University of Pisa
Scuola Superiore Sant'Anna

Department of Computer Science
Department of Information Engineering
And Scuola Superiore Sant'Anna

MASTER DEGREE ON COMPUTER SCIENCE AND NETWORKING
(Class LM-18)

Distributed Enabling Platform Project

Distributed File System

Stefano Ceccotti 456568

Accademic Year 2015/2016

Contents

Chapter 1

Introduction

The project of Distributed Enabling Platform consists in creating an eventually consistent distributed file system. The file system is built in a hierarchical way and it provides simple operations to play with it:

- **get**(key): to retrieve a file from the system;
- **put**(key): to store a file in the system;
- **delete**(key): to delete a file in the system. If the target is a folder all its content will be recursively deleted;
- **getAll**(): to retrieve all the files stored in a random node.

The system makes use of Gossiping, Virtual Nodes, Versioning (namely Vector Clock), Hinted Handoff, Anti-Entropy and Consensus (a.k.a. Quorum) protocol. The Anti-Entropy mechanism ensure that the system, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value (from here the eventual consistency). Using hinted handoff, we ensure that the read and write operations are not failed due to temporary node or network failures.

Chapter 2

System Architecture

The project is composed of 3 entities: **Client**, **LoadBalancer** and **StorageNode**, arranged in the following way:

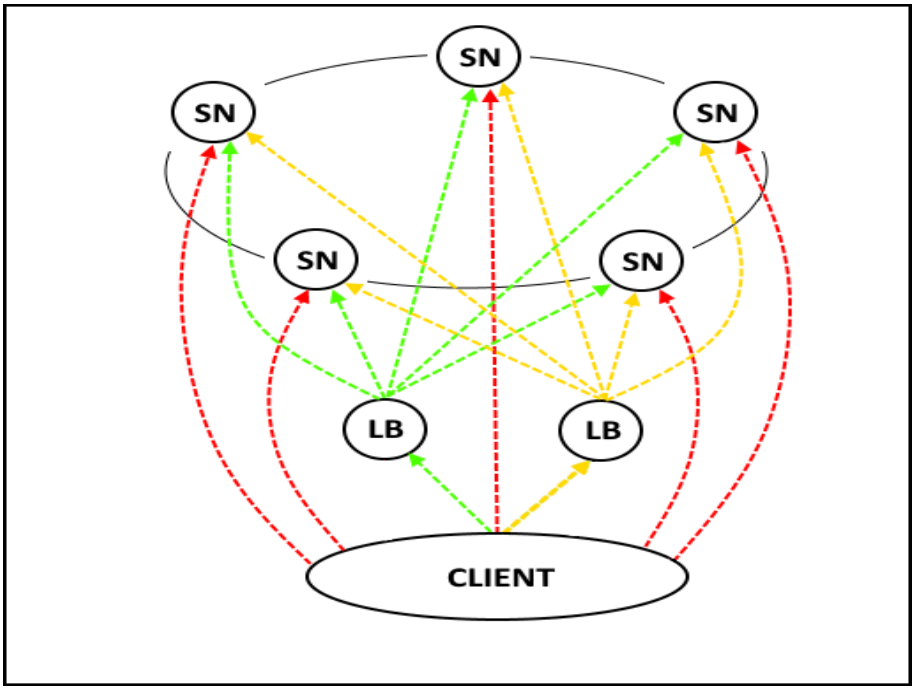

Figure 2.1: Abstract version of the system. The green and yellow dashed lines are possible paths that a request may follow when the client uses **LoadBalancer**s, while the red dashed ones are direct connections to the **StorageNode**s.

- **Client**: interacts with the user and transmits the message requests;

- **LoadBalancer**: receives the client requests and forwards them to the correct node based on the their workload;
- **StorageNode**: receives requests either from load balancers or directly from clients. It also provides a way to store the files in a consistent manner.

Most of the connections use TCP in order to have a reliable mechanism, especially when files are transferred, at the cost of a little overhead due to the connection establishment.
To reduce this latency the quorum phase makes use of the so-called **Reliable UDP** [1]: it's a network protocol that combines the speed of UDP (small packets) and the reliability of TCP (acknowledgment and retransmission). Every connection has a timeout of 2 seconds (except from StorageNode to Client that are 5 seconds), after that, if it has not been established, means that the destination node is dead or something wrong happened.

2.1 StorageNode

The StorageNode is the entity used to store the files received by client requests. Everytime a new request arrives, a new Thread is activated (from a fixed ThreadPool) and a TCP connection is open: if the request arrives from a load balancer, then a direct TCP connection is opened with the client, closing the initial one (Figure 2.2). This is done to reduce the workload on the load balancers, and to eliminate the triangular problem, similar to Mobile IP [2].

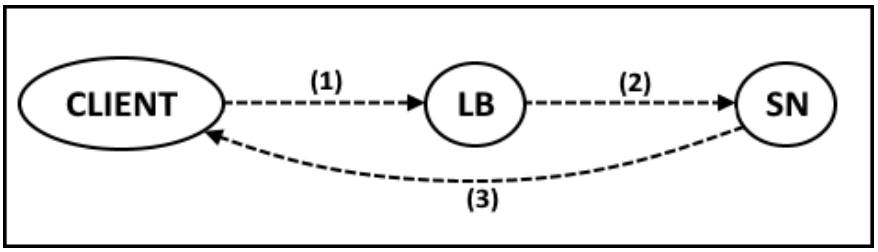

Figure 2.2: Communication pattern.

The logical structure of the node (Figure 2.3) is the following:

- **MembershipManagerThread**: Thread responsible to receive the membership requests from a client;

- **QuorumThread**: Thread responsible to perform the quorum protocol;
- **AntiEntropyThread**: Thread responsible to perform the anti-entropy mechanism;
- **MonitorThread**: Thread used to monitor the state of StorageNode instances;
- **FileTransferThread**: Thread used to send/receive files;
- **NetworkMonitorThread**: Thread used to send informations about the own workload;
- **Gossip**: used to discover and check the liveliness of a cluster;
- **DFSDatabase**: Class used to manage the files stored on it.

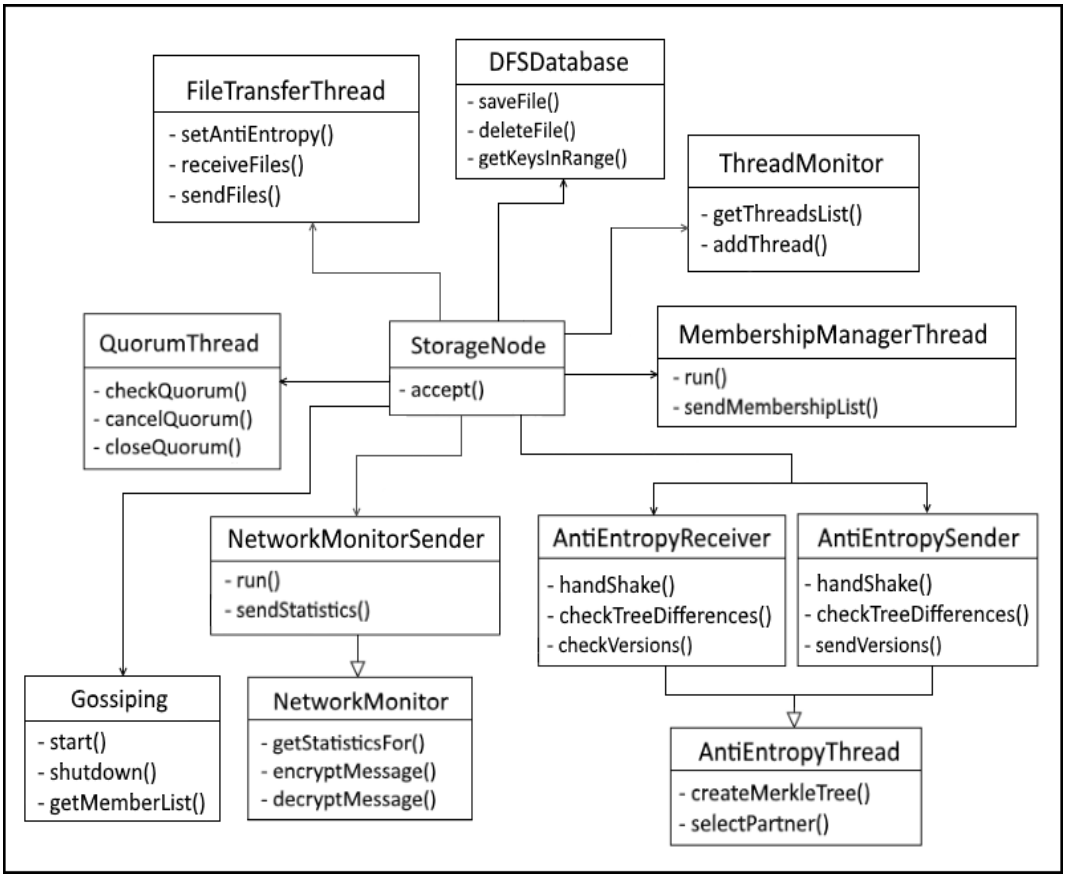

Figure 2.3: StorageNode structure.

When the port of the node is setted, it's assigned to the gossiping protocol, while the next 5 consecutive ports are assigned to different threads belonging to the node, included the StorageNode listening Thread.

Every StorageNode is in charge of all the keys whose id is less or equal than the node's identifier. The node id is computed applying the SHA-1 hash function to its Ip address, port number and virtual node instance.
The virtual node mechanism is a technique used to distribute, in a more uniform way, the nodes inside the DHT using the so-called "tokens" as a logical replication of a phisical node. The number of tokens can be either setted as an input parameter that will never change or dynamically computed as the logarithm of the number of nodes present in the system.

2.1.1 Storage System

All the files are stored using an ordered B-Tree structure of type (String, DistibutedFile) to efficiently put, get and delete files in memory and on disk as well. The key used to store and retrieve files is a SHA-1 hash function applied to the name of the file.
When a file is deleted it's not intended to be definitely removed from the database, but only marked as deleted. The file's metadata is keep on database until its TimeToLive is not expired. This is done to share this information when the anti-entropy is executed. The TTL of a file is about 1 hour.

Each file can be accessed simultaneously only in read mode, otherwise (write mode) in a mutually exclusive way [3]. To reduce the latency inside the operations, due to the access on disk, an **AsyncDiskWriter** Thread is enabled by default. In this way all the modifications are performed in asynchronous mode and do not block. Without async writes, all threads remain blocked until all previous writes are not finished (single big lock). The main drawback rises during a disk-read request: it could waits for all the pending operations for that particular file.

Along with data it is associated a **VectorClock** object, that keeps track of all the nodes that have updated the file. A VectorClock is effectively a List of (node, counter) pairs, where the counter is increased by one by the Coordinator node (the node in charge of the key). If more than 10 versions of the file are present, the one with the oldest timestamp is discarded.

When a file is added or removed its hinted handoff value is checked: if different from *null* the file will be stored (or deleted) also in a different database inside the **HintedHandoffThread**: this Thread saves all the files associated to a specific destination node and, periodically (every 10 minutes), tries to send out the associated files: once the transfer succeeds, the files will be removed from its local store without decreasing the total number of

replicas in the system.

2.1.2 Quorum Protocol

If a node receives a GET_ALL request then it simply returns all the files stored in its database, otherwise a quorum mechanism is started; this node is defined as Coordinator node. If the quorum is reached the requested operation is performed, otherwise a negative response is returned to the client. The quorum can be reached only if at least N/2+1 of the contacted nodes agree to it, for both read and write requests. The quorum request is managed by a background Thread called **QuorumThread**, which contains a HashMap of (String, QuorumFile) to efficiently checks if the requested file is locked or not. In addition, a unique identifier is associated to each request to prevent some mistakes due to reordered commits: requests with an id different than the expected one are discarded. It's used to prevent that a rigenerated thread (see section 2.3) sends a Commit request after that the file has been unlocked because of timeout (see later).

When a file is locked, the Thread waits for the Commit request: when it is received, and the id is the expected one, the file is unlocked. This mechanism ensures that each file can be accessed in an exclusive way by the node that has sent the request. Simultaneous accesses are allowed only for files accessed in read mode.
This approach is called Two-phase commit protocol [4], but with a small variance: to alleviate deadlock situations, due to some error in the Coordinator node, each file maintains the lock state for a default time of 60 seconds and updated based on the time needed to download it; after that the file is unlocked. The download time is calculated using a background Thread that compute the instant throughput of the transfer.

2.1.3 Anti-Entropy Mechanism

The anti-entropy mechanism is a background task used to resolve possible inconsistencies between files owned by different nodes. It's performed by means of a **Merkle tree** structure, which consists in a tree of hashes (tipically MD5) in which the leaves are hashes of a set of files (in this implementation the name of the file). Every level is the hash function applied to the concatenation of their respective children.
The **AntiEntropySenderThread** and **AntiEntropyReceiverThread** are the two classes used to perform the anti-entropy protocol. It's articolated in 4 phases, executed periodically by each virtual node:

1. **Node choosing**: the sender node (called A) tries a connection with a random node in the DHT (called B), to retrieve the keys managed by the requestor node. The destination node must belong to a different phisical server respect to the sender node;

2. **Handshake**: A sends to B the range of managed keys, that is the current virtual node identifier and its predecessor, and the height of the own tree. The virtual node id is used by B to check if A is in the synchronization state: a node enters in synchronization state when the anti-entropy protocol is completed (namely the 4 phases) and exits when the receiver node sent the up-to-date files. It is used to avoid that a node starts a new synchronization with the same destination node before have finished the previous one. If A is in that state the synchronization is refused, otherwise B sends to A the height of its tree. The tree's height is fundamental to reduce the cost of the operations because, in case the heights are different, the highest tree is reduced at the same level of the lower, since the hash of nodes belonging to different levels are obviously different and some exchanges are saved;

3. **Merkle tree exchange**: A sends the current level (typically starting from the root) to B. The level owned by B is then compared with the received level and a bit of 1 is set to every two equal nodes, using a specific BitSet object. B sends this set as a response to A.
   In addition, A and B, set, in a local BitSet object, a bit of 1 to all the leaves reached by those equal nodes. It will be used later to compare the common files.
   If the two levels are different the children of the divergent nodes are sent, repeating this phase until either two equal levels are not found or the leaves are reached;

4. **Versions exchange**: in case of common files (at least one bit sets to 1 on the **local** set) A sends a list of vector clocks, one for each common file, to verify if the version of the file owned by the sender is the most updated one. All the concurrent versions of a file are not resolved here, but only in the client during a *get* operation.
   So B sends all the up-to-date files, in addition to all the files not in common.

Assuming n_A the number of leaves of the A's tree and n_B the number of leaves of the B's tree, the cost of the operation may vary according to the level of the trees: if it is the same, the cost is $\mathbf{\Theta}(log(n_A))$ because only the different branches are taken, otherwise it's the worst case, namely $\mathbf{\Omega}(min(n_A^2, n_B^2))$.

Here the minimum is taken because we consider, as the number of leaves, the lower between the two trees. We can improve it to $O(min(n_A, n_B))$ exploiting the natural ordering of the keys (given by the database), remembering the position of the last two equal nodes during the scanning of the received level.

2.1.4 NetworkMonitorThread

Periodically (every 5 seconds) a StorageNode sends the own workload state to all the load balancer nodes of the system, using the UDP broadcast address. The workload of a node is computed using the SystemLoadAverage (method present in the **OperatingSystemMXBean** class) and the actual number of opened connections. These informations are then encrypted and sent via socket.

2.1.5 MembershipManagerThread

To reduce the network latency the client is able to talk directly with the storage nodes, avoiding the extra network hop that is incurred if the request were assigned to a load balancer node. To do that he must knows how many nodes are present in the system, in order to properly calculate the owner of a key. For this reason a background Thread is launched: it's purpose is to receives incoming requests and returns the list of nodes, discovered with the Gossip protocol. The client can now insert the nodes in a proper ConsistentHashing structure.

2.2 LoadBalancer

The aim of the LoadBalancer (Figure 2.4) is to balance the workload of each StorageNode. The node's workload is sent periodically by each storage node using the UDP broadcast address. The LoadBalancer use this information when it decides where to forward an incoming request.
When a LoadBalancer receives a request from a client, it uses the **ConsistentHashing** structure to decide which is the corresponding node (hence its successor). After that, it gets the preference list associated to the node (for simplicity the first N nodes encountered while walking the DHT), and, based on the nodes workload, it tries a connection with the most balanced one. If the connection is established the incoming one is closed, otherwise the hinted handoff address is set (only the first time) and the connection is tried with the next healthy node of the preference list. If there are no healthy nodes a negative response is returned to the client and the operation ends.

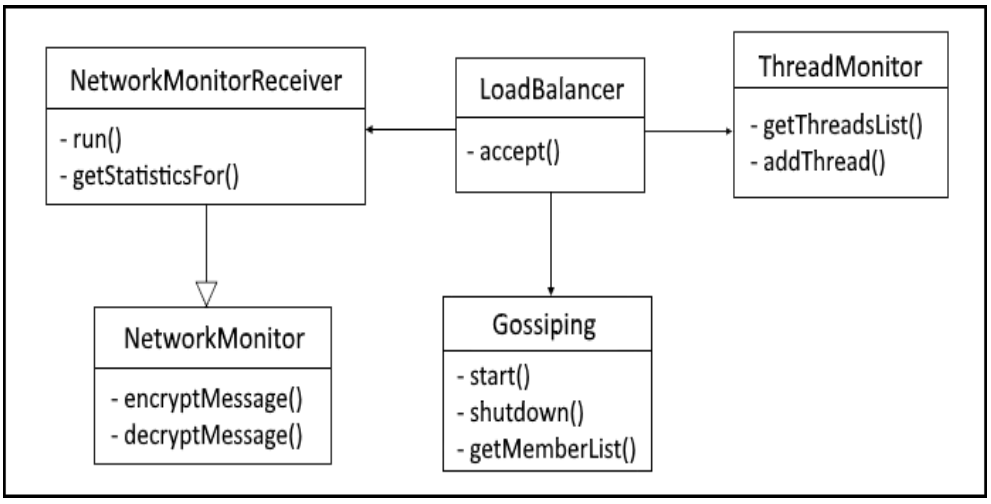

Figure 2.4: LoadBalancer structure.

A LoadBalancer node has an associated **NetworkMonitorThread** used to gather the workload informations about the storage nodes present in the system. The workload is calculated by averaging the values contained in the received data.

Unlike the StorageNode the only ports needed are the ones used to listen the incoming requests and for gossiping.

2.3 Fault Tolerance Mechanisms

Formally a fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components [5].
To prevent such situations the load balancer and storage nodes make use of a **ThreadMonitor** which checks, periodically, the state of the launched threads: in case one of them proves terminated prematurely (if the *completed* flag is false) a new Thread is launched to replace the dead one. The new thread, to recover from where the previous one died, retrieves and uses a list of actions done by the dead thread: if the list contains a reference to a certain operation, then it's skipped, otherwise it's executed.

2.4 Client

The Client entity (Figure 2.5) can be considered as an interface to the system. In addition to the operations provided by the service, the user has available

the following list of commands:

- **list**: print on screen a list of all the files present in the client's database;
- **enableLB**: enable the utilization of the remote load balancer nodes;
- **disableLB**: disable the utilization of the remote load balancer nodes;
- **help**: to show the helper;
- **exit**: to close the service.

The [TAB] key provides a list of suggestions and performs the auto completion of the commands, in order to execute them faster.

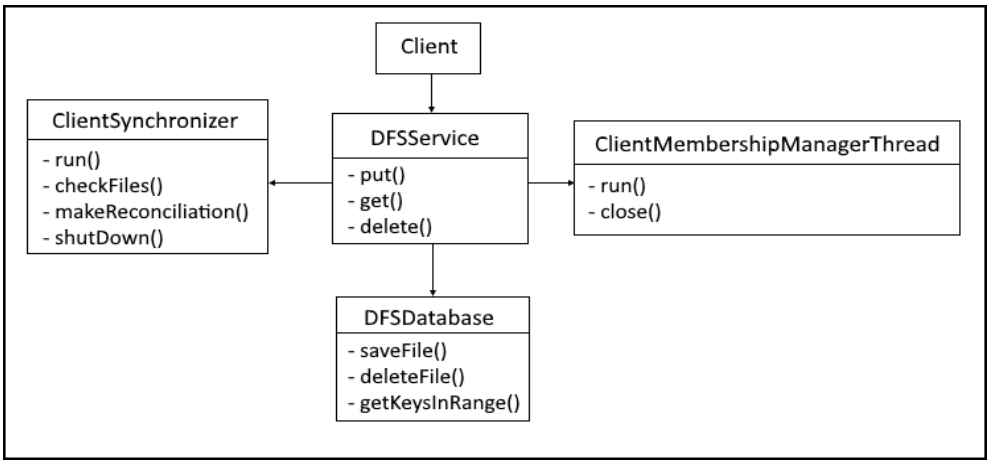

Figure 2.5: LoadBalancer structure.

When the client performs a **get** operation all the conflict versions of the file are retrieved. A first reconciliation is perfomed in the StorageNode to reduce the number of files to return. It's based only on the vector clock of the files, keeping all the concurrent versions. If at least two of them are returned all their vector clocks are merged and compared with the own one: in case of concurrent clocks then it asks the user which is the version he/she wants to keep: the selected version is then sent back via a **put/delete** operation in order to "notify" the storage node of the choice made.

By default the service makes use of load balancers, just to make the system more balanced. Through the **disableLB** command it is possibile to disable this mechanism, querying directly the storage node. This mode is maintained until the client owns at least one of them, otherwise the system automatically goes back to using again load balancer nodes. The client, by

means of the **ClientMembershipManagerThread** Thread, periodically picks a random storage node and downloads its current view of membership state. Using this information it can determine which set of nodes form the preference list for any given key.

2.4.1 Client Synchronizer

The Client entity has associated a **ClientSynchronizer** Thread whose goal is to keep updated the client's database. To perform it, the Thread periodically (every 500 milliseconds) scans the file system looking for any new brand spanking files. All these files are then sent in background using a *put* operation.
Every 30 seconds the Thread performs a *getAll* operation to retrieve all the files stored in a random node. This is useful in case of multiple clients present in the system. All the retrieved files are then reconciled with the own ones and a write-back operation is executed in order to notify the nodes.

Chapter 3

Project building

All the project can be built using either the **gradle** or **maven** tool.

3.1 Maven

The whole project can be automatically built using the script inside the **maven scripts** folder, just tiping:

sh buildAll.sh (for Linux and MacOS systems)
start buildAll.bat (for Windows systems)

Alternatively you can manually built the project using the following commands:

mvn install (server and gossip modules)
mvn package (client module)

3.2 Gradle

In Windows environment you have to set the path to the jdk. It can be done in 2 ways: setting the *JAVA_HOME* environment variable, or using the **gradle.properties** file. The latter method can be avoided if you want that your build is not dependent on some concrete path; in that case you can add this statement at the end of the subsequent commands:

-Dorg.gradle.java.home=/JDK_PATH

In addition the **./** is not needed when using gradle commands.

The Client can be built using the command:

./gradlew client:build

which recursively build the Server and Gossip modules.
The LoadBalancer and StorageNode nodes can be built using the following command:

./gradlew server:build

which recursively build the Gossip modules.
The Gossip module can be built using the command:

./gradlew gossip:build

3.3 Execution of the Code

The LoadBalancer and StorageNode nodes can be run using:

java -jar NodeLauncher-<version>.jar [parameters]

The only mandatory parameter is:

- **-t [--type] <type>** Start a LoadBalancer (type = 0) or a StorageNode (type = 1) node;

while the list of optional parameters is the following:

- **-f [--file] <value>** file used to configure the initial parameters. If present this must be the only specified option;
- **-p [--port] <value>** Port used to start the gossiping protocol. The next 5 ports are then used;
- **-a [--addr] <value>** Set the ip address of the node;
- **-n [--node] <arg>** Add a new node, where arg is in the format *hostname:port:nodeType*;
- **-v [--vnodes] <value>** Set the number of virtual nodes. This option is valid only for StorageNode nodes;

- **-r** [**--rloc**] <**path**> Set the location of the resources. This option is valid only for StorageNode nodes;
- **-d** [**--dloc**] <**path**> Set the location of the database. This option is valid only for StorageNode nodes;
- **-h** [**--help**] Show the help informations.

where the value inside [ ] is the long name for that option.
The Client can be runs using:

java -jar Client-<version>.jar [parameters]

All the parameters are optional:

- **-f** [**--file**] <**value**> file used to configure the initial parameters. If present this must be the only specified option;
- **-p** [**--port**] <**value**> Port used to receive the remote connection;
- **-a** [**--addr**] <**value**> Set the ip address of the node;
- **-n** [**--node**] <**arg**> Add a new node, where arg is in the format *hostname:port:nodeType*;
- **-r** [**--rloc**] <**path**> Set the location of the resources;
- **-d** [**--dloc**] <**path**> Set the location of the database;
- **-locale** Start the system in the local environment;
- **-h** [**--help**] Show the help informations.

where, again, the value inside [ ] is the long name for that option.

The setting files, when provided, can be putted in the **Settings** folder either inside or in the same folder of the jar.

3.4 Local Environment

When the Client is invoked with the **-locale** option, the system will be started in the local machine. If the list of input nodes is empty a fixed number of distributed nodes is used, namely 5 **LoadBalancer**s and 2 **StorageNode**s, executed in different Threads. For simplicity the number of virtual nodes for each storage node is 1.

3.5 Pseudo-Distributed Environment

Inside the distribution is present a **Vagrantfile** file used to create a user-defined number of virtual machines. In this way each VM can launch a different entity of the system simulating, in the local machine, the distributed environment.
If you want to run the file you need to have **VirtualBox** installed. Version 4.3 for Linux distributions is suggested because it's compatible with the provided Vagrant file (which is version 1.6.5). Unfortunately the VMs don't provide Java8, then you have to install it manually before to start the execution.

Chapter 4

External Resources

The Java version used is the 8th, then you need a JDK version 1.8 or higher. The following is the list of external resources used by the program:

- **junit**: used to test the program;
- **log4j**: used for the logging of the program (on output and on file);
- **mapdb**: used for in memory and on disk data storage;
- **commons-cli**: used to parse the command options;
- **commons-collections**: contains lot of efficient data structures;
- **jline**: used to read the user input;
- **guava**: used to compute the id of the objects using an hash function;
- **json**: used to read and write JSON files;
- **gossiping**: customized version of the gossiping protocol;
- **versioning**: cloned version of the Voldemort project.

Bibliography

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

[2] https://en.wikipedia.org/wiki/Triangular_routing.

[3] https://en.wikipedia.org/wiki/Readers–writers_problem.

[4] https://en.wikipedia.org/wiki/Two-phase_commit_protocol

[5] https://en.wikipedia.org/wiki/Fault_tolerance