focus on the most complicated case in which a client ask for a file to a *node* which is not responsible for it:

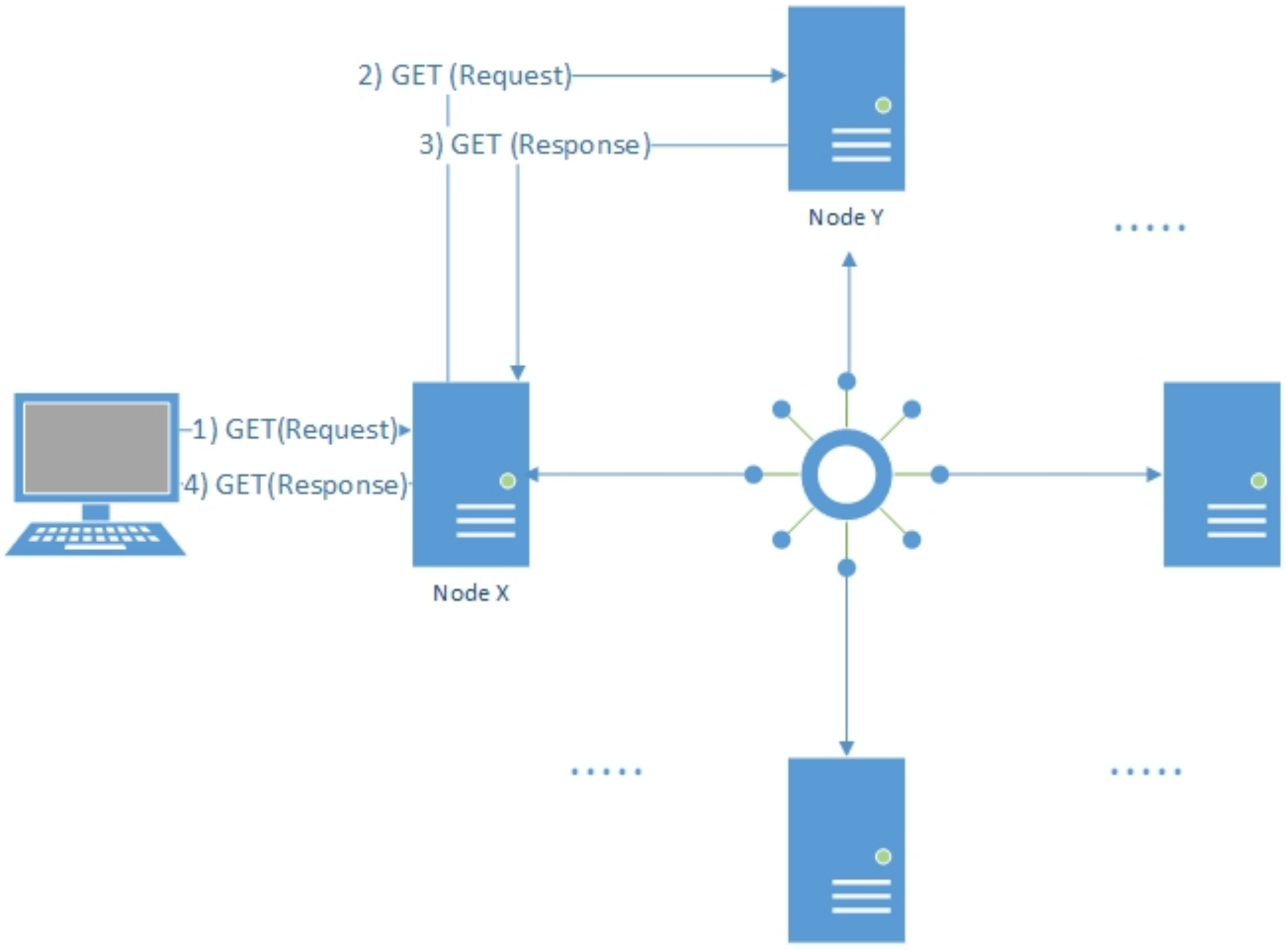

Figure n.3: An example of remote GET request. Remote means that the server which accepts the request is not the one that stores the requested files.

If the *node* responsible for the requested file, finds some conflict saved before, it has to send back to the client the set of versions in which the client has to choose the right one. Also in this case the intermediate *node* drives the communication between client and the responsible *node:*

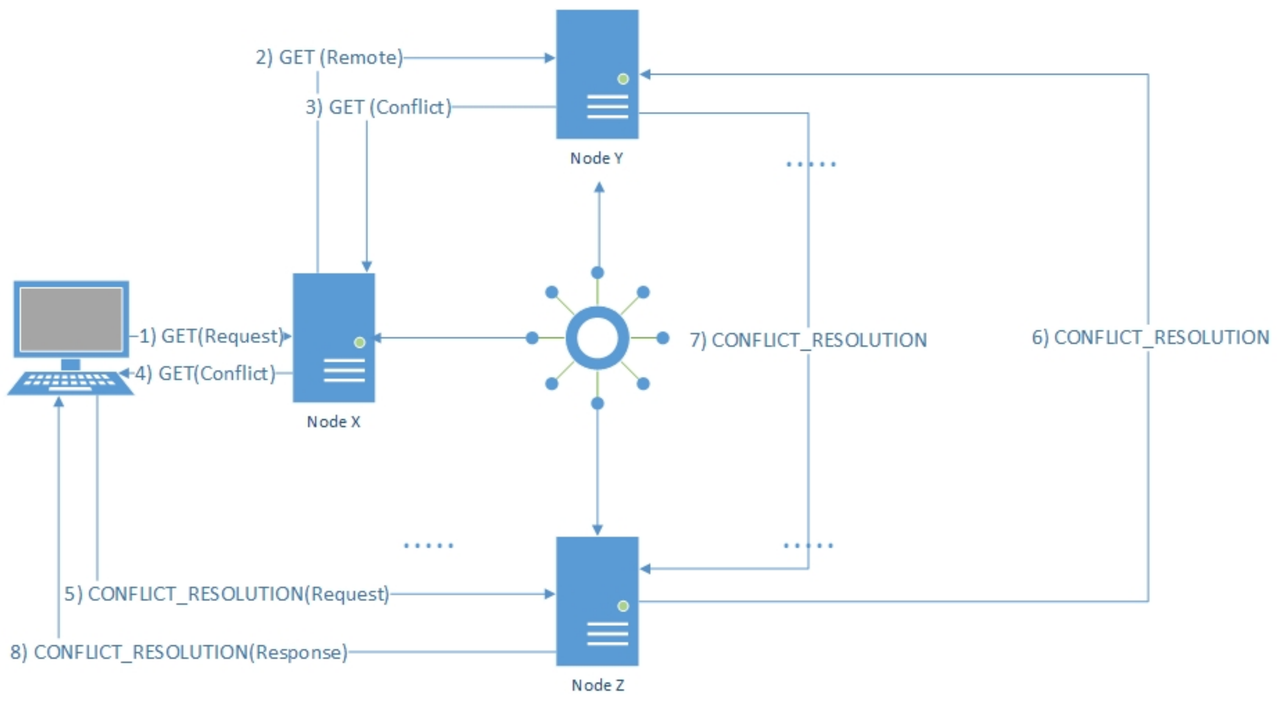

Figure. 4: an example of conflict resolution case. The GET request is served by Node X; the other CONFLICT_RESOLUTION request is server by Node Z.

If the requested file does not exist, a NOT_EXISTS message is send back to the client.

PUT
Also in this case, we focus on the most complicated case in which we ask to insert a new file to a *node* which is not responsible for the given file: the path is the same as in Figure 3, with the difference that here the request have the new file inside it (and of course those are PUT request☺).
In case of unsuccessful PUT, an ERROR message is returned back to the intermediate *node*, which gives it back to the client.

DELETE
Even the deletion procedure is equal to a GET command (look at Figure 3): the client sends the name of the file he want to delete. If the file does not exist, a NOT_EXISTS message is send back to the client. If an error occurs while deleting this file, an ERROR message is send back to the client.

LIST
This type of operation can be requested to each *node*, as usual. The *node* that receives this command sends to all other *nodes* a LIST command and wait for results. Finally, it packages all the names received by other *nodes* and send back to the client the result.

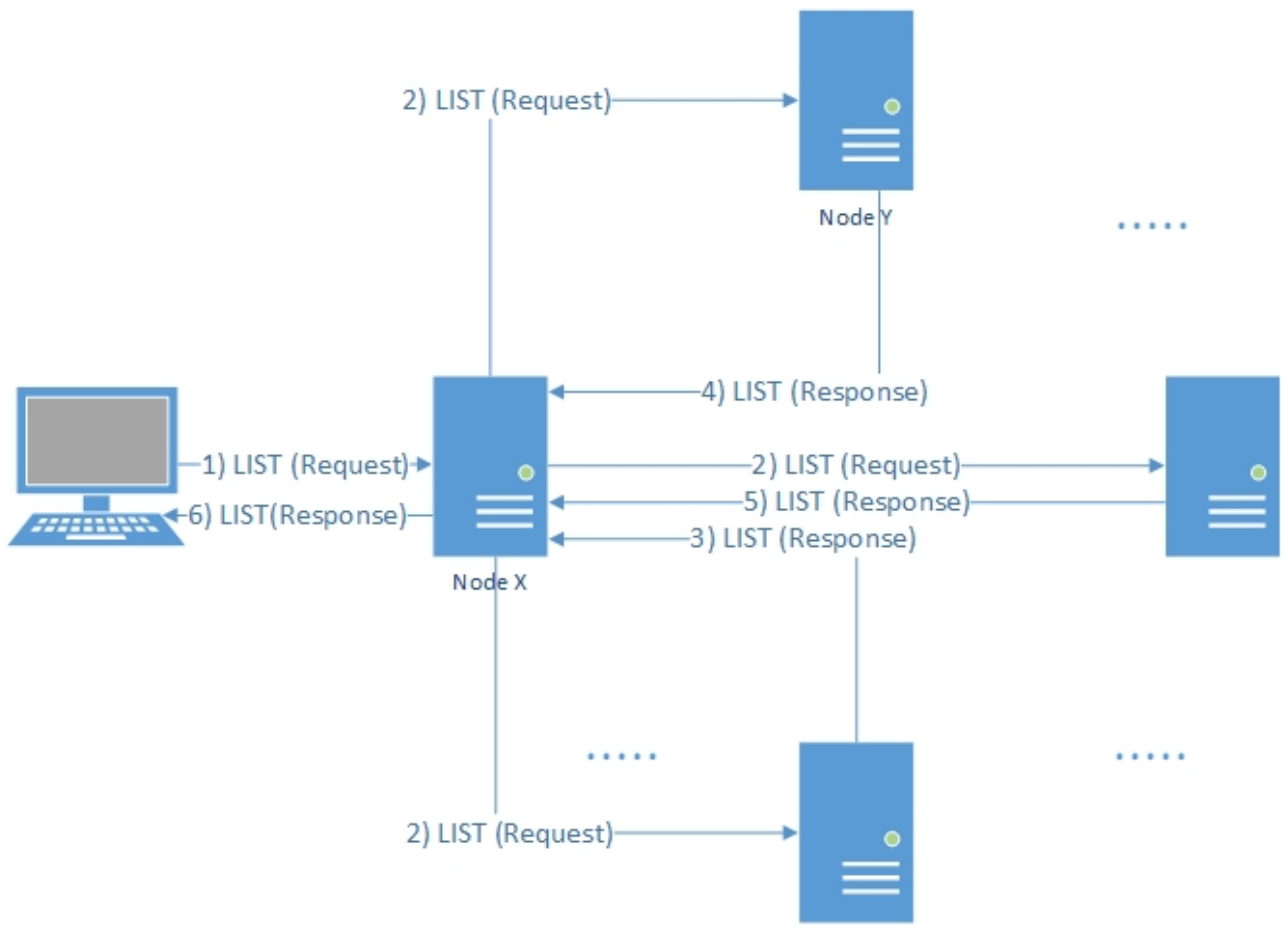

Figure n.5: an example of LIST request by the client. The Node X receives the request and ask for LIST to each node of the network: once they answer, it packages a new LIST message containing the list of files.

3.2 Failure detection and recovery

When a failure occurs, each *node* detects it within few seconds, thanks to Gossiping protocol. In this situation, each *node A* checks whenever the *node B* in fault is a predecessor or a successor in the array of ConsistentHashing: in the former case, since *A* acts as backup *node* for *B,* it has to take all backup data previously received, put them in the set of file it is responsible for and sends them to the replica *node* (actually, for each file a new backup *node* is selected as the successor in the ConsistentHashing starting from the virtual *node* which "logically" store the given file); in the latter case, since *B* was a backup copy of a subset of the files stored in *A,* a new backup copy *node* has to be selected and relative backup files send to it.

Similarly, when the *node B* previously died comes up again, each *node A* checks if it has some files that *B* has to receive and send them to it. This procedure ends up with the response from *B* with an "OK" message, which tells to *A* that it can remove and delete from disk the selected file.

4. Interfaces: API and command-line clients

As said before, this project includes a set of RESTful web services which implements the commands exposed to the clients. A REST-based command-line Java client is included in the project. Now I present all the REST endpoint and the relative command and syntax in the command-line client:

- Command **GET:**
  - REST path: <host:port>/PADfs/API/GET?filename=[name]
  - Request method: GET
  - Parameters: filename:String
  - Command-line syntax: GET:filename
- Command **LIST:**
  - REST path <host:port>/PADfs/API/LIST
  - Request method: GET
  - Command-line syntax: LIST
- Command **PUT**:
  - REST path: <host:port>/PADfs/API/PUT
  - Request method: POST
  - POST parameters: filename:String, file:String (base64 encoded)
  - Command-line syntax: PUT:filename
- Command **DELETE:**
  - REST path: <host:port>/PADfs/API/DELETE?filename=[name]
  - Request method: GET
  - Parameters: filename:String
  - Command-line syntax: DELETE:filename
- Command **QUIT** (only for command-line):
  - Command-line syntax: QUIT

An administrative command-line interface is also attached to each instance of PADfs: it gives the possibility to ask for cluster status and know about alive *nodes* and died ones, or simply print all mapping between *nodes* and files. The following list contains all the available commands:

- **CLUSTER_STATUS**: prints all *nodes* with UP/DOWN flag;
- **NODE2FILEMAPPING**: for each *node*, print all files store on it;
- **HELP**: to print a similar list like this;
- **QUIT**: to power off the *node*.

5. Running the project and the client

In order to execute this project, you have to type this command:

```
java -jar PADfs-1.0.jar gossip.conf
```

where gossip.conf is the JSON configuration file of gossiping protocol and it has the given structure:

- ***id***: identifier of the running *node*;
- ***port***: udp port for the gossiping protocol
- ***gossip_interval***: the time between two consecutive Gossip messages;
- ***cleanup_interval***: the timer length for failure detection;
- ***hostname***: the hostname of the running *node*;
- ***members***: array of other *nodes* with IP address, port and id.

To execute the client, simply type this command:

```
java –jar PADfs-cli.jar known_hosts.conf
```

where known_hosts.conf is a file containing the array of known *nodes* possibly active in the system (syntax: one IP address for each line).

6. Executed tests and discovered issues

I've made some tests during the debugging phase in order to discover both of issues with the code and strange behaviour of the file system. During this phase, I've discovered some problems that are still present in the file system. The first problem (actually a limitation) is correlated with the design choice about the use of UDP socket to send messages between *PADfs nodes*. This issues concers the maximum packet's length admissible by the UDP protocol: this is set to "65,507 bytes" which is the real length imposed by IPv4 protocol (65,535 – 8 byte UDP header – 20 byte IP header)[1]. To overcome this problem, the system must have some extension to deal with files larger than this number, maybe applying some fragmentation of this packets or something like that. This is a good starting point for future works.
The second problem is related to the Gossiping library: I'm trying to use it on the cluster in figure n°1, with five *nodes* which starts almost simultaneously.
I've discovered that there is an initial phase in which the system seems "not stable" in the sense that some dead buckets are discovered, also if they are alive. This is such a *transient phase* in which the Gossiping module stabilizing: in fact,

[1] https://en.wikipedia.org/wiki/User_Datagram_Protocol

after a few seconds (on average) the various *nodes* start to behave normally and they know each other. In my opinion, it is something related to the traffic generated when all *nodes* start together, but I'm still thinking a way to overcome this problem.
Another big limitation of this file system is that, as I said, the replication factor is equal to 1: if two consecutive *nodes* on ConsistentHashing array die at the same time, some files will be lost.

The executed tests concern the behaviour of the system in presence of faults.
I will present a little review of the test phase with two interesting cases:

- I've made several PUT commands (Put file X -> arrives in *node* 1; Put file Y -> arrives in *node* 5; Put file Z -> arrives in *node* 4;) then first I've killed the *node* 4 and after a few minutes the *node* 5: when *node* 4 dies, *node* 5 bring the file Z in its files (actually move the file Z from backup set to the current file one) and send its replica copy to new backup *node* 1. At this point I type a GET of Z and all is ok; then when *node* 5 dies, *node* 1 brings its files and now it has X, Y, Z in its set of current files. At this point I've restarted *node* 4 and *node* 5 and they successfully received the files from *node* 1 (and actually compare their version with the local one (previously saved when killed)) as soon as it received the "new bucket" notification from Gossiping module.

  Outcome: OK

- The second test case concerns the behaviour of the nodes when at start-up time they discover some previously saved files: when a *node* is powered off, it saves in the "files" folder a JSON file (storagedata.json), which is parsed at start-up (if present). Actually the files are stored in this file with the JSON representation of their VersionClock. So if we consider the scenario of the previous test case, when *node* 4 and *node* 5 comes up they parse their JSON file and store the data locally: then they receive from *node* 1 the actual version which is compared to the previous one. No conflict arises even if a modification is performed while *node* 4 and 5 stay down.

  Outcome: No conflicts generated

7. Future Works

As future works, I think that first this project has to be extended to supports multiple backup copies (k > 1), in order to reach a more reliable availability and fault-tolerance in the file system. Also the limitation about UDP packets length mentioned in the previous section has to be fixed: a fragmentation of the packet is needed when they exceed the permitted length.
Another good thing is to implement a quorum system in order to manage master and slave copies and also to be more stable and secure for what concerns the temporary and permanent failure tolerance. Together with this last point, an implementation of a permanent failure detection and recovery protocol is needed in order to reach a more stable system.
Then, this file system can be extended in order to have a hierarchical structure and to provide user authentication and relative permissions of read/write operation: this is a very basic property of each file system, but for simplicity reasons this project doesn’t cover this feature.