UNIVERSITÀ DI PISA

Distributed Systems and Middleware Technologies

Distributed File System with Pastry

Iacopo Canetta

Francesco Berti

ANNO ACCADEMICO 2024/2025

Contents

1 Introduction

This project consists in the implementation of a peer-to-peer distributed file storage using a distributed hash table for locating the files among the nodes. The distributed hash table is based on Pastry protocol.

2 Pastry

Pastry implements a logical ring of nodes, and each node has an ID. The files are hashed to get a key, and the key is used to identify which node is responsible for that file, checking which one has the longest common prefix. When a file is accessed or stored, a request is sent to the current node. When a node receives a request, it uses a routing table, structured as a tree, for finding the next hop for the request. Every time a row of the routing table is accessed, b bits of the prefix of the key are used to identify the correct column. If the b bits of the prefix are owned by the current node, the next row is visited, otherwise the address of the next hop is retrieved.

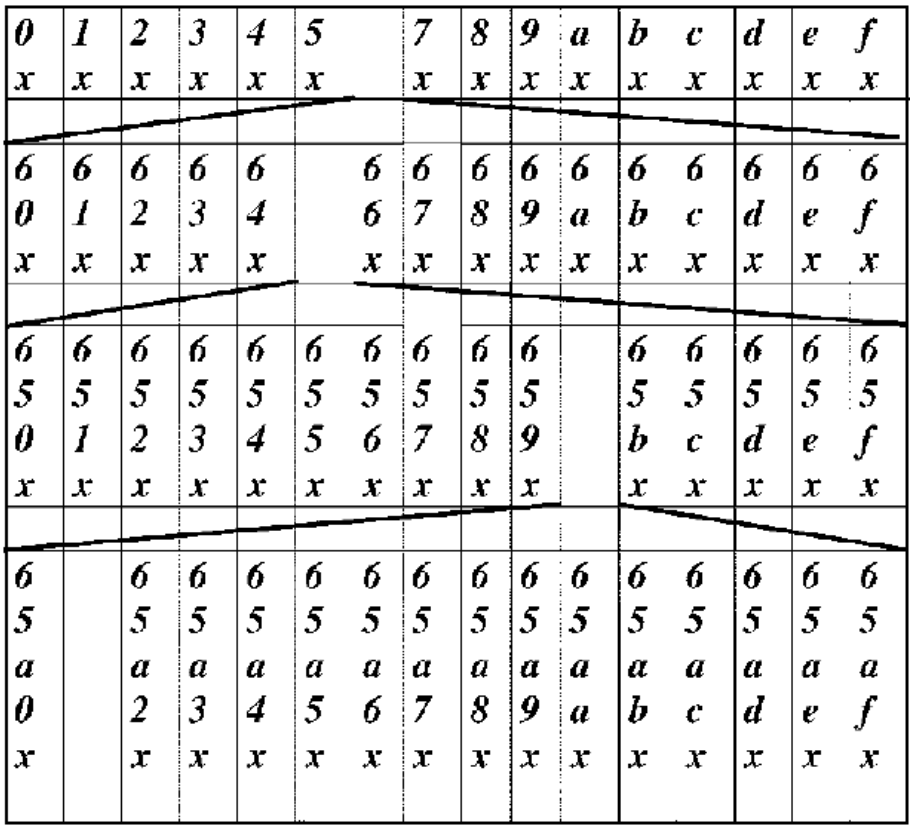

Figure 1: Routing table

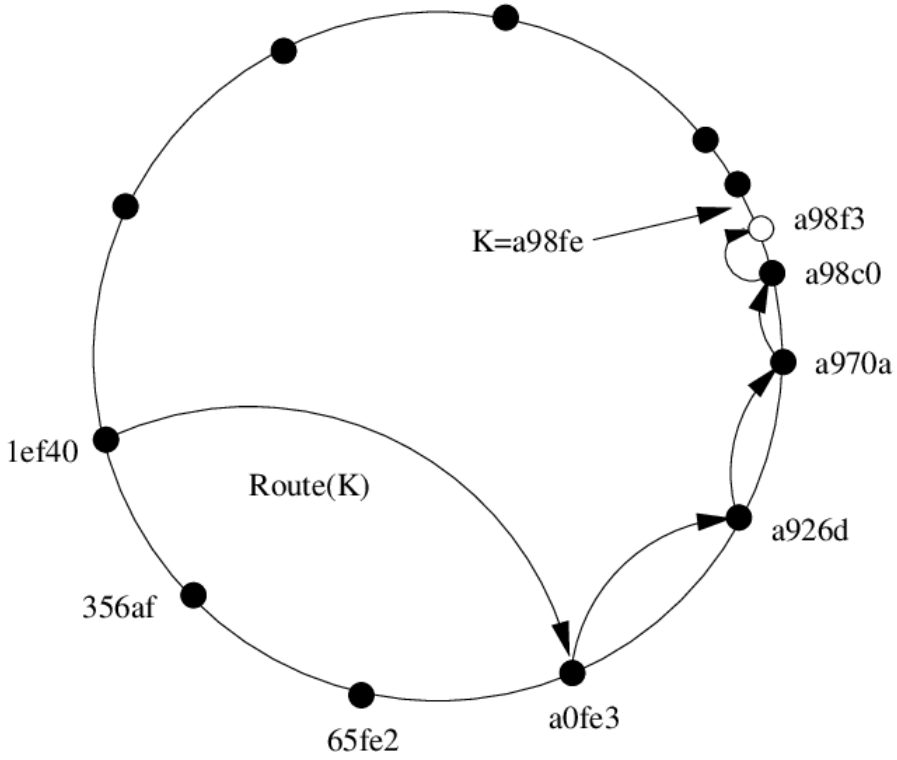

Figure 2: Routing example

Whenever an address is not found or the whole table is crossed, an extra table is checked, called leaf set and containing the nodes with id close to the current node; if no node in the leaf set has a longer shared prefix with the key than the current node, then the current node is responsible for the file. This routing table, using N nodes, guarantees a maximum number of hops equal to logn.

3 Implementation

The implementation of the system is divided in two parts: one for the distributed file system, created in Erlang, and a webserver realized in Java, also thanks to the Frameworks and implementions of the Java Servlets.

3.1 Erlang

The Erlang code is divided in multiple modules, in order to isolate functions and keep them short enough.

3.1.1 Util.erl, File_handler.erl, Network.erl

This files are created for having simple utility functions. *File_handler.erl* contains the utility functions for dealing with files, *Network.erl* contains simple functions for starting the Erlang nodes and settign the cookies, in order to receive messages from different machines, and *Util.erl* contains generic utility functions.

3.1.2 Key_gen.erl

Key_gen.erl is the file that implements the key system for Pastry. The keys are generated hashing strings and represented as *bitstrings*; this type is chosen because it can be used for representing streams of any number of bits, differently from the *binary* type, which is exclusively for multiples of 8 bits.

Bitstrings, however, have little documentation and almost no function dedicated, so *Key_gen.erl* also implements the functions for comparing these numbers and handling them as hexadecimals, since Pastry requires groups of 4 bits for the routing.

3.1.3 Routing.erl

Routing.erl represents the core of Pastry protocol, implementing the Routing Table. The routing table is represented as a Tuple containing the TableKey, so the key of the node owning the table, and the table. The table is a list of 32 lists, representing the rows; the single entries of the rows are represented by a tuple containing the hexadecimal identifying the column and the node information.

During the routing or any table modification, instead of doing the whole routing, this approach requires of detecting the number of hexadecimals in common in the prefix, which represents the index of the row, and the first hexadecimal not in common, which represent the Column key.

This module contains also the functions for modifying the table and some utility functions relative to it.

3.1.4 Leaf_set.erl

Leaf_set.erl implements the Leaf Set; the Leaf Set is represented as a Tuple of two lists, *Left* and *Right*, representing the two halves of it. The two list contain at most $L2$ elements, represented as tuples key-node information.

The module also offerts functions for modifying the leaf set and some utility relative to it.

3.1.5 Node_actions.erl

Node_actions.erl implements some utility functions used by the upper modules to interact with files, sending messages, or broadcasting messages to lists, routing tables, or leaf sets.

The most important function is *full_route()*, which implements the full navigation of the routing table, and, in case of no match or it reaches the bottom, the leaf set is analyzed; the function returns the information of the routed node, or *route_end* if the current node is responsible for the key in input.

3.1.6 Pastry_actions.erl

Pastry_actions.erl implements all the functions for handling the pastry requests and responses to the other nodes in the network, plus the handling of the self messages for peridoc events. The main functions are:

- *join()*: function used for responding to the *join* message with the routing table row, and for forwarding the message to the next node of the routing; function used by the node to join the network through another known node;
- *join_res_handle()*: function used for updating the routing table and leaf set after a *join_response*;
- *exit()*: function for handling the exit of another node from the network;
- *keepalive()*: sends keepalive to all known nodes;
- *update_keepalive()*: updates the keepalive of given nodes;
- *keepalive_res()*: responses to the keepalive to make sure that, even if a node knows another but not viceversa, the one knowing still receives keepalives from the other;
- *check_expired_nodes()*: checks which nodes are disconnected and updates the tables;
- *share_info()*: shares the info about the nodes with the known nodes.

3.1.7 File System

The files in this project, since it is run on a limited number of machines, are stored all in a */files* folder. The folder contains a subfolder per each node, which contains the files owned by the node and a folder */backup*, storing a folder per each node with their files backed up.

3.1.8 Backup_actions.erl

Backup_actions.erl implements all the functions for sending and handling the backups. For design choice, the backups are propagated to the whole leaf set and updated according to it. The main functions are:

- *backup()*: function used for propagating a file to the leaf set for backup;
- *backup_res()*: function used for saving a backup sent by other node;
- *backup_remove()*: function used to communicate the leaf set to remove a file from backup;
- *new_leaf_backup()*: function used to check if a node is added to the new leaf set, and in that case it receives a backup of each file;
- *old_leaf_backup()*: function used to check if a node is removed from the new leaf set, and in that case it receives a message to delete the backup folder;
- *update_leaf_backup()*: function used for checking in a list of nodes which ones entered the leaf set and which ones exited, in order to propagate or remove backups;
- *backup_update()*: function used to update the owner of the backups of a failed node. The routing is performed and, if the current node is the owner, the node moves the files to its folder, otherwise it forwards the request;
- *backup_find()*: request to find the owner of a file. If reaches the end, a *backup_found* response is sent;
- *backup_found()*: updates the owner of a file on a *backup_found*.

3.1.9 Web_responses.erl

Web_responses.erl implements all the functions for handling and responding to the webserver. The main functions are:

- *find_store()*: function that, given a filename, finds a storage for it using pastry routing and responds to the web server;
- *store()*: function that, given a filename, it stores it on the node;
- *find()*: function that finds a file and sends it back to the webserver;
- *delete()*: function that, given a filename, finds for it using pastry routing, deletes it, and sends and ack to the web server;
- *get_file_res()*: function that starts a flooding for retrieving all the names of the files in the network; it uses a blacklist on the *Msg_Id* for avoiding infinite flooding;
- *get_files_res_handle()*: function that updates the list of files with the new nodes received during the flooding;
- *all_files_res()*: function that responds to the webserver with the list of files;
- *check_expired_blacklist()*: function called periodically for cleaning the expired entries of the blacklist.

3.1.10 Node.erl

Node.erl implements a Node of the Pastry network. Nodes are started with the function *start_node()*, requiring the name and the name of their Erlang node, in order to register their mailbox and be reachable; the new node also initializes Pastry's tables. If *Starter* is defined, the node uses the starter node to join the network, sending a join message. At the end of the initialization, all periodical self messages are scheduled and the node enters an infinite loop, in which it just receives messages and handles them.

3.1.11 Controller.erl

Controller.erl is a simulation file used for spawning multiple nodes on one single erlang node and connecting them to the Pastry network. This file is also responsible for creating dummy files per each node, for making the simulation more real. Once the Controller is started, it receives in input commands from the user for spawning and killing nodes.

3.2 Java Web Server

3.2.1 ErlangMessageDTO

ErlangMessageDTO is a java class that acts partially as a DTO for the database of this project, which is represented by Pastry. This class is then responsible for storing all the information regarding the Erlang Messages for communicating with the distributed system, since the messages to the webserver have all a standard structure.

3.2.2 ErlangMessage

ErlangMessage is an abstract java class that acts partially as a DAO, since it allows to access both for reads and writes in storage by using Erlang messages. This class implements a set of functions to call in order, used for wrapping, unwrapping, sending and receiving message, and for some integrity checks over the messages. This class also has two abstract methods to get and set the content of the message, which are implemented in classes dedicated to the single message types, since the payload of them differs partially.

3.2.3 JavaErlangConnector

JavaErlangConnector is a class used to send requests from the webserver and get a response from Pastry. After the initialization, in fact, it is necessary to simply call the dedicated function for the web request to get a response; these function do not return *OtpErlangObjects*, in order to isolate those objects only to the layer directly communicating with Erlang, and to make it opaque to the webserver.
The constructor takes in input various arguments:

- *pastryNameIn*: name of the Erlang node to connect to, it can be any Erlang node. This implies that the system can be made fault-tolerant but for simplicity the same node is always used.
- *pastryMailBoxIn*: name of the mailbox of the previously mentioned Erlang node
- *cookieIn*: always *"pastry"*
- *selfNameIn*: the name of the Erlang node that will be created to communicate with Pastry, it has to be unique so it includes threadId
- *selfMailboxNameIn*: name of the mailbox for this node, it has to be unique so it includes threadId

Each time a new request is made and to fulfill that request a communication with Pastry is necessary, a new *JavaErlangConnector* instance is created. This is to ensure that requests are handled concurrently, the alternative would be to use a singleton shared across the servlets which would result in a performance hit.

3.2.4 WebServer

The webserver is implemented via *Java servlets* and *Tomcat webserver*. There are three packages:

- *pastry*: contains all the servlet responsible for creating responses for the various endpoints
- *javaerlang*: contains all classes mentioned above in the *Java Web Server* section, these are used by servlets to obtain data from *pastry*
- *cleaner*: contains classes to periodically clean the *files* directory in the webserver

3.2.5 SearchServlet

This servlet extends a *HttpServlet* and it's mapped to endpoint */search*. Once a *GET* request is received, the servlet opens a connection with *Erlang* via *JavaErlangConnector* and then calls the method *find()* with the filename extracted from the URL.
If the file is found, it's saved in the *files* directory of the webserver and the positive response is returned to the client in form of *JSON*. The *files* directory is there simply to avoid making two requests to *pastry*, the first one when the client searches the file and the second one when it downloads a file. By storing the file locally, the second request is avoided.

3.2.6 DownloadServlet

When this servlet receives a *GET* request on */download*, it will look into the *"files"* directory for the file specified in the request parameter. If the file exists, it will copy the binary data in the response using the *MIME* type *"application/octet-stream"*. If the file does not exist, it will return a JSON with an error.

3.2.7 UploadServlet

This servlet responds to a *PUT* request on */upload*. The request will contain the file name and the relative binary data. The servlet will extract the file name and binary data from the request and pass them to the method *store* of *ErlangJavaConnector*. If the file is successfully stored, it will return a *201 Created* status code. If there is an error during the storage process, it will return a *500 Internal Server Error* status code.

3.2.8 DeleteServlet

This servlet, when receiving a *DELETE* request at the endpoint */delete*, will extract the file name from the request object. It will then call the *delete()* method of *ErlangJavaConnector* passing the extracted file name. If the file is successfully deleted, it will return a *200 OK* status code. If the file does not exist or there is an error during the deletion process, it will return a *404 Not Found* or *500 Internal Server Error* status code, respectively.

3.2.9 FileGetterServlet

This servlet responds to the *GET* request on endpoint */allfiles*. It will call a method to retrieve the list of all file names present in the current *pastry* instance. The servlet will then return a *JSON* object with an array *names* containing the file names. If there is an error during the retrieval process, it will return a *500 Internal Server Error* status code.

3.2.10 Cleaner

The *"files"* directory used by the webserver to temporarily hold files searched by user is cleaned by a periodic task. The period is defined in the *Common* class, which also contains a method to get the full path of the *files* directory in the environment which is running *Tomcat*.
The *cleaner* uses a *ServletContextListener* to create an executor and spawn a single thread, which will run periodically, when the context is initialized. The executor will be shutdown once the context is destroyed.

3.2.11 SemaphoreManager

Since there are at least two threads working on the *files* directory, *DownloadServlet* and *Cleaner*, a semaphore is used to ensure mutual exclusion when making changes to this directory. The double check lock paradigm was used since this is a multithreaded environment.

3.3 Bash scripts

Two scripts are used to easily deploy this application:

- *deploy.sh*: copies the *erlang src* files as well as the *WAR* file into the containers via *"scp"*
- *start.sh*: creates three terminals, connects via *ssh* to the *VMs*, starts *erlang controllers* and *tomcat*