UNIVERSITY OF PISA

PAD-FS

Leonardo Gazzarri

February 16, 2017

Contents

1 Introduction

The *PAD-FS* is a simple distributed key-value data storage system, it is written using *Java 8* and uses *Maven* as build tool. It is *flat* (no hierarchies of directories), user authentication is not present and provides to the user a *client program* implementing a CLI interface for the following essential operations:

- **get**(key), returns to the user the value(s) associated to this key
- **put**(key,value), put in the system the association key-value
- **remove**(key), removes from the system the association key-value
- **list**(), returns to the user the set of keys for which exists a value in the system

2 Architecture

The fundamental entities that build the system are the *client* and the *PAD-FS node*, figure 2.1 shows an high level representation of the system.

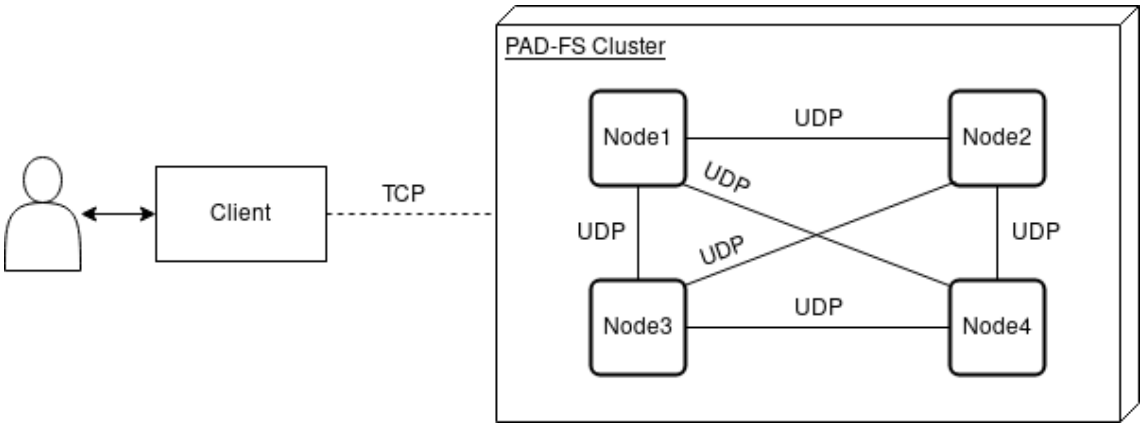

Figure 2.1: *High-level representation of the architecture with four nodes*

The *client* represents the front-end of the system and permits to the user to read/write items from/to the nodes. The client-node communications use TCP at the transport layer since it is a more reliable protocol than UDP for the Internet.

The *node* is responsible for managing the client requests and storing, versioning and replicating the associated data. The node-node communications use UDP at the transport layer in order to avoid the overhead of TCP and it is assumed a reliable network inside the cluster. A *node* is like a service container, inside it there are several running services, notably a *server service* handling the client connections, a *storage service* handling the updates and managing the conflicts, and a *gossip service* that discovers new nodes in a cluster and detects the dead ones.

3 Design choices

PAD-FS is designed to be a distributed and *fault-tolerant* data storage, therefore node and network failures are expected and the system have to deal with them. This section will try to describe the main design choices that lie behind our faulty distributed world.

3.1 Membership and Failure Detection

For the membership and failure detection of the nodes is used a simple dynamic gossip protocol implemented as an anti-entropy protocol, *i.e.* each node periodically chooses another node at random and exchanges contents (its view of the neighborhood) and resolving differences. Moreover, each exchanged message contains a *timestamp* information in order to signal its age, when a node doesn't hear updates from a partner within some interval (*cleanup interval*) the partner node has to be considered dead.

3.2 Processing of Requests and Replication

The replication strategy adopted by *PAD-FS* is a variant of the *passive* one

- each client can communicate with every node in order to perform an operation (get, put or remove) for some key k but only one node (the primary server) can process it
- if the receiving node is not the primary server for a client request on key k, it forwards the request to the considered primary server and waits a response acting like a *proxy*

A node individuates a primary server (or master, or coordinator) for a given operation with key k using *consistent hashing* with virtual nodes, using the membership view computed by the gossip protocol in order to build the set of buckets and inserting the key k. Thanks to the *consistent hashing*, when a primary server fails the election of another node as master for some key is immediate and it is performed without involving other nodes. Figure 3.1 shows the actions taken by the client, the receiving node *Node1* and the primary server *Node4* for a generic client request with key k.

As said before the intermediate node, waiting a response for a forwarded client request, acts as a proxy between the client and the primary node. The node receiving a client request with a key for which is the master, processes the request, eventually updates the local store (with versioning information) and sends back the response. In the case of a *remove operation* the primary server multicasts the remove message to all (*scalability problems could occur*) its known members (without waiting a response for each of them); the remove is handled like a write in the local store for managing cases in which a node is temporarily down or doesn't receive the message. In these cases the local store will be eventually updated by the background replication protocol.

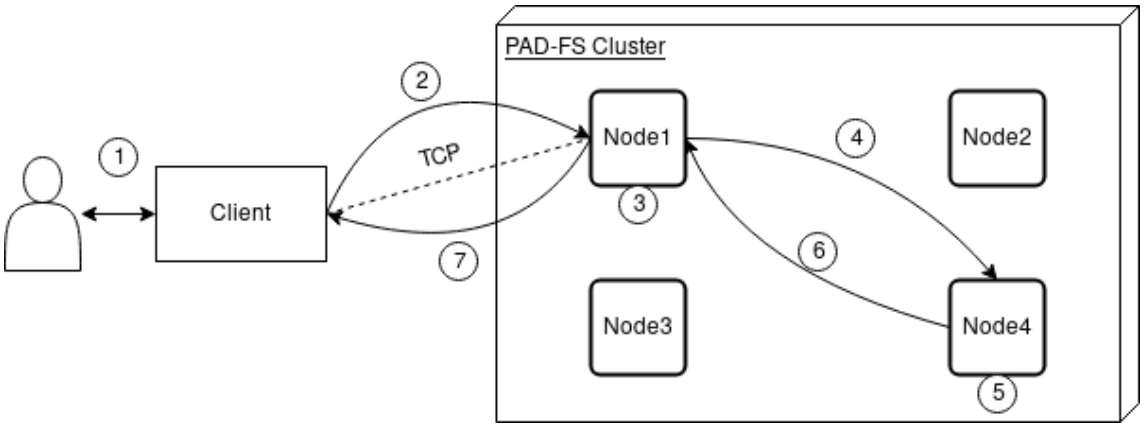

Figure 3.1: *High-level representation of how it is handled a request in* PAD-FS. *(1) user made a request, (2) client application builds a proper message and sends it to a node, (3)* Node1 *checks out who is the master for the key present in the message, (4)* Node1 *forwards to the primary node* Node4 *the message with a version, (5)* Node4 *processes the message updating its local store (in case of a remove it multicasts also to its neighbours the remove message), (6) the response is sent back to* Node1*, (7)* Node1 *sends back to the client the same response*

3.2.1 Replication Protocol

A primary node for some keys updates the other nodes in background using a sort of *push-pull* anti-entropy protocol. A node p periodically sends to a *randomly-chosen* alive neighbor n all the keys present in its local store in one or more *push* messages (in a single push message are carried multiple keys). For each received key, node n sends back to p

- a *pull* message asking for the value associated to the key if it is not present or not updated in p's local store
- a *reply* message with n's local versioned value if its associated version is more recent than the one associated to the key in the message

When a node p receives a *pull* message from n for some keys it sends back to n one or more *reply* messages carrying the associated versioned values for these keys.

When a node p receives a *reply* message for some keys it updates the proper entries in the local store if the carried value's versions are more recent of the local ones (or if the values are concurrent, but in this case both the values are kept).

At the end all the <key,value> pairs will be replicated in all the nodes of the system after some time, in other words the system *converges* after some time. The choice to send periodically **all** the keys to a random node is good in terms of a faster convergence of the system but it originates more internal traffic (more the keys, more the periodic push messages) and for a system with many nodes and big local stores this should be a negative overkill in terms of scalability and performances.

3.3 Conflict resolution

Conflicts between replicas are inevitable in a distributed system like *PAD-FS* and therefore it needs some form of conflict-resolution mechanism. Versioning is made by the use of *vector clocks* that facilitate data consistency and conflict-resolution since they are used in order to detect conflicts, meaning updates whose order cannot be determined. When a value from a client request is put by its master in the store it is also versioned with the local vector clock of the master. The chosen approach to handle conflicts for different values associated to a key is to save all the conflicting versions. When the client asks for a key that has multiple conflicting values the system returns all of these values.

4 Implementation

This section will describe some implementation details about the system referring to name classes listed in section 5. The basic idea of the implementation is to divide the system in "levels" (services), the *server level* handles the connections with the client, creating a short-living thread for each client request, the *storage level* handles in background the replication of items between nodes and the internal routing of the client messages in order to reach the primary nodes, the *membership level* handles in background membership and failure detection plus partitioning (primary server individuation service). Figure 4.1 shows the associations between some of the principal classes of the project.

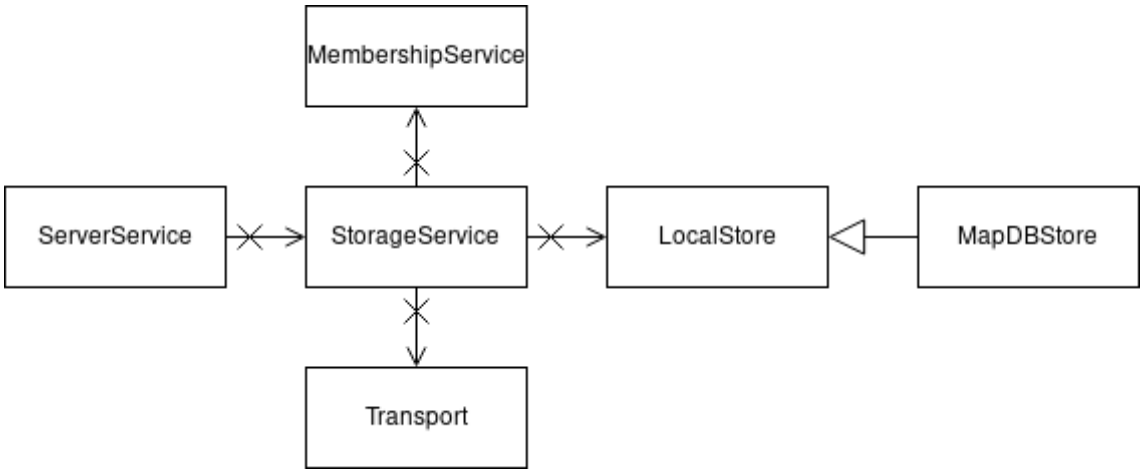

Figure 4.1: *UML Class Diagram of some of the principal classes of the project*

4.1 Messages

Messages are Java objects implementing `Serializable` interface and they are encapsulated into UDP packets in node-node communications or into TCP packets in client-node communications. Fields that are in common for each kind of message are the following:

- **status** field that identifies the response status (SUCCESS, NOT_FOUND, ERROR,...)

- **type** field that identifies how to process the message

The **type** field influences the successive fields of the message that are different for messages of different types.

4.2 Execution of Client Operations

When at the server layer a node receives a client message, the local running `ServerManager`'s instance spawns a new instance of `ServerThread` handling the request by invoking the method `deliverMessage` of the local `StorageService`. The method returns immediately a response for a LIST-type message (a list of keys owned by the node), otherwise it eventually forwards the received message to the primary node and then awaits for a response.

The primary node for a key is determined using *consistent hashing* (see `getCoordinator` method in `MembershipService`). The thread designated to receive and handle messages at the storage layer is a running instance of `StorageManager`. When the primary node running instance of `StorageManager` receives the client message it spawns a `ClientHandler` instance handling the request, processing it and sending back a response. When the node receives a response by the primary node it returns the response to the server layer that in turn will send back it to the client via the open TCP connection.

4.3 Storage Protocol

The replica manager (running instance of `ReplicaManager`) periodically picks a node from the members list and sends to it all the keys present in the local store (see the runnable `UpdateHandler`). The keys are sent with one or more `PushMessages` at the storage layer, for each key there is a list of vector clocks that indicate the times (more conflicting versions can be present) in which the key has been put/removed in/from the system. The storage manager (running instance of `StorageManager`), as said before, is the thread designated to receive all the messages at the storage layer and when it receives a message, it checks its type and calls the proper handler.

- `PushMessage` triggers the execution of a `PushHandler`, this handler for each received message's key compares the associated vector clock with the local vector clock stored for the same key in the local store. For the *happening after* message's keys, the handler builds one or more `PullMessages` in order to get their effective values, for the *happening before* message's keys the handler builds one or more `ReplyMessages` in order to acknowledge the push's originating node that exist newer versions for some keys.
- `PullMessage` triggers the execution of a `PullHandler`, this sends back to the originating node the keys with lists of versioned values through `ReplyMessages`.
- `ReplyMessage` triggers the execution of a `ReplyHandler`, this updates the local store with the message's value if its version is newer than the local existing one. If the version is concurrent the message's value is appended with the existing ones.

- `PutMessage`, `GetMessage` and `RemoveMessage` trigger the execution of a `ClientHandler` that processes the message in function of the type, in particular it calls the method `deliverMessage` of the local storage service.

4.4 Local Store

Different implementations of the local store are present, the default one is the implementation in `ActiveMapDBStore`. **MapDB** library is used in order to provide a persistent store and the proposed implementation avoids to perform a commit (persist change on disk that is very costly) for each put/remove by keeping a thread `putHandler` that continuously reads from a concurrent queue the *put requests* coming from other threads (the client/reply handlers). Each of these requests is processed and stored into the MapDB's HTreeMap. The `putHandler` commits the changes into disk every *commitPutThresh* put requests that have been processed and saved or, if the last processed put is older than *commitTimeThresh* milliseconds.

5 Project Structure

PAD-FS is structured in two sub-projects, **core** and **cli** that respectively implements the *node* and the *client*.

5.1 core

It contains the code needed to implement the *PAD-FS node*, the structure of the system, the communications and it defines the message types.

- **common**, contains utility classes and the interface `IService.java` that defines the fundamental methods `start`, `shutdown`, `isRunning`
- **membership**, contains the interfaces and the classes that implements the membership and failure detection
  - `Member.java`, defines a member in a cluster
  - `MembershipService.java`, implements a service that provides methods to get the alive members of the cluster and to get a coordinator (the primary server) for a key
- **message**, contains the classes that implement different message types (*e.g.* `PushMessage`, `PullMessage`, `GetMessage`, `PutMessage`)
- **server**, contains the interfaces and the classes that implement the server level of the system
  - `ServerManager.java`, implements the server as a thread that listens for new connections and for each connection it spawns an instance of the `ServerThread` class

  - `ServerThread.java`, implements an handler for a client's connection and it uses methods offered by the `StorageService` class in order to provide the a response

- **storage**, contains the interfaces and the classes that implement the storage level of the system
  - **local**, contains the interfaces and the classes implementing the local store, meaning the local database in which <key,value> pairs are stored
  - **runnables**, contains the handlers by the `StorageManager` for each type of message
  - `StorageService.java`, implements the method `deliverMessage` offered to the server level in order to provide a service and starts the instances of `StorageManager` and `ReplicaManager`
  - `StorageManager.java`, implements a thread that receive routed client messages or the ones used for the replication protocol and for each type of message call the proper handler
  - `ReplicaManager.java`, implements a thread that periodically sends *push* messages to a randomly chosen neighbor node

- **transport**, contains classes that implement the transport level of the system
  - `Transport.java`, implements send and receive methods and update properly local and message's vector clocks

- `Node.java`, implements the node service of the *PAD-FS* system as a *container* of different services (`MembershipService`, `ServerService`, `StorageService`)

5.2 CLI

It contains the code needed to implement the *client* of *PAD-FS* system. The principal implemented classes are:

- `Client.java`, implements the client request and the communication with a server node

- `ClientRunnable.java`, parses the command line and builds a proper client request

6 USER GUIDE

In order to install and use *PAD-FS* the requirement is to have installed *Java 8* on the running machine. The system can be used either in a multi-threaded version in the same machine (pay attention on binding different local addresses to the threads) or in a cluster.

6.1 INSTALLATION

In order to generate the jar executable files for the client and the servers the user can run the following commands:

- `git clone https://github.com/Murray1991/PAD-FS.git`
- `cd PAD-FS`
- `mvn clean package`

After the installation steps all the jar executable files will be in the **./target** directory.

6.2 RUN THE SERVER

The syntax to run the server node is the following:

- `java -jar target/pad-fs.jar [-b <binding-addr>] [-c <config-file>] [-p <path for the local DB>]`

If the binding address is not provided the program will use the address associated to the hostname of the machine (check out your **/etc/hosts**). If the configuration file is not provided the program will check **./pad_fs.config** file if present. If the path for the DB is not provided the program will use/create the DB in current directory

6.3 RUN THE CLIENT

The syntax to run the client is the following:

- `java -jar target/pad-fs-cli.jar -p <key> [<value>] | -g <key> | -r <key> | -l  [-d <dest-addr> | -c <config-file>] [-o <output>]`

The options `-p`, `-g`, `-r` and `-l` respectively refer to the basic operation put, get, remove and list. The option `-o` specifies the output file in which to store a get response, if it's not provided the response's value will be printed in the standard output. If the destination address or the configuration file is not provided, the program will check **./nodes.conf**, a file in which are stored the IP addresses of the PAD-FS nodes.

6.4 CONFIGURATION FILES

In order to run both the client both the server node, configuration files are needed (for the client is optional but recommended, for the server is mandatory). Examples of the configuration files for the client and a server are respectively **nodes.conf** for the client and **pad_fs.conf** for the server, both present in the root directory of the project.

7 TESTS

The tests are located under ***core/src/test*** directory and covers the main service classes used in *PAD-FS*.

7.1 SERVER SERVICE TESTS

The server layer of the system is tested in `ServerServiceTest.java`, this file provides two test methods that setup a server service thread (instance of `ServerService`) and check if it is able to handle client requests. The two test methods differ in the way these requests are generated, in `sequentialTest` are generated sequentially while in `threadTest` are concurrent. The server layer in both tests processes the client requests passing them to a dummy service (`DummyService`) implementing the `IStorageService`. The dummy service simply puts a delay in the implemented `deliverMessage` method.

7.2 MEMBERSHIP SERVICE TESTS

The membership and failure detection service is tested in `MembershipServiceTest.java`, this file provides several test methods in order to check if the gossip protocol has been configured correctly. All the test methods setup a multi-threaded environment of four `MembershipService` running instances.

- The method `gossipTest` checks after some delay time if each membership service knows exactly three neighbors.
- The method `gossipTestWithFailures` tests if multiple failures are detected by the alive membership service by turning off two of them, moreover it tests if they are added back in the membership list if they are restarted.
- The method `coordinatorTest` tests the `MembershipService`'s method `getCoordinator` in different situations.

7.3 STORAGE SERVICE TESTS

The storage protocol is tested in `StorageServiceTest.java`, this file provides several test methods in order to check if client messages are correctly handled by the node, if the anti-entropy protocol for replication works properly, if the membership and failure detection service is correctly integrated and how the system behave in the presence of updates or removes for some keys. All the tests setup a multi-threaded environment of four `StorageService` running instances. Failures are simulated by turning off the membership service of a node.

- The method `testWithoutUpdates` puts some <key,value> pairs at random nodes and check after some delay time if all the pairs are present replicated in each node. The method

- The method `testWithUpdates` recall `testWithoutUpdates` and updates some of the existing keys. Then wait some time and check if the updates have been correctly replicated in all the nodes.
- The method `testWithFailuresAndUpdates` basically turns off the membership service of a node and makes updates on the existing keys. Then the membership service is restarted and it checks after some delay time if the updates have been correctly replicated.
- The method `concurrencyAndRemoveTest` turns off for each storage service its membership service, then after some delay time it delivers messages with same keys but different values to two different nodes. Then it restarts the membership services of all nodes and after some delay time it checks (sending get messages and checking if the responses are the expected ones) if both the conflicting versions for a same key are correctly stored in each node. Then the test continues removing some keys from the system (sending remove messages to a node) and checks by sending get messages for the same keys if their responses have status equal to `Message.NOT_FOUND`.
- The method `removeTest` tests the situation in which a key is updated in all the nodes, the primary node for the key goes down and the key is removed.

8 Issues

This section summarizes in a very informal way some of the known issues related to the project, some personal considerations about them (*that could be horribly wrong*) and, possibly, future work ideas.

- **Each node periodically picks another node and sends to it all the keys it owns via multiple push messages**. Although this approach leads to a faster convergence and a stronger replication it is also a possible source of performance degradations. Growing the nodes and the stored keys, the internal network should be overloaded by the periodically generated *push* messages, this could increase the generation of *pull* and *reply* messages and affect in a very negative way the client's message process time.
- **At the storage layer no segmentation of client messages is provided**. Section 2 shows that client-node communications use TCP as transport protocol, while UDP is used for the internal node-node communications. TCP provides several features, one in particular is the *segmentation* of big messages in multiple segments, so there is no need to bound an application-level message to some bounded size. In *PAD-FS* when the client message comes to a node it is typically routed to the master via UDP and **no check size** is performed. This should throw a `IOException` if the client message is too big and return a generic `Message.ERROR` message to the client (not tested behavior).
- **The "proxy" node can fail or return a Message.ERROR to the client even if the request has been correctly processed**. Referring to figure 3.1, *Node1* can fail when

Node4 is processing the request or the response sent back to *Node1* by *Node4* could be lost (maybe dropped by an internal switch) or not arrive in time. In the latter case the response sent back to the client is a generic `Message.ERROR` (triggered by an associated timeout).