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.

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 *bitstring*s; 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.

*Bitstring*s, 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 Backup_actions.erl

Backup_actions.erl implements all the functions for sending and handling the backups. For design choice, the backups are kept only by the nodes in the leaf$_set$.

- *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;

4 Testing and Results

Once Mininet and ONOS are initialized, the network is tested to guarantee that it works as inteneded by the VPLS. The network is tested both with the *ping* and both with a configured broadcast ethernet frame. The broadcast is executed using the python script *frameBroadcaster.py*.

For checking that the VPLS are working as intended, we used *Wireshark*, checking which interfaces and devices receive the ethernet frames broadcasted. Different topologies were initialized as *net.json* and run. The topologies and some of the screenshots of wireshark are shown below, to prove how only the interfaces in the same VPLS are reached by the broadcast.

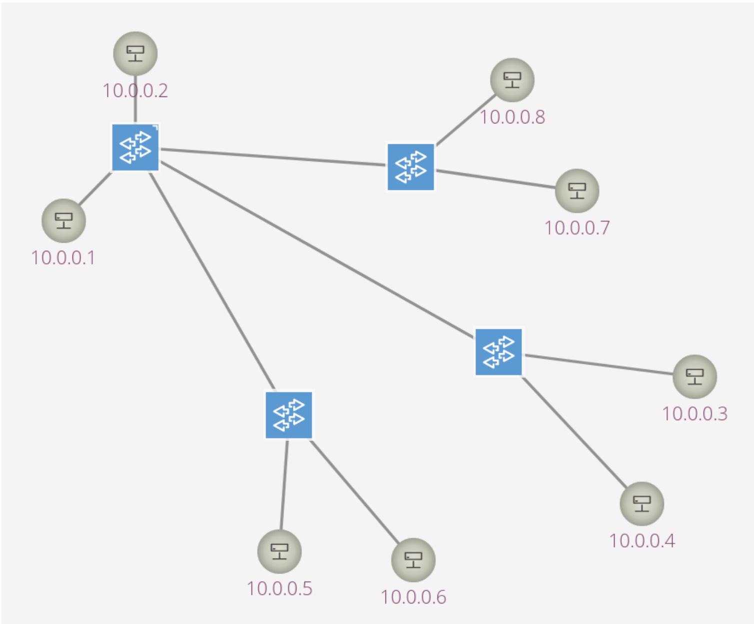

Figure 1: Topology 1

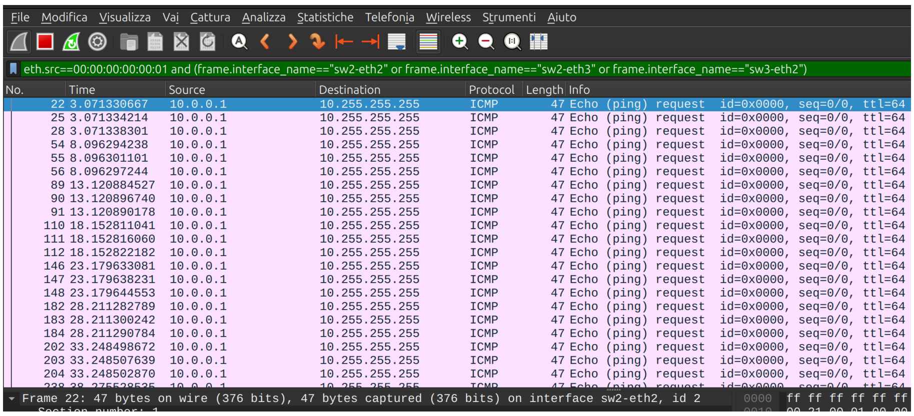

Figure 2: Broadcast example 1: servers reached

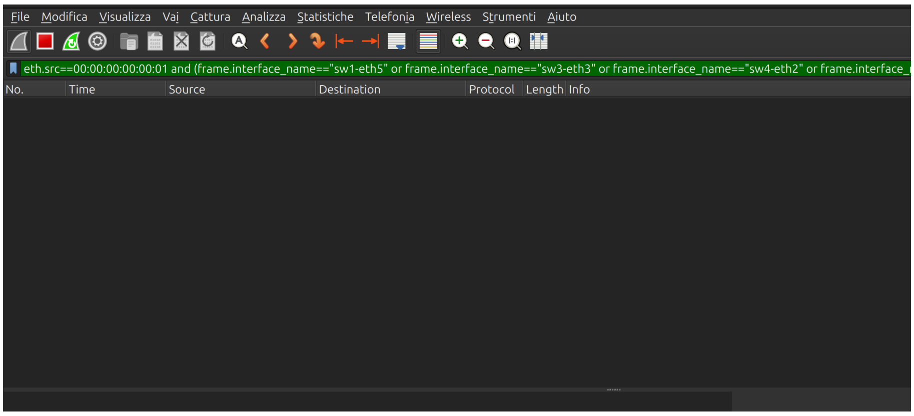

Figure 3: Broadcast example 1: all other interfaces not reached

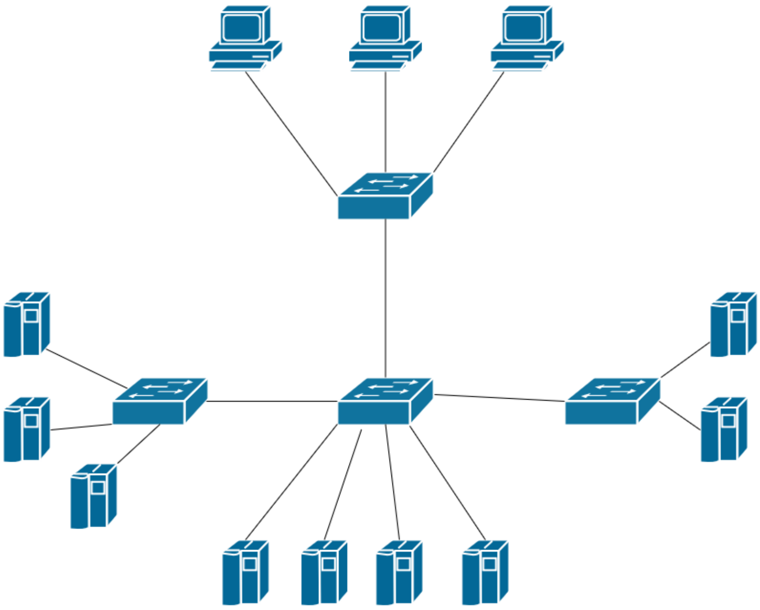

Figure 4: Topology 2

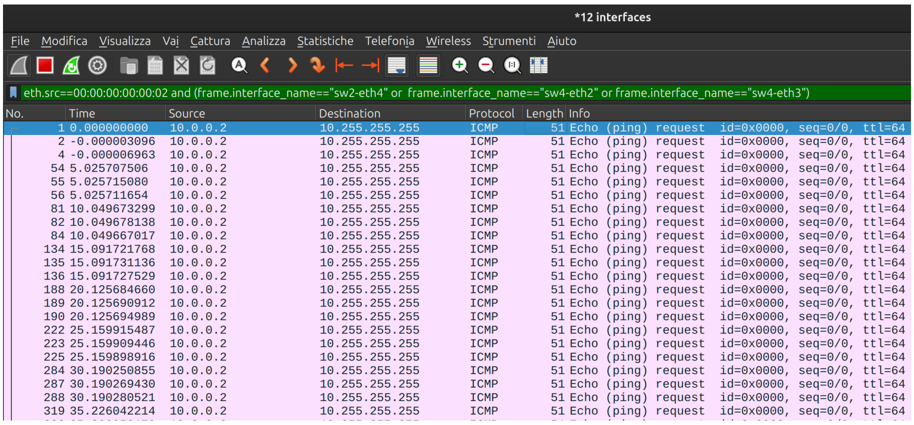

Figure 5: Broadcast example 2: servers reached

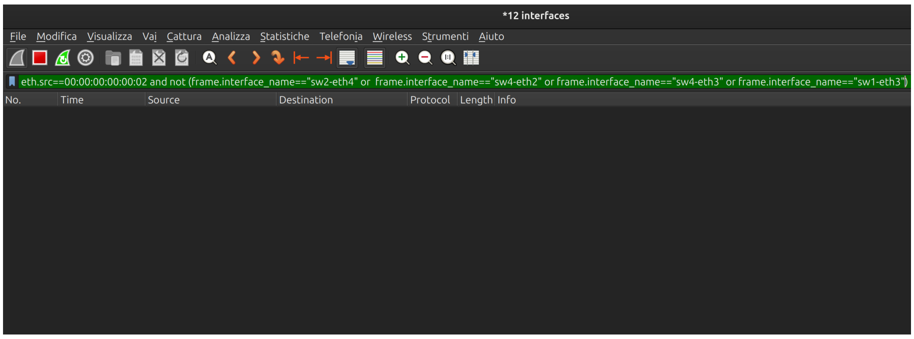

Figure 6: Broadcast example 2: all other interfaces not reached

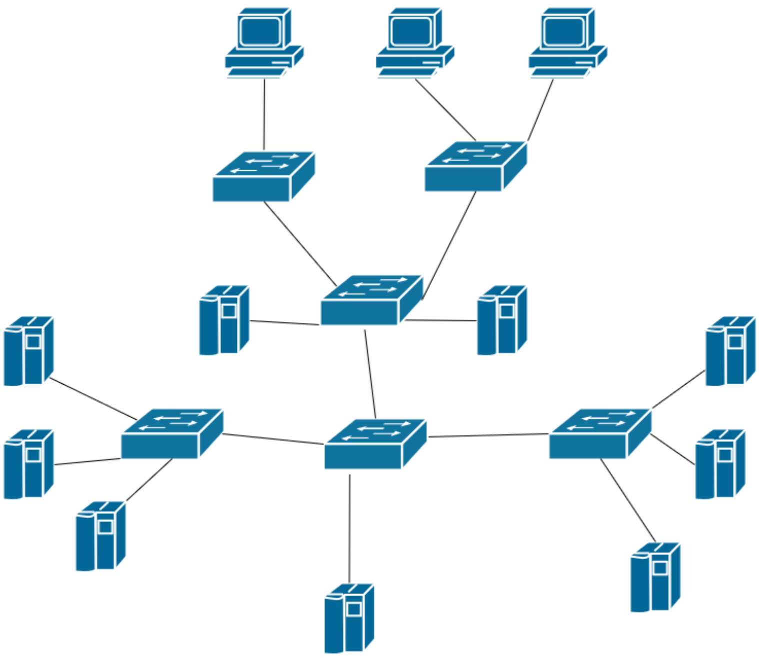

Figure 7: Topology 3

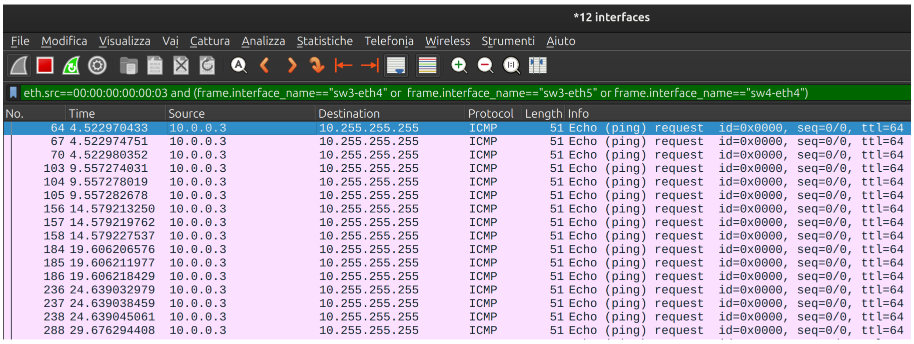

Figure 8: Broadcast example 3: servers reached

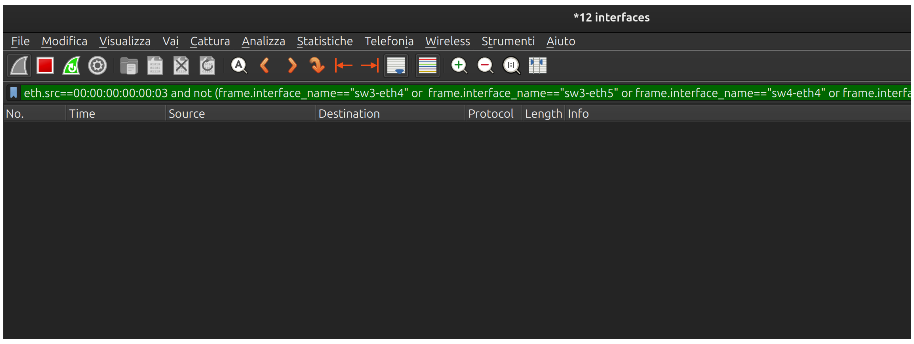

Figure 9: Broadcast example 3: all other interfaces not reached