Pad-fs a distributed data store

Course

Distributed Enabling Platforms

Project Report

Davide Neri

University of Pisa and SSSUP Sant'Anna

- a.a. 2015-2016 -

Contents

Chapter 1

Introduction

Pad-fs (Piattaforme Abilitanti Distribuite - File System) is a distributed file system that store key value pairs. It is written in *Java* and uses *Apache Maven* for the project management. The git version can be found `https://github.com/dido18/PAD-FileSystem`.

A high level overview of the *Pad-fs* system architecture is shown in figure 1.1. It is composed by two parts: the *storage system* and the *client* node.

- The **storage system** is a set of communicating storage nodes that are responsible to manage the data to be stored. Each storage node has a local database and it is composed by two main services:
  - *GossipService* is a background thread that perform the gossiping protocol.
  - *StorageService* is the service that listen for incoming messages on storage port and performs the operation on the persistent storage (add values, remove values, update versions, resolve conflict, quorum system, etc...).
- The **client** is an external independent node that interact with the storage system in order to perform the file system operations. An user interacts with the client through a *cli*. The client, upon an user input operation, determines the master node,

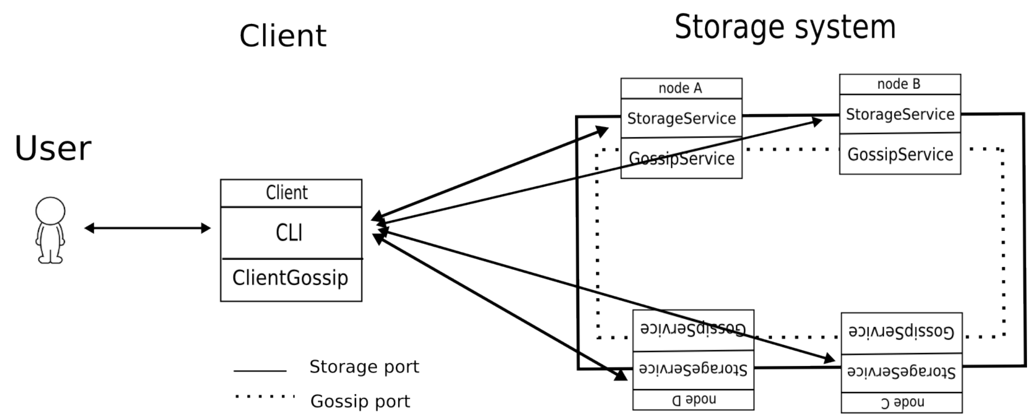

Figure 1.1: Pad-fs architecture overview

constructs the proper message and send it to the master node selected. In order to know the master node of a key, the client has the same consistent hashing logic of the storage system. Client exposes four API operations to the user: get, put, list, rm.

The client node has two main modules:

- *Cli* (command line interface) receives user's inputs and sends messages to the storage system.
- *Client Gossip* keeps updated the client's view of the storage system system. It asks periodically to a random node in the storage system the current active nodes.

- **User** is who submit the operation into the system through the cli.

Chapter 2

Pad-fs design choice

In this section are listed the main characteristics of *Pad-fs* and the protocols that has been chosen. *Paf-fs* exploits a weak consistency model, based on quorum protocol and primary-backup protocol where writes and read operation are forwarded to a single master server.

- **Partitioning**. In order to scale incrementally pad-fs use *consistent hashing* . The hash function can be defined by the programmer, by default *Pad-fs* uses *SHA1* hash function for hashing both the data and the nodes.
- **Replication**. In *Pad-fs* a data is replicated in multiple distinct nodes: into the master node and into `N_REPLICAS` backup nodes in the clockwise direction. The master node is responsable to manage the keys. `N_REPLICAS` is the variable that indicates the number of nodes after the master node that has to receive a copy of the data (by default `N_REPLICAS` is equal to two).
- **Quorum system**. It is used for handling temporary failures. `WRITE_NODES` are the number of backups nodes that must responds successfully at at put operation. `READ_NODES` are the number of backups nodes that must terminates successfully to a get operation. If not all the backups responds a error message is shown to the user and the operation is not performed (by default `WRITE_NODES=1`, and `READ_NODES` = 2)
- **Versioning**. Pad-fs uses vector clocks associated with the data in order to resolve inconsistency. The vector clock is a pair <id:n> where id is the node id and *n* is an integer number.
- **Resolve conflicts**. An important design choice is when resolve the conflicts. *Pad-fs* resolves the conflicts during read (GET) operation. The conflict resolution procedure is performed by the user and not by the storage service. When a conflict is detected, all the concurrent version are sent to the client that asks to the user to select the right version of the data. When the right version is chosen by the user, the version is updated in all the backups nodes
- **Gossip protocol**. is used for membership and failure/update detection of the nodes. Pad-fs admits only the case when a already present node goes down and than return up, it doesn't admit a totally new node join the storage system.

Chapter 3

Implementation

3.1 Execution of get, put, list operations

In the first section are presented in detail the implementations of the API exposed to the client, in particular are listed the steps needed to perform the operations. In the second part is explained the structure of the messages exchanged onto the system.

The APIs exposed to the user are:

- *put(key, value)*: inserts/updates the value associated with the key.
- *get(key)*: retrieves the value associated with the key.
- *list(ipNode)*: shows the key values pairs stored in the node with IP=ipNode.
- *rm(key)*: removes the key and the associated value in all the node where key is stored.

Get operation

The get(k) retrieve the value associated with the key k. The figure 3.1 shows the steps of a successfully get operation.

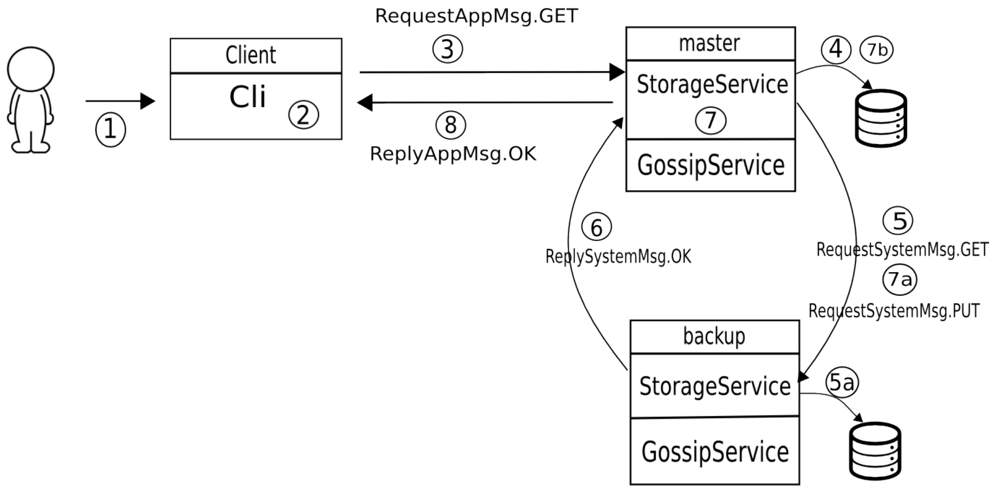

Figure 3.1: Steps of a successfully get operation: (1) get(k). (2) selects master node for k. (3) send get message to the master node. (4) get(k) from db. (5) send get msg to READ_NODES. (5a) get(k) from db. (6) send data to master, (7) compare (master vs backups) version (7a) if master is after than send master version to backups (7b) if master is before than store backup. (8) send reconcilied version to client.

If get operation found conflicts among data, the conflict is resolved by the user. The master node send a *ConflictMsg* to the client that expose to the user the choices. The figure 3.2 are shown the steps of a get operation with conflict.

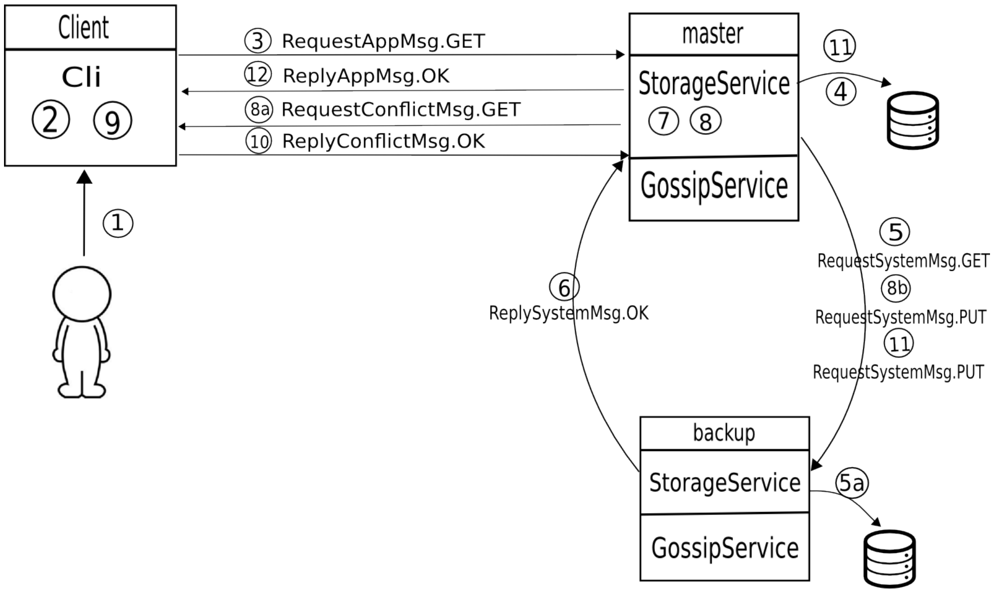

Figure 3.2: Get with conflicts:(1)get(k).(2)select master node for k (3) send get msg to master. (4)get(k) from local db. (5) ask version to Read nodes. (5a) get(k) from db. (6) send data/version to master. (7) wait at least read nodes response. (8) if concurrent versions. (8a) send msg to client with versions (9) user insert the selection. (10) send to master the selection (11) store into db and send to backups the selected version. (12) reply to client.

Put operation

The figure 3.3 shows the steps performed by a successful put operation. The client receives a put operation from the user, selects the master node of the key and sends the put message to the master node. The master store the new value for the key to the local database and send a copy to the backup node.

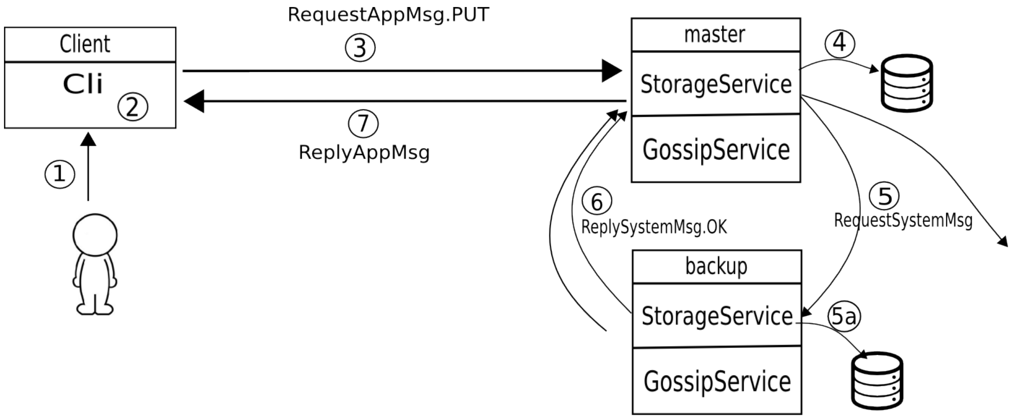

Figure 3.3: Steps of a successfully put operation: (1) put(k,v) from user. (2) select master node for k. (3)send put msg to master. (4) put(k,v) into local db. (5) send put msg to write nodes. (5a) put(k,v) into local db. (6) reply successful put (7) reply to client after write nodes has responded.

List operation

The figure 3.4 shows the steps for the list operation in the system.

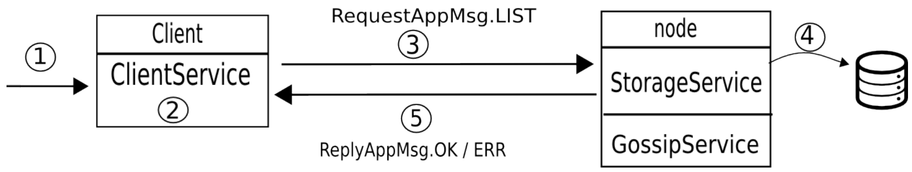

Figure 3.4: Steps of list operation

The list operation is used to retrieve all the key value pairs stored in a node. The client selects the master node for the key and send the list message. The node, upon the message received, gets all the key value pairs on its local database and reply to the client.

3.2 Messages format

The messages in the system are sent into UDP packets. The figure 3.5 shows *Pad-fs* message inside the payload of UDP packet. The main fields are:

- **ipSeder** (String) identify the ip address of the sender node of the message.

- **Port sender** (integer) identify the port of the sender node where the message has been sent.
- **Type** defines the two types of messages: *request* and *reply* messages.
- **OP** is the operation requested in the message. Can be one of the following: put, get, list, ok, err, dscv, rm (dscv messages are used by the client to discover nodes in the system).
- **data** (different type) contains the payload of the message. Can have different types, for example in a get operation the data is the String of the key.

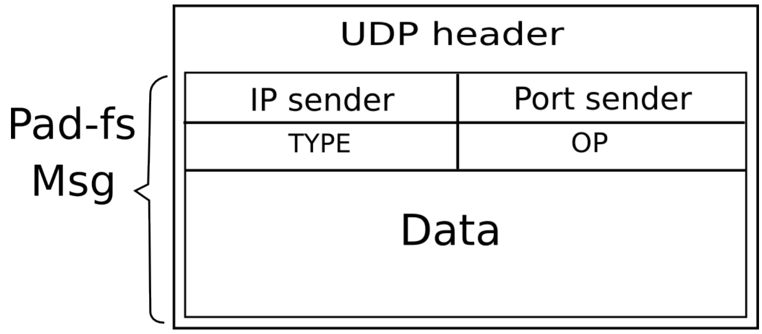

Figure 3.5: Pad-fs message

The messages can de divided in two subset:

- *Application messages*: are used only to send messages from the client to the storage nodes.
- *System messages* are used only by the storage nodes to exchange data version and control messages in the storage system.

Chapter 4

Project structure

Pad-fs is divides in three sub projects:

- `cli`: package containing the source codes for the client node
  - `Client.java`: is the code of the client node. It has the service that executes the client's gossip.
  - `Cli.java`: is the command line interface exposed to the user. It contains the same consistent hasher of the node in the storage system.
- `app`: package containing a runnable simple storage system composed by a set of nodes, that can be run on a single machine (useful for testing).
  - `AppRunner.java`: starts a set of the storage nodes onto the local machine. It provides also the possibility to remove/add an existing node in the system in order to simulate a leave/down event in the system.
- `core`: contains the source code of a single storage node.
  - `data`: contains the structure of the data stored onto the database.
    * `StorageData.java`: represents the key value that can be stored into the system. The key is a string and the value is a generic type.
    * `Versioned.java`: wraps the StorageData with a version.
  - `hashing`: contains the interface and the classes to define the consistent hasher logic used by the nodes and the client.
    * `Hasher.java`: is the consistent hashing code. It maps the nodes and the data into the same space. The hash function used for the hashing procedure can be any unless it implements the *IHashFunction*. The hashing of the node is done by the concatenation of the ip and the id.
  - `messages`: contains the structure of the messages exchanged in the system.
    * `AppMsg` is the top level class that define the type (request, reply) the operation (put, get, list, ok, err, dscv) the IP of the sender, and the listening port.

    * `ReplyXXX.java`: are reply messages.
    * `RequestYYY.java`: are request messages.
  - *versioning*: contains the version type.
    * *Version.java*: is the interface that a concrete version must be implemented.
    * *VectoClock.java*: is a concrete version that represents the vector clock. It provides all the method needed to a vector clock: compare different version, merge with another vector clock.

4.0.1 Tests and run the projects

The tests are located into the *core* package. They cover the most important class of the system. In the `core` package the test cover the three main important aspects:

- *Consistent Hashing*: test all the classes related to the hashing procedures.
  - *testHashing.java* test the insertion of nodes and check presents of the nodes.
  - *testMoreServer.java* test the insertion of data and the server for the data inserted.
  - *testRemoveAddServers.java* test adding or removing nodes that change data association with the nodes.
  - *testNextPreviousServer.java* test previous or next nodes retrieving.
- *Versioning*: tests the versions and the operations among different data (merge, compare).
  - *testVectors.java* test clone and merge operation on vector clocks. Tests also the comparison among vector clocks.
  - *testVersioned.java* test the data into the storage associated with a version. It update the versino and check the comparison among versioned data.
- *Quorum system*: tests the replication of the data and the quorum system running a simple storage system.
  - *testGetMerge.java* run a storage node of three nodes and test the get and merge operations and the data. It tests also the message exchange among nodes.

In order to run the **tests** the syntax is:

```
$ mvn test
```

For testing the execution of the operations can be used the *AppRunner.java*. It starts, on local machine, a set of storage nodes that can be used for testing some operations.

Run the projects

In order to run a **storage node** the syntax is:

```
$ java cp core-1.0-SNAPSHOT-jar-with-dependencies.jar \
 com.dido.pad.PadFsNode  [options] ipSeed:id[:gp]
```

The example below run a node with ip 127.0.0.2 and node2 id and sets the 127.0.0.1 as the seed node.

```
$ java cp core-1.0-SNAPSHOT-jar-with-dependencies.jar \
  com.dido.pad.PadFsNode -ip 127.0.0.2 -id node2 127.0.0.1:node1
```

In order to run a **client** with IP address 127.0.0.254 and one seed node with IP address 127.0.0.1, can be used the command:

```
$  java -cp target/cli-1.0-SNAPSHOT-jar-with-dependencies.jar
   com.dido.pad.cli.MainClient  -ip 127.0.0.254 -id client 127.0.0.1:node1
```

4.0.2 External libraries

I have used the sequent external libraries:

- `com.github.edwardcapriolo`: provides the gossiping protocol. I have used this library has a background service to the storage node in order to update the consistent hashing when a node in the network goes down or up. The version used in the project can be found here. I have used this version because in the google version there is a bug when a node goes up.
- `log4j`: is used to perform the logging procedure. Each log level is configurable for each class in the `log4j.properties` in the core project. Essentially there are two log level: debug and info.
- `com.fasterxml.jackson.core`: is used to parse the messages into a json format.
- `org.mapdb`: is used to implement the persistent storage in the node. It permits to store the key value data store into a file and performs useful operations.
- `junit`: is used to implement the unit tests.
- `com.beust.jcommander`: is used to parse the command line options of the programs.