

Peer-to-peer Streaming System

Distributed Systems Project

Gabriele Voltan, Gianluca Zavan
DMIF, University of Udine, Italy

April 10, 2024

Abstract

This paper presents the work done for the distributed systems project. In this work, we have created a peer-to-peer streaming system capable of searching for the best channel to use between two nodes to communicate.

Chapter 1

Introduction

In this chapter the problem and the goal of the project are presented, with the aim of introducing the work done.

1.1 Problem

The problem which this project is meant to solve is the following:

“N peers want to share data with each other in pairs (P_1, P_2) , using the available communication channels, each having a bandwidth B . When a peer P_1 wants to establish a connection with another peer P_2 it is necessary to establish the links it will use. These links must be chosen in such a way that the final communication channel has at least a bandwidth equal to that requested by P_1 and that it has maximum bandwidth.”

Glossary

To avoid any misunderstandings caused by the incorrect use of similar terms, we now provide a list of the main terms with their meanings:

- **Peer:** a component of the network, represented as a node in the graph;
- **Link:** a connection between two peers, represented with an edge in the graph;
- **Channel:** an edge from P_1 to P_2 or a path in the graph starting from P_1 and terminating in P_2 ;
- **Path width:** the minimum weight of a path.

Problem description

In this subsection the main points of the problem are highlighted, which are:

- Peers should be free to join or leave the network at any time;

- If a channel fails, the connection should be recovered, if possible;
- The system should support multiple streams at the same time;
- Every request should be satisfied or rejected explicitly (peers should not wait indefinitely to get the bandwidth they want)

In the future, other important features that should be implemented are:

- When a channel is used by a peer, the capacity of each edge of the channel should decrease according to the bandwidth requested;
- If a bandwidth request is satisfiable by the network (there exists a path with a width greater or equal to the one requested), then it should be eventually satisfied;

Problem modeling

Since the problem concerns a network of peers, connected to each other via links, the simplest way to model this structure is a graph. As mentioned in the glossary, in our case the nodes of the graph will be the peers of the network, and the edges will be the links between peers in the network.

The problem that the project aims to solve can be thought of as a Widest Path problem on graphs. The *widest path problem* is the problem of finding a path between two designated vertices in a weighted graph, maximizing the weight of the minimum-weight edge in the path[1].

Chapter 2

Analysis

In this chapter, we describe in detail the functional and non-functional requirements of a solution for the problem.

2.1 Functional requirements

The functional requirements of the proposed solution are:

- Peers must be free to join or leave the network at any time;
- Peers must be able to choose who should send them the data;
- Peers can choose how much bandwidth they want to request;
- If the request made by a peer is satisfiable (i.e. there exists a path, from the peer that makes the request to the peer that it wants to contact, such that each link of the path has a bandwidth greater or equal to the one requested for the communication), the peer must be able to calculate the best channel to use;
- Every peer must be able to communicate with every other peer in the network;
- A peer must be notified if their band request is not satisfiable by the network;
- If a peer wants to communicate with someone who is unreachable, it gets notified.

In the future, other functional requirements that should be implemented are:

- If the request is not satisfiable at this moment, but can be satisfiable in the future when other nodes terminate their communication, the request is queued and will be satisfied sooner or later;
- If the request can never be satisfied by the network (even when all the other nodes stop communicating), the request will be rejected.

2.2 Non functional requirements

The non-functional requirements of the proposed solution are:

- Reliability of control messages (i.e. all the messages exchanged except streaming ones) exchanged over links between peers;
- A centralized monitoring interface to explore and inspect the network, for debugging purposes;
- Reliability of the solution (widest path) found by a peer at any given point;
- All the information regarding the graph will be written in files saved locally, without using a DBMS;
- A piece of data is provided only by a node at a time, so there is a single source for it (only the provider can change it);
- Every peer has a unique name.
- When a peer joins the network or when a peer detects the crash of another peer, a new instance of a MST calculation will be launched

In the future, other non-functional requirements that should be implemented are:

- If a peer wants to leave the network, it should wait that the connections passing through it terminate;
- If a peer P_1 asks for bandwidth before another peer P_2 , P_1 's request should be handled before P_2 's.

To fulfil the requirements, Erlang has been chosen to realize the distributed components of the system because it's designed exactly for this kind of task. Python has been chosen to implement the web interfaces, thanks to the great availability of libraries and because it allows fast prototyping. To simulate the network and allow the hosts to be autonomous, Docker has been preferred over virtual machines because of its lighter resource requirements.

CAP Theorem

The system must guarantee:

- **Consistency:** since every piece of data can be written only by the node that provides it, when that data is read by a peer it is guaranteed to be in its last version; moreover, also the consistency of the Maximum Spanning Tree is guaranteed, thanks to the fact that each node participates to the distributed computation of the MST and if a node/link crash, the MST will be re-calculated. Note that this solution is an **eventually-consistent** guarantee;

- **Partition Tolerance:** if one or more peers crash the system can continue working correctly, without problems. It's clear that after the crash of one or more nodes, the topology of the network can be different from the one before the event. For this reason, a peer may no longer be able to reach a certain part of the network, but will still be able to continue to function correctly with the peers to which it remains connected.

If a peer, providing specific data requested by another peer, is down, then the data can't be accessed by other peers, so **availability** *can't* be guaranteed.

Transparencies

- **Access transparency:** all resources provided via streaming can be accessed with the same operations;
- **Location transparency:** all resources can be accessed without knowing their physical location;
- **Failure transparency:** if the channel between two nodes fails, then the system tries to establish a new one, otherwise the peer is told the connection is impossible;
- **Mobility transparency:** the resources can move without affecting the system;
- **Performance transparency:** if the configuration of one or more peers changes, or the bandwidth of the links, new connections directly use the new configuration;
- **Scaling transparency:** there are no limits on the number of peers in the network.

Chapter 3

Project

This chapter is devoted to the description of the general architectures and specific algorithms.

3.1 Logical architecture

The architecture of our streaming system is composed of a P2P network, *without assumptions on the topology*. We can't make assumptions on the topology because, during the working life of the system, it could be possible that some peers will be on different connected components of the graph. As we can see from figure 3.1, the architecture of a single node is composed of:

- Computational core unit which is responsible for calculating, in a distributed way, the Maximum Spanning Tree, and communicating with the other peers;
- Web interface that can be used by the user for requesting a connection with another peer specifying also the bandwidth required;
- Web server to offer the Web interface function and to communicate the requests to the computational core unit.

The admin node has an architecture similar to the single-peer one. The main differences are:

- Web interface can be used by the admin to modify the components of the P2P network and to show the current status of the network;
- Module for inspecting the states of the various peers of the network;
- There is not a computational core unit.

Note that the admin node is a part of the architecture inserted only for debugging and managing purposes. In a real scenario, this unit shouldn't exist, because each peer manages itself.

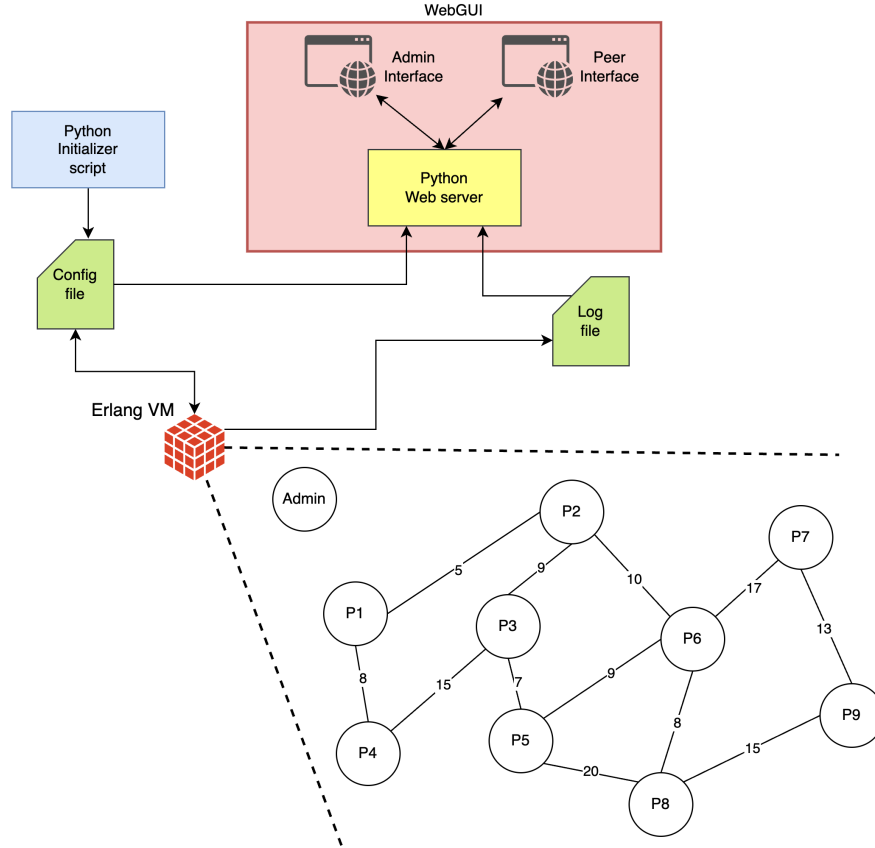


Figure 3.1: Logical architecture

Finally, there is a DNS which allows the peers to locate the peers by name and provide location transparency.

3.2 Protocols and algorithms

Most of the messages exchanged between the components of the network are in JSON format. The specific protocols used for communicating are:

- Web Server \leftrightarrow Erlang node use TCP;
- Admin Web Server \leftrightarrow Erlang node use TCP;
- Log Parse \rightarrow Admin Web Server use TCP;
- Peer \leftrightarrow Peer use TCP to establish the connection;
- Peer \leftrightarrow Peer use UDP to exchange streaming data.

In the following subsections we explain in detail the various phases and situations of our streaming system.

Initialization of the network

The initialization of the network begins with the choice of the number N of peers that will make up the network and with the random, but consistent, generation of the configuration files, which will contain the links between the various peers and their bandwidth. Once the peers have been created, each of them will ping its neighbours, which will be found in the configuration file generated in the previous step.

Peer Join network

When a new peer wants to join the network, it is first assigned a unique ID. Subsequently, it will send to neighbouring peers (indicated in its configuration file) a message $(ID_{newPeer}, B)$, where B will be the link bandwidth between the new peer and its neighbour.

When an existing peer receives this type of message, it understands that it has a new connection with the peer having $ID_{newPeer}$ as its ID and will consequently update its configuration file, adding the information contained in the received message.

Finally, since the topology of the network will have changed due to the new peer, it will begin calculating the Maximum Spanning Tree with the rest of the network.

Peer Leave network

When an existing peer in the network wants to leave it, regardless of whether it has connections using it or not, it will leave the network.

In a future implementation of the project, as indicated in the non-functional requirements, before leaving the network, the peer will need to ensure that it does not have any connections passing through it. In the latter case, it has to wait.

Communication request

If a peer P_X wants to communicate with a peer P_Y , using a bandwidth B , it will have to use the Maximum Spanning Tree of the network (which it will already know) and must not already have other open connections with the peer P_Y .

If the two preconditions are satisfied, then it will identify the path from P_X to P_Y on the MST, verifying that each link present on this path has a bandwidth greater than or equal to that requested B .

If the capacity verification phase is accomplished, then it will send along the path a message of the type $(P_X, P_Y, B, PATH)$, where $PATH$ will be a list of peers present on the chosen path between P_X and P_Y .

At this point, every peer that receives this message will know who to forward it to, and will know that a new connection is about to be established.

Maximum Spanning Tree computation

The well-known Gallagher-Humblet-Spira algorithm[2] has been chosen to perform the computation of the Maximum Spanning Tree because of its proven properties and because the system satisfies the assumptions of the algorithm. Obviously, since the original algorithm was designed to calculate the Minimum Spanning Tree, it has been slightly modified in order to be able to calculate the Maximum Spanning Tree. Since we can't assume that the weights of the edges representing the links in the network have all distinct weights, a specific strategy is used to break ties. Let's consider two edges $A = (w_A, from_A, to_A)$ and $B = (w_B, from_B, to_B)$ where w_x represents the weight of the edge, $from_x$ is the label of the node from which the edge is coming and to_x is the label of the node to which the edge is directed to, then:

1. If $w_A > w_B$ then A is chosen;
2. If $w_A = w_B$ then $\min(from_A, to_A)$ and $\min(from_B, to_B)$ are compared, and the edge providing the greater value is chosen;
3. If the values are equal, then $\max(from_A, to_A)$ and $\max(from_B, to_B)$ are compared in the same way.

Data streaming

Once a connection is established between a peer P_X and a peer P_Y , they can begin to communicate, initiating data streaming. The messages that peer P_Y will send to peer P_X will have a form like $(P_Y, P_X, PATH, MSG)$ and should be forwarded by all the nodes specified in $PATH$ until it reaches P_X .

If during message routing, a peer on the $PATH$ notices that it cannot reach the next peer, then it will consider it as it has failed and will initiate the crash handling procedure, illustrated below.

Crash management

This procedure is invoked whenever a peer P_X wants to communicate with a peer P_Y , that it previously knew, but is unable to do so. The problem is detected when a peer, on the channel from P_X to P_Y , can't ping anymore a neighbour that has to be contacted for a specific request.

If this condition occurs, the peer that detects the crash closes all its active connections and starts a new calculation of the Maximum Spanning Tree.

Once the new MST is calculated, the peer tries to re-establish its previous connections, using the procedure explained in the “Request to Communicate” section.

MST calculation request management

When a peer receives a request to start calculating a new MST it will apply a procedure similar to that described in the Crash management subsection.

Initially, it will close active connections to take part in the calculation of the new MST. Subsequently, it will try to re-establish the connections it had previously opened.

Communication termination

When a peer P_X wants to terminate its communication with a peer P_Y , it sends along the chosen path a message of type $(P_X, P_Y, PATH, close)$ and closes the connection from its side. The message should be forwarded along the path and when P_Y receives it, it closes the connection as well.

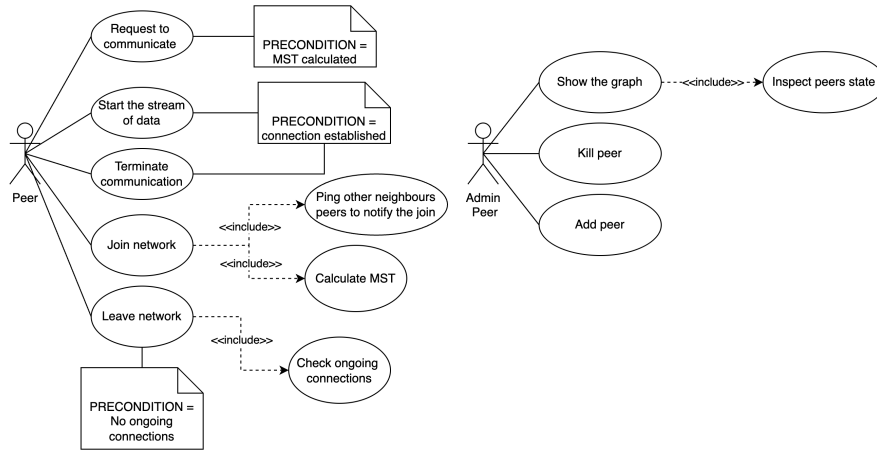


Figure 3.2: Use Case diagrams

3.3 Physical architecture and deployment

This section aims to give a comprehensive description of the physical architecture of our streaming system. In general, all the system runs on a local network (for validation and testing purposes): it is clear that in a real scenario, all the peers would be physically dislocated.

The physical architecture of each peer of the P2P network is composed of:

- Docker container which contains all the other components;
- Erlang node which executes all the peer core logic. In detail, as we can see in Figure 3.3, the structure of the Erlang node is composed of various processes:
 - A supervisor whose goal is to restart the other processes in case of failure;
 - A handler for the commands received from the web interface of the peer itself and from the admin. This is realized with a TCP socket that accepts JSON formatted data;
 - Multiple connection handlers which are independent to allow the node to maintain the connections and stream data;
 - A process to compute the MST when it's needed;
 - A logger which monitors all the other processes and collects data about them;
- Web interface, written in HTML, that can be used by the user for requesting a connection with another peer specifying also the bandwidth required;
- Web server, implemented with Flask, to offer the Web interface function and to communicate the requests to the Erlang node;
- JSON configuration file which contains the links of the peer;
- Log file which contains useful information about the peer (e.g. ongoing connections).

The physical architecture of the admin component is similar to the one implementing the peers. The main differences are:

- Web interface can be used by the admin to modify the components of the P2P network and to show the current status of the network;
- Log parser which collects all the information stored in the various log files of the peers in the network;
- There is not an Erlang node.

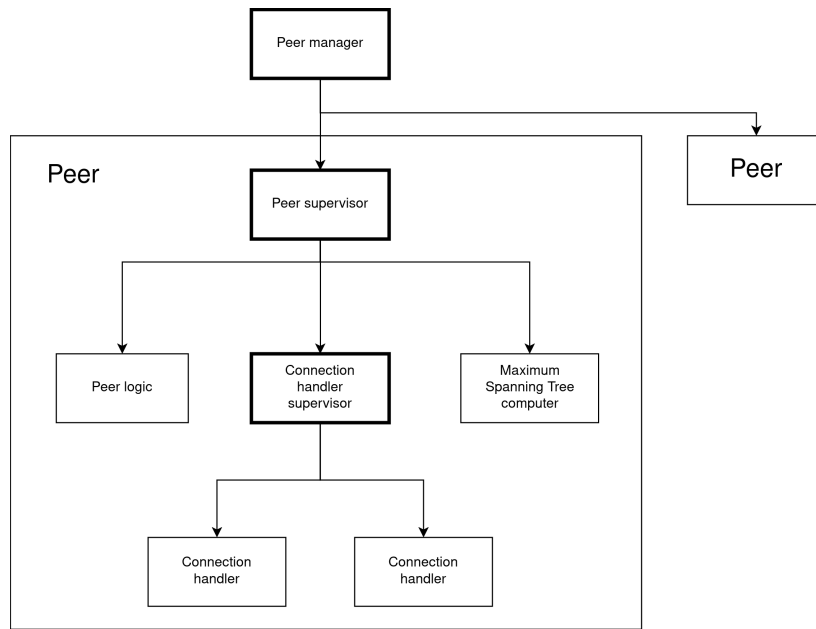


Figure 3.3: Erlang Peer node architecture

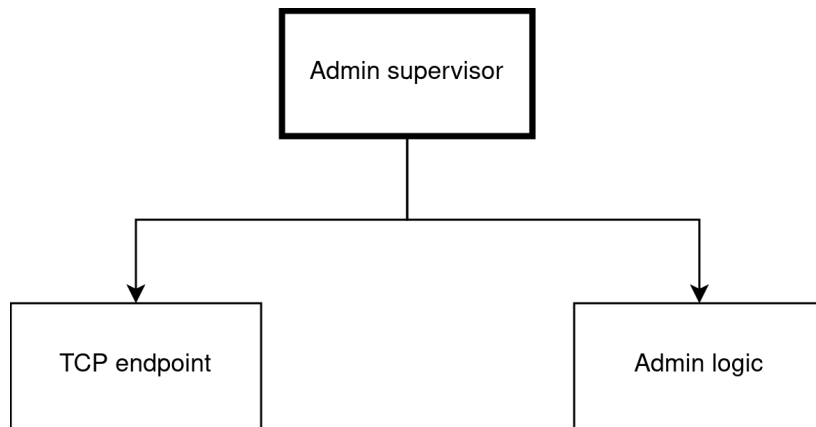


Figure 3.4: Erlang Admin node architecture

All the hosts in the network can refer to each other with their hostnames, thanks to a DNS. Since Docker Compose already configures each container in such a way that each of them knows the others' hostname, a custom DNS won't be implemented.

3.4 Development plan

The development plan of the project should start from the development of the basic components and features of the system and then continue enriching the system with more complex features. Since it's very difficult to predict with accurate precision the future plans, we give a generic development plan that can guide us through the implementation of the system. The steps are:

1. Creation of the network, with its peers and connections, in such a way as to be able to establish communications between them. In this phase the main goal is to make Docker containers coexist with Erlang nodes;
2. Implementation of the Maximum Spanning Tree calculation algorithm;
3. Creation of the admin interface, in Python and HTML, to get a view of the graph and implementation of the join/deletion of a peer;
4. Implementation of Erlang functions to manage events, such as requests to communicate, node crashes and participation in the MST calculation;
5. Creation of the peer interface to make a request to communicate and to stop an existent communication, in Python and HTML, and implementation of the external request handler for the peer, in Erlang;
6. Implementation of basic data streaming, to simulate a real scenario;
7. Adding, in the specified order, the following extra features to the system:
 - a) Peer leaving only after all the connections passing through it are terminated;
 - b) Guarantee the delivery of requests in the order they are sent;
 - c) Management of the reduction of available bandwidth after the allocation of a channel by a pair of peers;
 - d) Management of bandwidth requests that can be satisfied by the network but at a later time.

After each development phase, an accurate testing phase will follow, in order to validate the functioning of both what has been implemented in the specific phase and to test the integration of the latter with everything that has been implemented previously.

Chapter 4

Implementation

This chapter provides a detailed explanation of the implementation of the system, illustrating the patterns, techniques and technologies used.

4.1 Detailed architectural overview

The system has been designed to be as distributed as possible. Every component exchanges information with the others only by message passing, so every process could seamlessly be physically located in a different location with respect to the others. The network itself is composed of peers, each of which is implemented using:

- An Erlang supervisor that manages all the other components, which are the core logic module, the connection handler supervisor and the process used to compute the MST;
- A “peer logic” module, which implements the OTP *gen_server* behaviour and acts as an interface for all the peer functions. The process communicates with the admin module and with the other peers, and can ask its supervisor to spawn a new connection handler when needed;
- A process that executes the GHS algorithm for the module, and reports to the core logic module the information about the MST or if another peer becomes unreachable during the MST computation;
- A connection handler supervisor, which manages the connection handlers started by the peer logic module;
- Multiple connection handlers, one for each established connection.

Since in the demo setting the peers are created and added to the network by the admin node, a “peer manager” is used to start each peer, and has all of them as children. This component exposes a *spawn_node* function that is called by the admin when needed. In a real setting, where the peers are not controlled

by the admin, the peer manager should not be present, and the whole system has been designed to work without it, since it has been used only to allow a better decoupling between the peers and the admin.

All the processes run on a single Erlang VM, but with minimal effort, the system can run on distributed nodes.

High level implementation

The main operations implemented in the systems are here described from a high-level perspective, with a focus on the messages that are exchanged between the peers. The considered functions are the joining of a peer in the network (fig. 4.1), the start of a new session to compute the MST in a distributed fashion (fig. 4.2) and the request to communicate from a peer to another (fig. 4.3).

Join network

When a peer D wants to join the network by connecting to other peers which are known, first it spawns a new process Mst_D , to participate in the following sessions of computation of the MST, and announces its presence to its neighbours with a message of the form $\{new_neighbor, D, Weight, Mst_D\}$. When this message is received by the neighbours, they update their neighbour list and send back the PID of their own process used to compute the MST. Peer D will consider as actual neighbours only the peers that sent a response to it. At this point, a new MST will be computed starting from D .

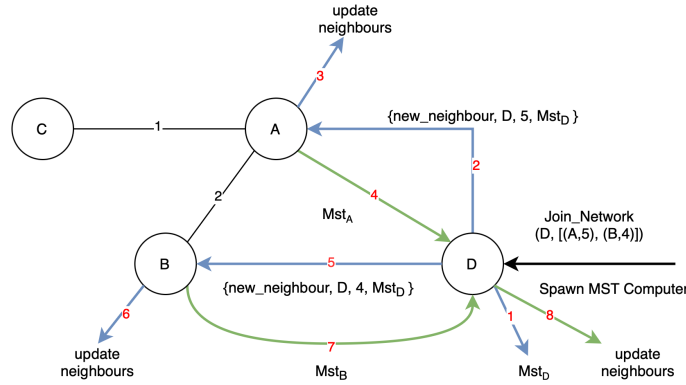


Figure 4.1: Join network executed by node D

Start the computation of the MST

When a peer D wants to compute a new MST with the others, first it asks the admin node for a new session ID, which is an integer guaranteed to be greater

than all the session IDs used before. This ID will be used by the peers to determine if the information about the MST that they have stored is updated or not. After having received the new session ID, peer *D* starts an *echo wave* to inform the rest of the network that a new MST is about to be computed. During the echo wave, each time a peer receives back all the tokens sent to its children peers, it starts to compute the MST using the GHS[2] algorithm by sending messages to all the neighbours who sent a response (the others are considered as unreachable), then sends back the token to its parent. The messages exchanged during the algorithm should always be answered with an acknowledgement: if the acknowledgement is not received in a specified time frame, then the peer who should have sent it is considered as unreachable, and this will trigger a new echo wave to recompute the MST. Eventually, all the peers who are unreachable for any reason will be not considered anymore as part of the network.

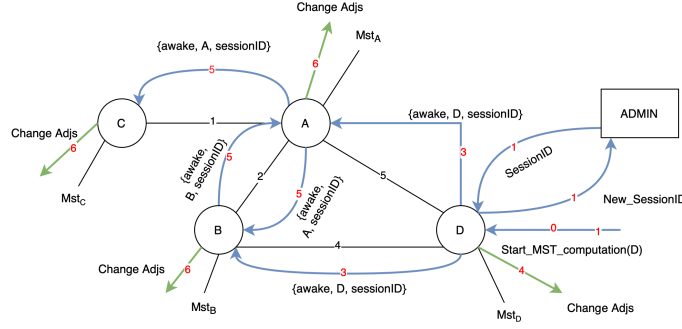


Figure 4.2: Start MST computation from node D

Request to communicate

If a peer *A* wants to establish a connection with a peer *C* using band *Band*, first it spawns a new connection handler ConnPid_A process that will be used to exchange data with *C*, then sends a $\{\text{request_to_communicate}, A, C, \text{Band}, \text{ConnPid}_A\}$ message along the link which is part of the MST and that goes towards *C*. If *A* has been a sink tree root during the GHS algorithm, then it could have stored a partial routing table and know exactly on which link to send the message. If otherwise it was a leaf, the message is sent over its only link which is part of the MST. If a peer *B* is on the path from *A* to *C*, it spawns as well a connection handler ConnPid_B , then it forwards a message of the type $\{\text{request_to_communicate}, A, C, \text{Band}, \text{ConnPid}_B\}$ on its link towards *C* in the MST. When the request finally arrives to *C*, it spawns a connection handler ConnPid_C and sends a message of the type $\{\text{ok}, \text{ConnPid}_C\}$ to the peer from which the request came from, in this case *B*, so *B* can update the information about the connection and report back $\{\text{ok}, \text{ConnPid}_B\}$ to *A*,

which will know that the communication request has succeeded. At any time, if the link to use to forward the message has a bandwidth that is strictly less than *Band*, the request to communicate is not forwarded and the error is reported back to node *A*.

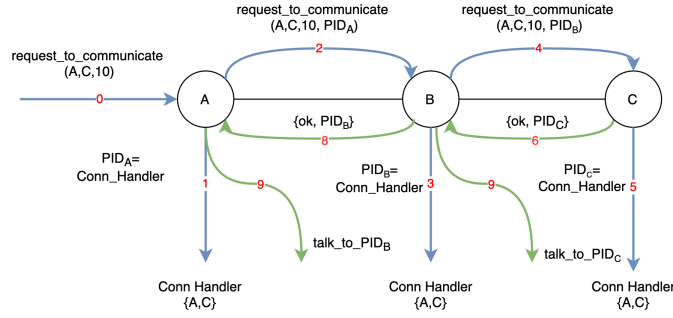


Figure 4.3: Request to communicate from A to B

4.2 Modules documentation

This subsection describes the most important Erlang modules, listing and explaining the public functions, i.e. the exported ones, and also mentioning some private functions, considered important for the functioning of the module.

Admin module

The admin module, whose file name is *p2p_admin.erl*, is the module that centrally deals with receiving orders for the creation of peers. The functions it exposes are:

- **spawn_node/2**: this function has the task of sending a request to the Peer Manager module, for the creation of a new peer. The parameters required by this function are the name of the peer, i.e. an integer, and the list of its neighbours, i.e. a list containing lists of the form [NeighborID, EdgeBand];
- **start_link/1**: this function has the task of starting the admin process.

The private functions are:

- **init/1**: sets up the state of the process and starts the event logger for the GHS algorithm that will be executed later;
- **loop/1**: listens for messages tagged with **new_id** or **timer** that can be sent by peers to ask for a new session ID for the MST computation or a timeout value (in seconds) for their requests;

- **get_peer_count/0**: inspects the network to count how many active peers are present;
- **compute_timer_duration/1**: applies a heuristic based on the number of peers in the network (obtained with `get_peer_count`) to produce a reasonable timeout duration for peer requests;

Peer supervisor module

The supervisor module of a peer, whose name is *p2p_node_sup*, is the module that creates the processes for managing connections and calculating the MST. The functions it exposes are:

- **start_connection_handler/1**: start a new connection handler for the peer;
- **start_mst_worker/2**: starts a worker process for the Minimum Spanning Tree (MST) computation. This function is typically used by a supervisor to start a worker process;
- **start_link/2**: starts the node's supervisor process.

The private functions are:

- **init/1**: sets up the child specifications for the peer logic process;

Peer logic module

The peer logical module, whose name is *p2p_node.erl*, is the most important module, as it is the one responsible for carrying out all the main activities that a peer can perform, such as opening a connection, sending data, calculate MST, leave the network, etc. The functions it exposes are:

- **start_link/1**: starts the server;
- **init_node_from_file/1**: allows you to start a new peer, having the characteristics (ID and neighbours) indicated in a JSON file taken as input;
- **request_to_communicate/3**: allows you to send the request to start communicating. It takes as input the node that wants to send the request, the node to which to send the request and the requested bandwidth. Return `{ok, ConnHandlerPid}` if the request is successful, `{timeout, NodeName}` if `NodeName` on the path to destination peer times out on the request, `{noproc, NodeName}` if `NodeName` doesn't exist on the path to destination peer, `{no_band, NodeName}` if the connection to `NodeName` on the path doesn't provide enough band;
- **close_connection/2**: closes an open connection between two peers, requested as input. Returns `ok` if the request is successful, `{timeout, From}` if `From` didn't answer in time, `{noproc, From}` if `From` doesn't exist, `{shutdown, From}` if `From` has been shut down from its supervisor;

- **send_data/3**: given an existing connection, previously established with the `request_to_communicate` function, sends data, requested as input, from one peer to another, also requested as input. Returns `ok` if the request is successful, `{no_connection, {From, To}}` if there is no active connection between the peers, `{timeout, From}` if From didn't answer in time, `{noproc, From}` if From doesn't exist, `{shutdown, From}` if From has been shut down by its supervisor;
- **start_mst_computation/1**: allows the peer whose PID is the one taken as input to begin calculating the MST, notifying its neighbours. Returns `ok` if the request is successful, `{timeout, Ref}` if Ref didn't answer in time, `{noproc, Ref}` if Ref doesn't exist, `{shutdown, Ref}` if Ref has been shut down from its supervisor;
- **leave_network/1**: allows the peer, whose PID is requested as input, to leave the network, also closing any open connections and abandoning any calculation of the MST. Returns `ok` if the request is successful, `{timeout, Ref}` if Ref didn't answer in time, `{noproc, Ref}` if Ref doesn't exist, `{shutdown, Ref}` if Ref has been shut down from its supervisor;
- **join_network/2**: allows the peer to join the network, with a given PID and indicated neighbors;
- **get_state/1**: analyzes the state of a peer whose PID is taken as input.

The main private functions of the module implement the **gen_server** OTP behaviour:

- **init/1**: initializes the internal state of the peer, with information such as the neighbouring peers and the references to the connection handlers for each active connection;
- **handle_call/3**: handles synchronous requests to the peer for the messages
 - **get_state**: returns the raw state of the peer as shown in listing 4.1;
 - **{join, Adjs}**: update the list of neighbours of the peer, ask them for the PID of their process to compute the MST, and send them their own MST process PID;
 - **{new_neighbor, NeighborName, Weight, MstPid}**: message received by a peer when a new peer enters the network and announces itself and its MST computer PID. The peer updates its neighbour list and answers with its own MST computer PID;
 - **start_mst**: when a peer receives this message, first it starts an echo wave to inform all the other peers that a new MST is about to be computed. This is done by calling **echo_node_behaviour/5**. After the convergecast phase of the wave is completed, the peer can trigger the start of the computation of the MST by sending a message to its MST computer process;

- `{request_to_communicate, {Who, To, Band}}`: this message is received by the peer who has to start the communication. It asks its supervisor to start a new connection handler, and forwards its PID along the MST in order to reach the destination peer;
 - `{request_to_communicate, Who, To, Band, LastHop}`: message received by the intermediate and destination nodes when a connection has to be established. The peer starts a new connection handler and remembers that the messages coming from `To` in the future (since the communication can happen in both ways) have to be forwarded to `LastHop`, which is the connection handler PID of the peer from which the message was received;
 - `{close_connection, To}`: this message causes the peer to shut down the connection handler for its connection with peer `To`, and send *asynchronously* a `{close_connection, To}` message along the path that was used to establish the connection;
 - `leave`: the peer shuts down all its connection handlers and its MST process, then terminates;
 - `{send_data, From, To, Data}`: this message means that the peer should check if there is a connection with the peer `To`, and if it is the case, send to it the binary data `Data` using the proper connection handler;
- **handle_cast/2**: handles asynchronous requests to the peer for the messages:
 - `{close_connection, Who, To}`: the peer `Who` wants to close the connection with `To`, so the intermediate/destination node receiving this message should shut down the specific connection handler and forward the message asynchronously;
 - **handle_info/2**: handles all other types of messages of the type:
 - `{awake, NameFrom, SessionID}`: if this message is received, it means that a node in the network has started the computation of a new MST. The peer knows that it has to send as well an awakening message to its neighbours and start to compute the MST, so eventually all the peers in the network will know that they should participate in the computation. The message is considered only if `SessionID` is greater than the session ID currently known by the peer;
 - `{done, {SessionID, MstParent, MstRoutingTable}}`: the message is received by the peer whose MST process has completed its task, providing the links found to be part of the tree. The peer now updates the state, and is ready to establish communications;
 - `{unreachable, SessionID, Who}`: this message is received when a message sent from the MST computer doesn't get an acknowledgement in a specified time window, so the MST process `Who` is

considered unreachable. The peer looks up in its state the other peer associated with `Who` and deletes it from the neighbours' list, then starts the computation of a new MST by awakening the remaining neighbours. From this point, the messages from `Who` will be ignored unless it announces itself again with a `new_neighbor` message as seen previously;

- **terminate/2**: handles the intended termination of the peer, who shuts down its other processes.

In particular, the state that each peer maintains is a data structure with the following fields:

```
-record(state, {
    % Name of the peer
    name :: atom(),
    % List of incident edges, with other node names
    adjs = [] :: [#edge{}],
    % List of incident edges, pids of MST computers
    mst_adjs = [] :: [#edge{}],
    % Pid of supervisor process
    supervisor :: pid(),
    % Current session of stored MST
    current_mst_session :: non_neg_integer(),
    % List of tuples {From, To} that represents ongoing
    connections
    connections = [] :: list(),
    % Pid of own MST computer
    mst_computer_pid :: pid(),
    % Current state of MST from peer's perspective
    mst_state = undefined :: computed | computing | undefined,
    % MST routing table (map) in the form #{NodeName => Edge}
    mst_routing_table = #{} :: #{atom() => #edge{}},
    % Parent of peer in MST
    mst_parent = none :: #edge{} | none,
    % Connection handler pids for each connection. #{From,To} =>
    Pid}
    conn_handlers = #{} :: #{term(), term()} => pid() }).
```

Listing 4.1: Peer state

Other private functions used mainly to perform side tasks are:

- **echo_node_behaviour/5**: implements the logic to initiate/participate in an echo wave. If during the wave a peer *P* expecting the token from its children doesn't get messages in a specified time frame (by default obtained with `get_timeout`) then all the children that didn't send back the token will not be considered anymore neighbours by *P*;
- **start_mst_computation/5**: implements the awakening of the neighbouring peers and sends a message to the MST process to trigger the start of the algorithm;

- **get_mst_worker/3**: returns the PID of the process that participates in the MST algorithm for the peer if it already exists, otherwise asks the supervisor to spawn a new one;
- **get_new_session_id/0-1**: asks the admin to provide a fresh session ID. Each call is guaranteed to return an integer that is strictly greater than the one before;
- **dump_config/1**: writes part of the node state in a JSON file to be later consumed by other components of the system;
- **validate_edges/2**: checks that all the edges passed do not cause self-loops and have a strictly positive weight;
- **get_neighbors_mst_pid/3**: implements the request to the peer's neighbours for the PID of their MST process;
- **get_connection_handler/1**: asks the supervisor for a new connection handler process;
- **get_timeout/0**: asks the admin for a timeout value to use for requests. It is computed based on the current number of nodes in the network (the greater it is, the bigger the timeout value) which only the admin knows;

Connection handler module

The connection handler module, whose name is *p2p_conn_handler.p2p*, is responsible for managing, in a practical way, connections and communications with other peers on the network. The functions it exposes are:

- **start_link/0**: starts the connection handler process;
- **talk_to/4-5**: called by the peer when it receives the PID of the connection handler of the next hop towards the destination. Changes the internal state of the connection handler so it can forward data;
- **send_data/2**: send arbitrary binary data to the next connection handler towards the destination;

The private functions are:

- **init/0**: sets up the internal state;
- **loop/1**: listens for **send** or **forward** tagged messages and performs the appropriate operation;
- **process_received_data/2**: called by the receiving end in the connection. At the time of writing, the data simply gets saved on a file.

TCP endpoint module

The requests handled by the process should be formatted in JSON, and should contain a command name under the “type” field, which can be one of: `req_conn`, `new_peer`, `rem_peer`, `close_conn`, `get_state`. Each command has its own parameters that should be specified in other fields. The name of this module is *p2p_tcp.erl* and the only exported function is:

- **start_link/1**: starts a process which opens a port on the machine and listens for incoming TCP requests;

The private functions are:

- **init/1**: opens the socket and calls the loop function;
- **loop/1**: accepts TCP requests and spawns a process for each new request, executing `handle_request`;
- **handle_request/2**: decodes the data received, calls `process_data` and then sends a reply depending on the return value of the latter;
- **process_data/1**: looks for the command specified in the JSON data and performs the appropriate operations on the P2P network;
- **process_reply/1**: depending on the outcome of `process_data`, produce a reply message in JSON format to be sent to the other endpoint of the TCP communication;

Utils module

The `utils` module, whose name is *utils.erl*, offers support functions, useful for carrying out certain necessary operations that must be repeated several times during the life of the system. The functions it offers are:

- **build_edges/2**: transforms the edges from JSON into internal format;
- **get_pid_from_id/1**: transforms numeric ID into an atom that represents the peer node;
- **init_network/1**: initializes the network from the JSON config files stored in `InitDir` taken in input.

4.3 WebGUI documentation

To interact with the system it offers a web interface, implemented using Flask and Javascript, with which you can have a global view of the network and with which you can carry out the main operations. The web interface is divided into two pages, which are:

- **localhost:8080/**: this page is the admin page, where you can see the network graph and system logs updated in real-time. More in detail, the topology of the network, whose image is generated every time the page is reloaded or an operation is performed, shows both the edges belonging to the MST (green edges) and those that do not belong to the MST (blue edges). Furthermore, the admin interface allows *adding* a new peer and *removing* an existing one;
- **localhost:8080/peer/**: this page is the peer control page, where it is possible to request to *establish the connection* between two peers, specifying a certain bandwidth, and where it is possible to *terminate* a previously established connection.

It is important to underline the fact that both interfaces, after each operation performed, show a response message useful for understanding whether everything worked correctly, or if there were errors.

Chapter 5

Validation

This chapter shows the tests conducted in order to validate the correct functioning of the system. In particular, what we want to verify is that the system respects the functional and non-functional requirements listed in the chapter 2. Moreover, we also want to provide a script for checking the correctness of the MST calculated in a distributed way with our system.

Correctness of MST

In order to verify the correctness of the MST calculated by the distributed system, we implemented a Python script. The latter, after having calculated the MST in a centralized way, using the `networkx` library, compares the MST calculated in a distributed way with the one/s (if there were more) calculated in a centralized way. If at least one is equal, it returns true, otherwise false.

Introduction to the tests

The tests carried out are of two types: automatic, i.e. carried out using a specific Erlang module; manual, i.e. carried out using the two web interfaces described in the chapter 4.

In order to verify all system requirements, the following tests were designed:

1. Creating the network
2. Adding a peer to the network
3. Removing a peer from the network
4. Opening a connection
5. Closing a connection
6. Adding a peer while computing the MST
7. Removing a peer while computing the MST

8. Removing a peer during a communication

9. Request for unavailable bandwidth

It is important to highlight how some tests, such as the addition or removal of a peer during the calculation of the MST, are carried out only with automated tests, as it would be difficult to carry them out manually, due to the little time taken by the system to calculate the MST.

Unit tests

Unit testing has been performed in Erlang's *EUnit* framework, in order to streamline the development process. Programming the tests has been useful to debug scenarios that are difficult to simulate manually, such as concurrent requests or peer failures during critical operations such as the computation of the Maximum Spanning Tree, since the operations are typically carried out in a matter of milliseconds.

Creating the network

```
init_network() ->
    % Start the main components
    p2p_node_manager:start_link(),
    p2p_admin_sup:start_link(),
    % Initialize the network
    Nodes = utils:init_network("../src/init/config_files/"),
    Outcomes = lists:map(fun(T) ->
        try
            % The call is succesful only if the peer started
            p2p_node:get_state(T),
            true
        catch
            error:_ -> false
        end
    end, Nodes),
    % Every peer should must be up
    ?assert(lists:all(fun(X) -> X == true end, Outcomes)).
```

Listing 5.1: Network initialization

Adding a peer to the network

```
join_after_init(Nodes) ->
    % Network already initiated
    % A new peer joins the network
    p2p_node:join_network(node500, [{edge,node1,node500,1}]),
    % Then it starts the computation of a new MST
    p2p_node:start_mst_computation(node500),
    timer:sleep(5000),
    AllNodes = [node500 | Nodes],
    % Check that the MST has been computed
```

```
check_mst(AllNodes).
```

Listing 5.2: Join network after initialization

```
check_mst(Nodes) ->
  States = lists:map(fun p2p_node:get_state/1, Nodes),
  Info = [FinishedMst || #state{mst_state = FinishedMst} <-
    States],
  Sessions = [SessionID || #state{current_mst_session =
    SessionID} <- States],
  % Check if all the peers have finished computing the MST
  ?_assert(lists:all(fun(X) -> X == computed end, Info)),
  % Check that all the MST sessions are the equal (no one left
  behind)
  ?_assert(lists:all(fun(X) -> X == hd(Sessions) end, Sessions))
  .
```

Listing 5.3: Check that all the peers have computed the MST and that are in the same MST session

Removing a peer from the network

```
remove_peer(Nodes) ->
  ToRemove = hd(Nodes),
  p2p_node:leave_network(ToRemove),
  % The process should not exist anymore
  ?_assertEqual(undefined, whereis(ToRemove)).
```

Listing 5.4: Remove peer

Opening a connection

```
request_to_communicate(_Nodes) ->
  % Add a new node
  p2p_node:join_network(node21, [{edge, node1, node21, 10}]),
  % Make it start the computation of the MST
  p2p_node:start_mst_computation(node21),
  timer:sleep(10000),
  Reply = p2p_node:request_to_communicate(node21, node2, 1),
  % The request should be successful
  ?_assertMatch({ok, _Pid}, Reply).
```

Listing 5.5: Successful request to communicate

Closing a connection

```
close_connection(_) ->
  p2p_node:start_mst_computation(node1),
  timer:sleep(5000),
  p2p_node:request_to_communicate(node1, node4, 1),
  % Close connection starting from node4
  p2p_node:close_connection(node4, node1),
```

```

timer:sleep(100),
% Try to send data from node1
Reply = p2p_node:send_data(node1, node4, <<">>),
% The connection must have been closed from both sides
?_assertMatch({no_connection, {node1, node4}}, Reply).

```

Listing 5.6: Closing an opened connection

Computing the MST

```

start_mst(Nodes) ->
{timeout, 30,
 fun() ->
    % Make one of the nodes start the computation of the MST
    p2p_node:start_mst_computation(hd(Nodes)),
    % Wait for all the messages to propagate
    timer:sleep(10000),
    % Get the internal state of each peer
    States = lists:map(fun p2p_node:get_state/1, Nodes),
    Info = [FinishedMst || #state{mst_state = FinishedMst} <-
    States],
    Sessions = [SessionID || #state{current_mst_session =
    SessionID} <- States],
    % Check if all the peers have finished computing the MST
    ?assert(lists:all(fun(X) -> X == computed end, Info)),
    % Check that all the MST sessions are equal (no one is
    left behind)
    ?assert(lists:all(fun(X) -> X == hd(Sessions) end,
    Sessions))
    end}.

```

Listing 5.7: Start computing MST

Adding a peer while computing the MST

```

join_during_mst(Nodes) ->
    % A new peer joins the network
    p2p_node:join_network(node500, [{edge, node1, node500, 10}]),
    % Another peer starts the computation before the new one
    p2p_node:start_mst_computation(node1),
    % The new one starts the computation aswell
    p2p_node:start_mst_computation(node500),
    timer:sleep(5000),
    AllNodes = [node500 | Nodes],
    check_mst(AllNodes).

```

Listing 5.8: Peer starts the computation of the MST while another one does it

Removing a peer while computing the MST

```

kill_node_while_computing_mst(Nodes) ->
{timeout, 300,
 fun() ->

```



```

p2p_node:start_mst_computation(node1),
% Make node1 leave the network as soon as possible
p2p_node:leave_network(node1),
timer:sleep(10000),
% Get the state of all the remaining nodes
States = lists:map(fun p2p_node:get_state/1, lists:
subtract(Nodes, [node1])),
Info = [FinishedMst || #state{mst_state = FinishedMst} <-
States],
Sessions = [SessionID || #state{current_mst_session =
SessionID} <- States],
% Check that everyone finished computing the MST and in
in the same session
?assert(lists:all(fun(X) -> X == computed end, Info)),
?assert(lists:all(fun(X) -> X == hd(Sessions) end,
Sessions))
end}.

```

Listing 5.9: Node leaving the network while computing MST

Sending data in an open connection

```

send_data(_) ->
p2p_node:start_mst_computation(node1),
timer:sleep(10000),
% Establish a connection
p2p_node:request_to_communicate(node1, node2, 1),
ToSend = <<"Hello World">>,
% Send some data
p2p_node:send_data(node1, node2, ToSend),
timer:sleep(1000),
% Verify that the data has been actually written on a file
{ok, Read} = file:read_file("node1node2.data"),
file:delete("node1node2.data"),
?_assertEqual(ToSend, Read).

```

Listing 5.10: Send data

Sending data to a failed/unresponsive peer

```

send_data_to_dead_peer(_) ->k
p2p_node:start_mst_computation(node9),
timer:sleep(5000),
% Establish a connection
p2p_node:request_to_communicate(node2, node10, 1),
% The other node leaves the network
p2p_node:leave_network(node10),
% Try to send some data
Reply = p2p_node:send_data(node2, node10, <<"Hello">>),
% An ACK is not expected, sending data shouldn't fail
?_assertMatch(ok, Reply).

```

Listing 5.11: Sending data to a peer that left the network

Request for unavailable bandwidth

```
request_to_communicate_no_band(_Nodes) ->
    p2p_node:start_mst_computation(node1),
    timer:sleep(5000),
    % Ask for an enormous amount of band
    Reply = p2p_node:request_to_communicate(node1, node2, 100000),
    % The request should fail with a {no_band, _} return value
    ?_assertMatch({no_band, {_Hop, _Weight}}, Reply).
```

Listing 5.12: Request to communicate when the bandwidth request is unsatisfiable

Web Interface test

This section shows some screenshots taken during the execution of some tests carried out using the graphical interface.

Figure 5.1 shows what the admin interface looks like after the network has been created. As we can see from the graph, the MST has already been calculated.

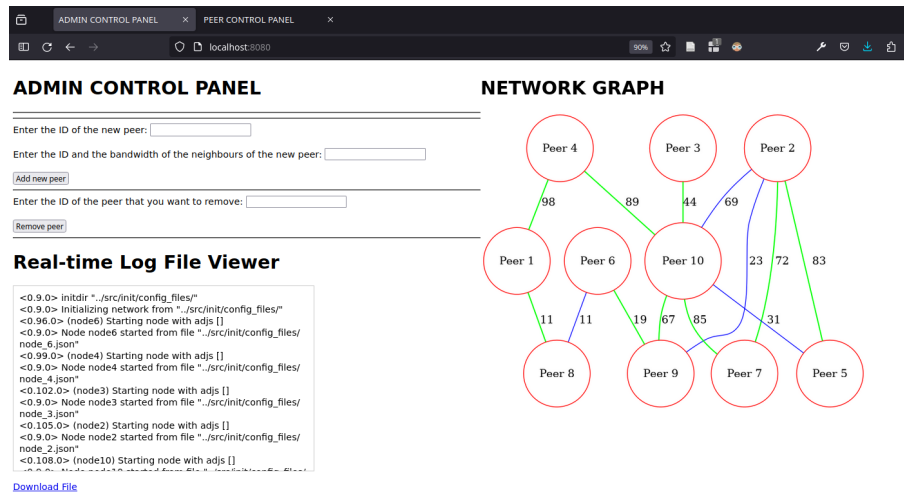


Figure 5.1: Admin Web GUI after the creation of the network

The figures 5.2 and 5.3 show respectively the before and after of adding a peer to the network, using the admin interface.

Figures 5.4 and 5.5 show respectively before and after removing a peer from the network, using the admin interface.

Figures 5.6 and 5.7 show respectively the before and after the request to open a connection between Peer 2 and Peer 11, with bandwidth 2, using the peer interface.

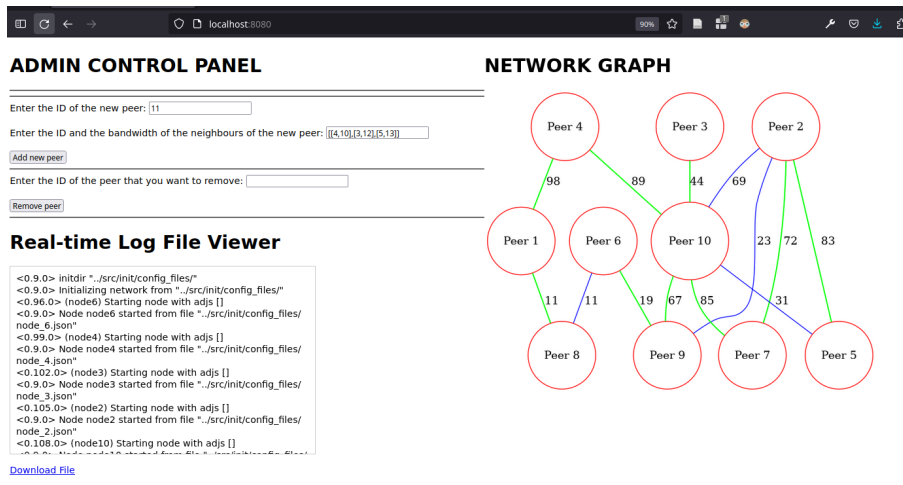


Figure 5.2: Before adding Peer 11

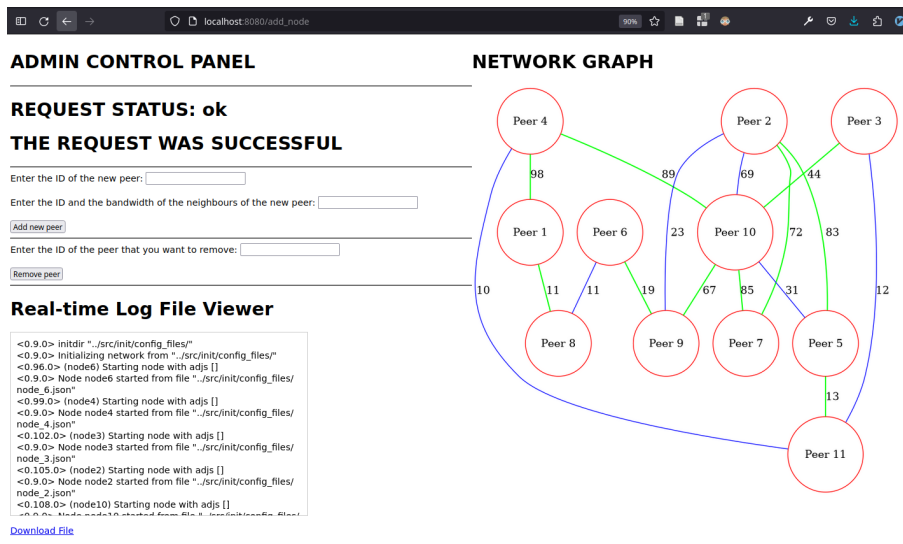


Figure 5.3: After the Peer 11 has been created

Figures 5.8 and 5.9 show respectively the before and after the request to close the connection between Peer 2 and Peer 11, using the peer interface.

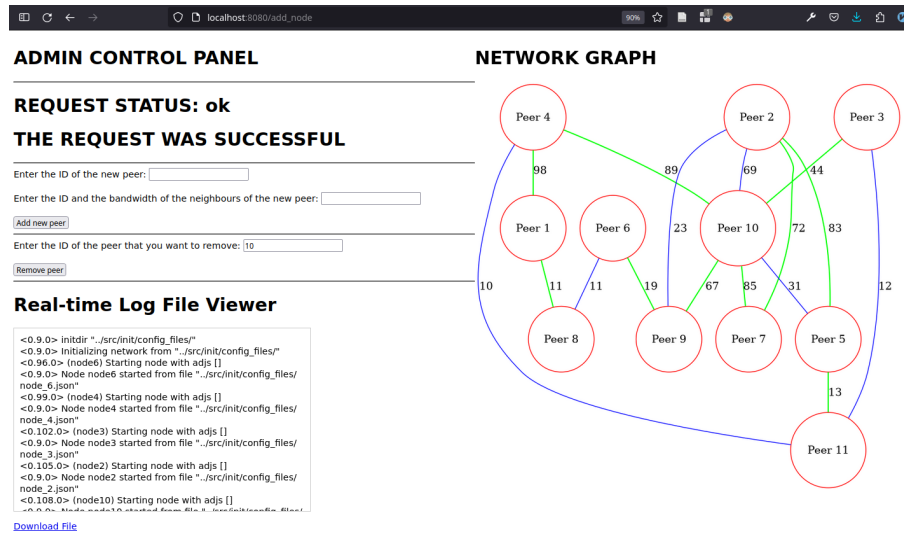


Figure 5.4: Before removing Peer 10

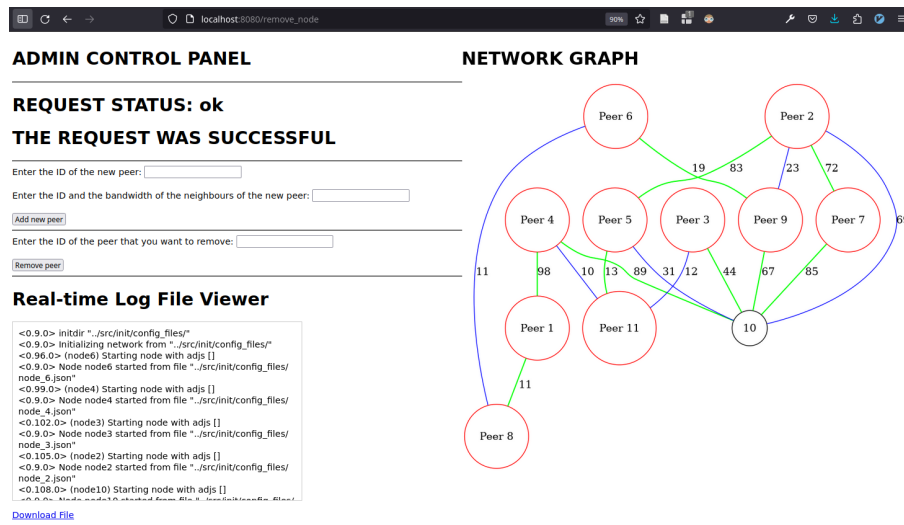
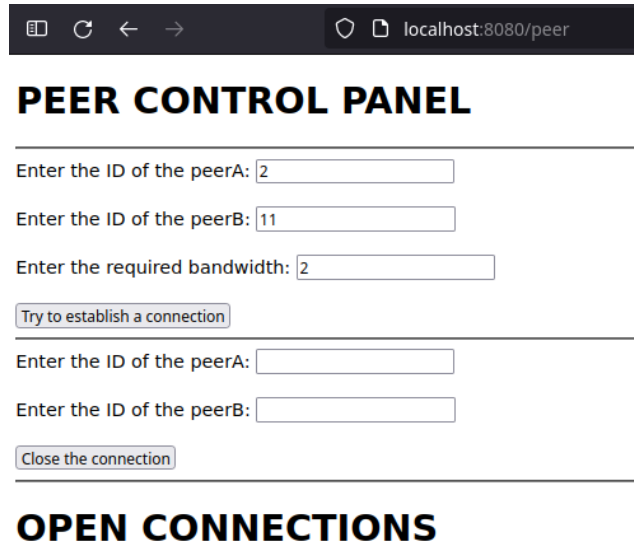
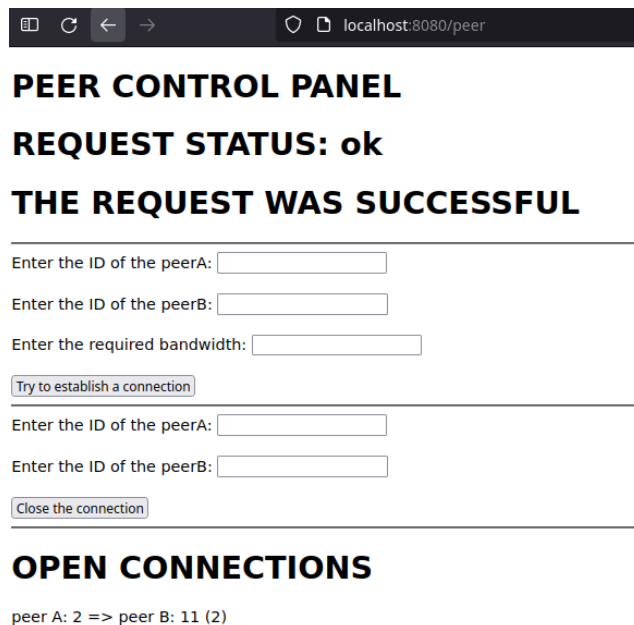


Figure 5.5: After the Peer 10 has been removed



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/peer'. The page has a dark header with navigation icons. The main content area is titled 'PEER CONTROL PANEL' in bold. Below the title, there are three input fields: 'Enter the ID of the peerA:' with the value '2', 'Enter the ID of the peerB:' with the value '11', and 'Enter the required bandwidth:' with the value '2'. A button labeled 'Try to establish a connection' is positioned below these fields. Below the button, there are two more input fields: 'Enter the ID of the peerA:' and 'Enter the ID of the peerB:', both of which are empty. At the bottom of this section is a button labeled 'Close the connection'. The page is divided into sections by horizontal lines.

Figure 5.6: Before requesting to open a connection between the two peers



The screenshot shows the same web browser window as Figure 5.6, but with additional text displayed. Below the 'PEER CONTROL PANEL' title, the text 'REQUEST STATUS: ok' and 'THE REQUEST WAS SUCCESSFUL' is shown in bold. The input fields and buttons remain the same as in Figure 5.6. Below the 'OPEN CONNECTIONS' section, there is a line of text: 'peer A: 2 => peer B: 11 (2)'. The page layout and styling are consistent with the previous figure.

Figure 5.7: After the open connection request has been made

PEER CONTROL PANEL

REQUEST STATUS: ok

THE REQUEST WAS SUCCESSFUL

Enter the ID of the peerA:

Enter the ID of the peerB:

Enter the required bandwidth:

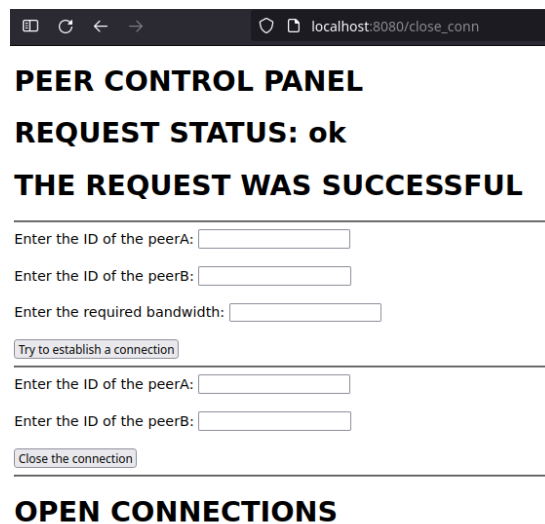
Enter the ID of the peerA:

Enter the ID of the peerB:

OPEN CONNECTIONS

peer A: 2 => peer B: 11 (2)

Figure 5.8: Before requesting to close a connection between the two peers



PEER CONTROL PANEL

REQUEST STATUS: ok

THE REQUEST WAS SUCCESSFUL

Enter the ID of the peerA:

Enter the ID of the peerB:

Enter the required bandwidth:

Enter the ID of the peerA:

Enter the ID of the peerB:

OPEN CONNECTIONS

peer A: 2 => peer B: 11 (2)

Figure 5.9: After the close connection request has been made

Chapter 6

Conclusions

In this project, a first idea of a streaming system was developed, based on a Peer-to-Peer network. The main problem that we wanted to solve was to allow the various peers to communicate with each other using the best channel, i.e. that channel with the highest minimum bandwidth, in order to avoid bottlenecks. This problem, in the graph domain, is known as the Widest Path Problem, which can be solved by calculating the Maximum Spanning Tree. Wanting to develop a distributed application, such as a real streaming system, to solve this problem, we used a slightly modified version of the Gallagher-Humblet-Spira [2] algorithm.

In chapter 2 of this work, the functional and non-functional requirements that we wanted the system to have were drawn up. The mandatory requirements have all been implemented. However, the most advanced features of the system, as described in sections 2.1 and 2.2, are left to future implementations. One thing to underline is that obviously some assumptions were made to simplify the project, such as the non-consumption of bandwidth despite use.

Furthermore, as specified in the introductory phases of Chapter 4, during the design phase, it was decided to implement the system so that it ran on a single Erlang VM. This led to the implementation of some centralized parts of the system, such as the presence of an admin and node manager. However, the code that has been written, as previously mentioned, is already structured to be easily adaptable to production on different Erlang VMs, in such a way as to better simulate a real distributed scenario.

Future works

While most of the requirements have been met, some additional requirements could be implemented.

The first concerns the responsible abandonment of a peer from the network. By responsible we mean the fact that a peer, before leaving the network, must ensure that it is not part of any communication taking place at that moment. In case he was part of it, he would have to wait.

The second is certainly to take into account the residual bandwidth on each link, taking into account the current use of each one. This obviously would also mean that a communication request from one peer to another peer might not be satisfiable at a specific moment, but perhaps in the future. This feature introduces the concept of “not satisfiable at this time” and “never satisfiable”.

The third, deriving from the previous one, concerns the temporal priority of requests. If a peer initiates the communication request before another peer, it should be handled before the other request.

These are just some of the possible future improvements: obviously, if we compare this P2P streaming system with a real one, we easily realize how many improvements could be made (if only we had at least 5 months to do this project).

Appendix A

Appendix

A.1 Building and running the system

To run the system with minimal effort, first install the essential tools which are Erlang, Python, Rebar3 (Erlang build system) and Pip, then install the needed dependencies with `pip install -r src/webgui/requirements.txt`. At this point the wrapper scripts `launch_p2p.sh` and `launch_server.sh` can be executed from two different terminals to start respectively the Erlang backend and the Python web application. The next sections describe in more detail how to build and run the system if needed.

Network initialisation

Before starting the whole system, it's important to generate the initial configuration of the network. To do so, navigate to the `src/init` directory and run `python init.py`, which will create a `config_files/` directory in which there will be the JSON configuration files for the peers. The script accepts two optional arguments:

- `-n [integer > 0]` specifies the number of peers in the network;
- `-e [integer from 1 to 100]` specifies the percentage of edges that should be present in the network graph, with 100% being the number of edges in a complete graph.

By default, a network with 10 peers and 80% of edges is created.

Erlang backend

To run the Erlang application, first install `rebar3` (<https://rebar3.org>) that is needed to download and install all the required libraries and compile the source code. In the `p2p.app` directory, run `rebar3 escriptize`, which will build the application. At this point, the executable can be found in

`p2p_app/_build/default/bin/p2p` and can be directly executed from the command line. The program accepts two arguments:

- `-i [PATH]` which is **mandatory** and is the path of the directory containing the initialisation files. If the previous instructions have been followed, the argument should be `../src/init/config_files/` (if executing from `p2p_app`);
- `-v` to enable verbose logging. The log file is located in the `p2p_app/logs/` directory.

Web application

The web application requires two libraries, Flask and Graphviz. These can be installed with the command `pip install -r src/webgui/requirements.txt`, working from the project's root folder. To start the application, run `python main.py` from the `src/webgui` directory.

MST validation

To validate the MST the script `p2p_app/utils/validateMST.py` can be used. This requires the `networkx` library which can be installed via `pip install networkx`. The script checks the config files of the peers, builds a graph and computes the Maximum Spanning Tree using a traditional algorithm. The output is `True` if the MST computed by the GHS algorithm is one of the possible Maximum Spanning Trees given the network graph.

Bibliography

- [1] Wikipedia contributors. Widest path problem — Wikipedia, the free encyclopedia, 2024.
- [2] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.