Università degli studi di Firenze
(University of Florence)

Master's degree in Software: Science and Technology

Academic Year 2025/2026

Distributed Programming Report

Authors: **Student ID:**

Francesco Biribò 7199598

Contents

1 Decentralized chat system

1.1 Brief description

This projects aims to implement a very simple chat system where users (the *clients*) can send messages between each other both in private chats and in group chats.
The main twist is that the chat *server* is not a single host, but rather a whole distributed system.

The nodes in the distributed system will cooperate with one another both on the Control and data Plane. On the Control Plane, they try to achieve, and possibly maintain, system stability. On the other hand, on the data Plane they try to achieve (almost) fair both workload distribution (to avoid bottlenecks) and data consistency.

These two are obtained by assigning different roles to the nodes in the system:

- Multiple nodes are assigned the *input* role, that is, they become an endpoint of the system, where clients can connect to send their request. This creates multiple flows of messages inside the Topology, avoiding the bottleneck that a single point of entry would create;
- Multiple nodes are assigned the *persistence* role, that is, they are tasked with storing the actual data (users, groups and messages). This, in turn, allows for multiple nodes to handle users' requests, while also guaranteeing that, even if a *persistence* node were to fail, all (or part) of the data would still be accessible;
- Some nodes will be just *forwarding* nodes, that is, their task is to correctly route messages in the Topology. For an example, correctly forwarding a request from an *input* node towards a *persistence* node, and then forwarding the response back to the original *input* node.

The problem of assigning roles to nodes is solved by running an election between the nodes, and then constructing a spanning tree of the network, starting from the election winner, who will be the root. The nodes understand their role based on the spanning tree (explained in details later).

As a final touch, since the distributed system should appear a single chat server to clients, there is a central name server where the *input* nodes register themselves to. Users can connect to the name server and get redirected to any of the currently active endpoints.

1.2 Main goals

The primary objective of this project is to bridge the gap between theoretical foundations and practical application within distributed environments. Specifically, this project aims to

explore how fundamental algorithms such as leader election and spanning tree construction, even simple ones like *Yo-Yo* and *SHOUT*, operate in practice, addressing the need, and complexity, of synchronization between nodes.

Furthermore, while the front-end presents a very simple chat interface with users, groups and messages that support only C_RD operations (not updates), the actual goals of the project were to address how a decentralized system can achieve transparency, appearing as a single centralized server.

2 Design & Architecture

2.1 Architecture overview

The system architecture is designed arount principles of Separation of Concerns and defining a clear distinction between the Control Plane and Data Plane. This choice ensures that network setup and maintainance logic, as well as the whole network management traffic is separated from application data flow, in order to keep the system both maintainable and resilient.
Furthermore, the system presents a modular, and even reactive, behaviour: while the Control Plane layers are continuously active, working from the start of the system, all the other modules follow an "On-Deman" activation approach:

- The Data Plane layer is activated only after the system is in a stable configuration (as seen later, that means completing an election);
- The Input/Presentation layer is activated only for *Input* nodes;
- The Storage/Persistence layer is activated only for *Persistence* nodes.

The reason is that these three layers require proper configuration before starting. Such configuration is determined after an election has terminated, and each node retrieves its roles.
In the end, to achieve a compromise between stability and persistence, a single writer approach was taken when assigning roles to the nodes.

2.1.1 Minimal node configuration

Each node requires three things minimum to be started, these are: a unique ID, a port used for Control Plane communication and one for Data Plane communication. The need of an ID is to make elections possible (since algorithms require that each node should be uniquely identified, for this application a simple `uint64` was used).
To actually communicate with other nodes, additional information is needed, but that will be explained later.

2.1.2 Shared definitions

Before continuing, it is important to introduce some of the definitions used in the project, to have a better understanding of some concepts or names used.
While the ID of a node is a `uint64`, it has been defined as `type NodeId uint64`. Also, in order to avoid collisions with possible already used port, the valid port range is defined as `[46000-65535]`, to make sure ephemeral ports are used.
Later, the term 'Identifier' will be used when referring to inter-nodes connection. That is because `ZeroMQ ROUTER` sockets require an unique identity (to know where to send data, since multiple connections are multiplexed over the same socket). Since Node IDs are already unique, the *identifier* of a node (with id `<%d>`): `'node-<%d>'`

2.1.3 Custom logging

Since there are multiple layers and components running simultaneously, which operate at diffent levels, a single logger object was not enough, there was the need of multiple loggers, to end up with multiple log files. Hence a NodeLogger struct was implemented, this helps by abstracting multiple loggers behind a single one, identifying each one with a string. In order to share this object with the various components, while avoiding them writing on a wrong file, a node.Logger interface was defined. This interface is implemented by SubsystemLogger, which basically stores one single string, and can only use the original NodeLogger on that corresponding file (limiting a subsystem to its dedicated file, while the ClusterNode has still access to each one).

2.2 Control Plane

The Control Plane Manager is responsible for almost everything in the Cluster Node. It handles the connection between the neighbors in the Topology. And most importantly, it handles both the election process, as well as the spanning tree construction.
The last thing it does after the system is in a stable, post election, state is to inform the Cluster Node (it's container) of the configuration of the nodes for the other modules, telling what modules to run and with what permissions and configurations.

A principle of composition and delegation was applied when designing the Control Plane manager, hence the following components:

- A Topology Manager, which is in charge of handling neighbor connections. It knows who the neighbors are and how to reach them. Is also can communicate with them;
- A Tree Manager, which is in charge of storing the spanning tree relationship between nodes after an election. That is, if the node is root, who the parent node is, if there is one, and who the children nodes are;

- A Heartbeat handle, which is just a goroutine that sends messages periodically to neighbors, to ensure they know that the sending node is alive (more on this later).

More in general, the Control Plane Manager is a mini-orchestrator for the Control Plane. It also uses the Topology Manager to send messages to other neighbors, both to handle the election and spanning tree construction. It basically uses these components to achieve topology stability (with each node knowing its role).

2.2.1 Topology Manager

The Topology Manager is responsible for handling the local view of the node inside the Cluster's network. It abstracts the Topology and neighbor communication for the Control Plane Manager.
One of the main ideas for this project was that, even though nodes are scattered over the internet, each nodes believes to be in a LAN with only its neighbors (each node doesn't know anyone else outside their neighbors, as if they were connected by a physical cable, creating an actual overlay network over the physical internet).

For this reason, this component handles the Topology view, ensuring that each node can only send messages to its neighbors, returning an error when trying to retrieve information about a non neighbor, or sending a message to one.
It knows which nodes are neighbors by using a map of `NodeIds` to `Address` (which is just a pair of hostname and port), which maps each neighbor's ID to his IP and port used for the Control Plane.
However, the actual communication is delegated to a Connection Manager, which has actual methods to connect to another node, send messages to and receive messages from it.

Finally, Topology Manager handles the state of neighbors, by keeping track of the last time we received a message from them. After the last seen time of a certain neighbor goes over a time threshold, the node is considered OFF.

2.2.2 Connection Manager

The Connection Manager is the node's component that handles communication with other nodes in the system. Any message that is sent or received is done so by this component. It is defined as an interface to ensure interchangeability and agnosticism to the actual implementation.

For the scope of this application, this Manager was implemented using `ZeroMQ` sockets, in particular, `ROUTER`-type sockets in a `ROUTER-ROUTER` pair setup. The reason is that `ROUTER` sockets allow bidirectional communication, and they require an unique identity. That is because `ZeroMQ` handles multiple connections over a single socket, by multiplexing using the socket's ID. Since each node already had an unique ID, the same was used for the socket's

identity (As a matter of fact, the ID used was the node's Identity, which is just `node-<d>`).

Another reason that lead to choosing `ZeroMQ` is that it automatically tries to (re)connect to a peer, that means that even if a node is faster than another, the connection message will not get lost, and arrive when the other node opens, and binds, his socket[1].
Furthermore, the identity of a socket is sent with the payload, as first frame (zeromq divides a message into more frames), meaning that on the receiving hand, a node can actually extract the ID of the sender (because it uses its ID as socket identity), simplifying the routing of election messages, as well as understanding who sent the message.
Finally, the socket is set as non blocking, and a Poll method is defined. This was mainly to ensure that timeouts (for example during election) are correctly noticed, and handled.

2.2.3 Communication protocol

Inter-node communication relies on a strictly typed messaging protocol that is designed with reusability in mind, and for that reason, it supports polymorphism. To ensure flexibility and type safety, a generic Message interface is defined, which is implemented by the various types of messages.
Each message contains a message header that carries the essential information for the message to actually get routed correctly, which are:

- Source ID, the ID of the sender node (even though `ZeroMQ` adds the socket identity in the frame, this cannot be taken for granted. If `ZeroMQ` were to be swapped with another framework we would not have this convenience);
- Destination ID, the ID of the destination node (used by such nodes to check if a message is truly for them);
- Message Scope, which defines what component is interested in this message, it is an enumarator (`Tree`, `Topology`, `Election`, `Data`, `Heartbeat`);
- Timestamp, value of the logical clock of the node, used to understand the *happened-before* relation between messages.

Each type of message adds its own necessary fields, and each one will be briefly explained in the corresponding section.

[1]Usually, just one peer binds the socket, and the other connets, however, since there is no deterministical order in this scenario, each node both uses `bind`, and `connect`. To avoid opening two different connections, the `ZMQ_ROUTER_HANDOVER` is set to true. This ensures that if `A` binds and `B` connects (a connection is set) and later, `B` binds and `A` connects, ZeroMQ will reuse the same, existing, connection.

2.2.4 Dispatcher goroutine

To ensure thread safety and prevent race conditions on the underlying socket (which is not thread-safe by design), the system adopts a Single-Reader pattern. The `RunInputDispatcher` method operates within a dedicated goroutine that holds exclusive read access to the network resource. This component functions as a Demultiplexer: it continuously waits on the socket, deserializing incoming payloads, and insping the message Scope within the message header. Based on the message scope, it dispatches the data to specific buffered channels. This design effectively decouples the physical packet reception from its logical processing, allowing the Election and Topology modules to consume messages asynchronously without blocking network traffic.

2.2.5 Synchronization mechanisms

An integral part of this project was to handle concurrency, both between nodes in the network, and between goroutines inside a single node.
With regards to single-node concurrency, it was handled by guarding critical resources or code portions with locks, through both the `sync.Mutex` and `sync.RWMutex` objects.

On the other hand, to address multi-node concurrency, a simple logical lamport clock was added on each node.
The Clock is just an `uint64` counter, guarded by a `RWMutex` (since multiple goroutines access it). The clock ensures that nodes are aware of the *happened-before* relation between messages that are received.
Total ordering (e.g. using Vector clocks) would have been an unsustainable solution, since we would need to both keep a counter for each node, and also send our clock with each message. Considering the small, but consistent traffic that will be generated, that solution have impacted performace and latency. As a bonus, for both election and spanning tree construction algorithms an *happened-before* relation is sufficient, as election algorithms use the ID of the node as discriminant, and spanning tree algorithms terminate under graph connectivity and message causality.
For this reason a scalar logical clock was used. This clock is also used to mark each outgoing message. Specifically, before delegating the message to the Topology Manager, the clock is updated and the new value is inserted in the message's header. When receiving a message, from the Topology Manager, a node extracts the clock timestamp from the message header, and updates its own clock using lamport's rule, $ReceiverClock = max\{ReceiverClock, SenderClock\} + 1$.

2.2.6 Election

Election was one of the primary goals for this project and it was designed to happen very early in the topology, as a matter of fact, each time a new node enters the topology, a new election is started. However, this only happens if there is not a current leader in

place. In a current Leader is present, an entering node would be greeted instantly with a LeaderAnnouncement Message. On one hand, this would make the topology *stale*, not changing leader frequently, even if the entering nodes would be leaders, by criteria (e.g. smaller ID). The reason is that since the system aims at stability and data persistence, changing continuosly the Data Plane, HTTP servers and Databases just becase one node entered the topology would have been detrimental for performance and data consistency.
It is far more stable to add the neighbor to the existign topology and tree.

A new election is started only when the current leader is unreachable, or, as previously stated, each time a new node enters the topology, if a leader is not present.

Control Plane Manager uses an Election Context to handle the election state and rounds. This component is an ephemeral data structure that is only created during an election cycle. Its main goal is to completely encapsulate the election process, separating the volatility of the election from the from the stable runtime environment of the Cluster Node.
Election Context is also used to contain all the critical data regarding the Yo-Yo algorithm, such as topological state (referred to as status in the code), that being Source, Sink or Internal Node; it also maintains the state for a FSM that is used to transition to specific phases of the algorithm (YO-DOWN, YO-UP and IDLE). Moreover, it handles links orientation between neighbors, by using a map of `NodeId` to `Orientation` (defined as bool, since there are only two orientations. incoming and outgoing.)
Furthermore, to ensure consistency in a distributed, asynchronous environment, an Election Context also uses a Round, which contains all the necessary information for one round of the election cycle. These information include a map of votes, proposals and an epoch counter, to trace the current round of the election.

The type Election Message is a message designed for this scope. If defines different flags that are used to determine what type of election message it is: *Start*, *Proposal*, *Vote*, *Leader*. Each election message also carries the Election Id, to ensure nodes participate on the same election, and a round epoch, to make sure the nodes are also aware of what round the election is ON.

The election starts (locally, in the node) after joining a topology and not receiving any leader information. It starts by sending a "START" message, also generating an election ID to ensure that the election messages are for the correct election. After sending start, it enters YoDown, after calculating its status (it's calculated based on the links orientation).
First of all, nodes have to agree to an election ID, that is, following the same election. Each node, by default, even if it had already started yo down, switches election and restarts when it receives a stronger START (a start with a stronger ID). The rule to determine what id is stronger is: *biggest clock-smallest-id*. Each time a new, stronger, election arrives, a node restarts and switches to the new one.

Ideally, after agreeing upon an election ID, the election should carry out as normal:

- Nodes start in Yo-Down, where Sources send proposals, while Internal nodes and sinks wait;
- After yo-down, Sources wait for yo-up, while internal nodes, compute the smallest ID, then forward it to the Sinks. The sinks compute the smallest ID, and compute the votes, starting yo-up;
- During Yo-Up sinks send votes to the internal nodes. Both the latter and sources wait to receive all the votes;
- When the internal nodes receive all the votes, they can send them to the sources.

This completes a single election round, which ends with the nodes inverting the links where a negative vote was received or sent, and pruning the links that asked to. To ensure that the Yo-Yo algorithm actually converged, it is implemented with pruning.

The main problem is that messages can arrive out of order, for example for previous elections, newer elections or for the correct election but for a different round, to correctly handling all cases the following rules were defined:

- A message for an older election ID is ignored, as it is old;
- A message for a newer election ID always resets the node and makes it forward START to each neighbor;
- A message for the current election ID is correctly registered, but then a check on the round is run.
- A message for the current election ID and an older round is ignored;
- A message for the current election ID and an newer round is stashed in a map, and automatically processed when getting to that round;
- A message for the current election ID and the current round is correctly processed with for the Yo-Yo algorithm;

Furthermore, two other problems became evident, since the original FSM did not cover all cases:

- Any start message, independently from the state, is handled the same;
- A leader message is responded with either a leader response, or a stronger leader message (if some node calculated a different ID somehow, we need to tell him);
- Any proposal or vote received in idle or in the opposite phase are ignored.

Finally, there was the need to implement timeout handling since node could fail anytime, and failing during the election should not block the other nodes' election. They should just discard of the OFF node and continue. To achieve this, the goroutine that is tasked with handling election messages, does a periodic timeout check, every 15 seconds or so. During each timeout, each node checks the status of all neighbors (using Topology Manager's last seen time).
Each neighbor that is OFF will be marked as such and will not be counted towards the next election. Unfortunately, the current one needs to be restarted, to ensure that the removal of a neighbor does not interfere with the outcome. During the next election the node will not be considered. However, if it happened to come back on, he would either be greeted with a LeaderResponse, if a leader was elected in the meantime, or would start a new election, participating (possibly) correctly.

The election ends when the last source goes to update its role, realizing it is alone, and claiming the Leader Role, starting a SHOUT to announce it while also building a spanning tree. Moreover, it stores the election result in a stable Runtime Context structure.
Each node that receives an election message will enter a routine of setting up the Runtime Context and respond to the spanning tree constructrion messages.

2.2.7 Spanning tree construction

The spanning tree construction is a process that is dealt by two components together, the Election Context with the Tree Manager. The latter is responsible for handling the links between tree neighbors during the spanning tree construction.

It does not handle communication by its own, which is still done by the Topology Manager. The tree manager just keeps track of the links in the tree and also tries to heal the tree when a node goes OFF.
The spanning tree construction starts as soon as the election is ending. When the last source realizes it has won the election, instead of just flooding, it announces it using the protocol *SHOUT*. Since the election is not yet complete, Election Messages are still used, as Tree Messages are designed to be used when the tree has already stabilized.

SHOUT was implemented as theory defines it: each node has a TreeState, either Done, Active or Idle, and they exchange Question and Answer(Y/N) messages.
The two types of messages used are *LeaderAnnouncement*, which serves as **Q**, and *LeaderResponse*, which serves as **A**. Initially, the Leader sends the LeaderAnnouncement message to all of its neighbors, switching to Active state (waiting for responses).

All other nodes initially are Idle, and when they receive a *LeaderAnnouncement* message, they send back a *LeaderResponse(Yes)*, to notifiy the neighbor of two things: having acknowledged the leader, and agreeing to become its tree child.

The node that sent a Yes also added the original sender as Parent in the tree, sends *LeaderAnnouncement* to all its neighbors, and becomes Active. Any Active node that receives a question answers with *LeaderResponse(No)*, since it's either Root or already has a parent. Every Active node that receives a *LeaderResponse*, can mark the sender adds the sender as tree child if it responded with YES.
In each phase (Idle or Active), each *LeaderResponse* (and the *LeaderAnnouncement* that the we responded Yes to) increments a counter by 1. When the counter reaches the number of neighbors, *SHOUT* is completed locally, since the node got an answer by everyone he knows.

After this step, the root starts two processes, at first it sends down the tree (to all children) its port for the Data Plane. The other process is an incremental counter that is explained later. For both processes, the actual Tree Message type is used.
The Tree Message type is pretty straightforward. It just contains an epoch field, for a counter and a flag field. Flags are used to determine the meaning of the message.

2.2.8 Tree healing

The second process started by root is an incremental epoch counter. Basically, at a regular interval, the root send to all children an `EPOCH` Tree Message (I omitted the `TFlags_` prefix), with the current counter, then increments it. Each node that receives it updates the current local counter, and forwards it to its children. This creates a flow of messages, that start from the root towards the leaves, which contain a counter that is used to tell how much old the tree is.
The Tree Manager tries to heal the tree. It tries to do so by checking parent timeout and, when a node realizes that a tree parent is off, they send all non-tree neighbors (to avoid a cycle) a Tree Message with the flag `NEPARENTREQ` (asking to join the other neighbor's subtree). The node does so in order to connect back to the root.
And here, the epoch counter comes in play. Since it's a message started from the root, it can be used when responding to a node asking for help. Reattaching a subtree to the tree is treated as part (a very, later part) of the *SHOUT* protocol. Basically, the node sending the `NOPARENTREQ` is basically sending a reverse **Q** to its neighbors (asking them to send a *SHOUT* **Q**).

If a node recieves a `NOPARENTREQ` message, it checks the epoch counter. If the local one is smaller, it implies that either this node is disconnected from root as well, or it could be a descendant of the sender (helping that create a cycle), so it responds with a `NOPARENTREP` (which, by itself, means that a node cannot help the sender). On the other hand, a node that has a greater epoch counter responds with a `NOPARENTREP` plus the `Q` flag, effectively sending *SHOUT*'s **Q** (going back to Active state in the meantime).
When the node asking for help receives a `NOPARENTREP` plus `Q`, it understands that sender node can help, and since multiple nodes could answer positevely, the following criteria is used: the first one is instantly accepted by sending an `A(yes)` Tree Message, while

the following ones are responded with `A(no)`. These messages act the same way as the `LeaderAnnouncement/Response` ones, the logic is the same (it could be seen as a partial SHOUT) for the other *SHOUT*.

While it could be true that nodes with better routes answer with Q, it's not guaranteed that any node responds at all, hence the instant accept.

2.2.9 Roles calculation

After the construction of the spanning tree is complete, each node should be assigned a role for the data Plane.
Who is the one assigning the node its role? It's the node itself. Each understands its role by looking at the local tree topology, following these rules:

- The Root (Leader) becomes the single writer in the system.
- The leaves become Input nodes;
- Children of root become Persistence replicas (reader-only), if they are not already classified as Input.

After each node determined its role, it just needs to configure the last steps before starting the data Plane.
First of all, commucation over the data Plane will be based on the spanning tree structure, effectively creating an overlay network over the original topology, as a matter of fact, different ports will be used for the Data Plane, to ensure that Control Plane traffic is isolated from Data Plane traffic.

The root starts by sending its port for the Data Plane to its children. This will, in turn, start a process of registering the parent's Data Plane port and forwarding ours to the children.
The children, on the other hand, will send their parent next hop information, starting from the leaves and going up. The reason is that the routes of requests from the input nodes to the persistence nodes are well known, each node just needs to forward to the parent in the spanning tree.
However, going back to the original node is not as immediate. For this reason, each node with a parent, forwards next hop logic to it.

For example consider a tree, where 1 is root, 2 and 3 are 1's children, and 4 and 5 are 3's children:

- 4 will tell 3 that 4 is reachable through 4, 5 does the same;
- 3 will tell 1 that 4 is reachable through itself(3), same goes for 5;
- Also 2 will tell 1 that 2 is reachable through itself;

- In the end, the next hop maps will be 3 with [4:4, 5:5] and 1 with [4:3, 5:3, 2:2].

The actual final step, is to send the Data Plane configuration up to the Cluster Node. This is simply done by injecting a channel from the Cluster Node into the Control Plane Manager, and start waiting on it.
The data is sent after each node forwarded their port to their children. While it's true that all nodes could not be entirely finished, at this point they just need to update their next hop map; it can also be done later, as the routing map is stored inside the Runtime Context, which is shared as a pointer. The underlying structure is guarder by a `RWMutex` to ensure thread safety, since it's shared.

2.2.10 Three way handshake

Topology Message is a type of message used by nodes when connecting to a neighbor. They implement a three-way handshake. Supposing sender S and receiver R:

- S sends a Topology Message with flag `JOIN` to R and sets R as joinack-pending (S is waiting for R to answer the `JOIN-ACK`);
- R receives and sends S a Topology Message with `JOIN-ACK`, setting S as ack-pending (R knows that if S sends `ACK`, they are neighbors);
- S receives the `JOIN-ACK`, so it can remove R from the pending list, actually mark it as neighbor and send R an `ACK`;
- R just receives the `ACK`, then removes S from the pending list, and mark S as neighbor.

`JOIN` Topology messages are sent by the Control Plane Manager. This utilizes an exponential backoff when retrying failed connections.
Briefly, each node, after discovering their neighbors, tries to connect to them, however, they could be too fast and the underlying network socket could not be ready yet.
To handle this data race, a small timer waits before the first send. However, this does not ensure that the connection on the other hand receives the `JOIN`[2]. For this reason each node sends `JOINs` with a time interval in between that doubles each time, up to a max of 16 seconds.
If the neighbor were to come ON in the meantime, this process is stopped. Topology Messages also serve the purpose of exchanging IP addresses, in order to enable the connection Managers to talk to one another.

[2]Even though `ZeroMQ` uses TCP as transform protocol, sometimes the `JOIN` messages got lost, especially if both nodes turned on and connected to one another at the same time.

2.2.11 Heartbeats

Since the Topology Manager keeps track of alive neighbors by checking the last time they sent a message, there was the need of a mechanism to ensure that neighbors know that their peers are one even when not exchanging messages (for example after an election the topoly stabilized, and no one sent messages). This was solved by introducing Heartbeat Messages. This kind of message does not have a body, but only an header, since they just need to update the last seen time.

Furthermore, any message is actually used as an Heartbeat Message, that is, even a non Heartbeat scope messabe updates the last seen time. The Heartbeat Message is specifically for when other kinds of messages are not being sent.

An interesting scenario is when a node turns OFF and comes back ON while not remembering its neighbors. This was fixed in two different ways, the first one is by using a bootstrap server, which will tell the node who their neighbors are. The other one is to use a self detection mechanism, this is based on the idea that heartbeat messages are sent also to OFF neighbors using an optimistic approach (the node could turn back ON and receive them). This means that if a node does not remember his neighbors he would see heartbeat messages coming from a stranger. Nodes try, after a certain number of Heartbeat Messages from strangers, to send a `REJOIN` message, with the intent of entering back in the Topology.

`REJOIN` is just a flag used for Topology Messages, that is added to `JOIN`, `JOIN` and `JOIN-ACK` Messages to mark a three-way handshake between nodes that could be old neighbors. Ideally, a three-way handshake with rejoin ends by just joining back the Topology, without starting a new election, but rather, just receiving the current leader. If an election were currently running, the newly entered node, would see election messages for an election he doesn't recognize (as it is too new) and would try to start a new election.
On the other hand, if an election had already taken place, and the system is in a stable state, the newly entered node would just be notified of the current leader. This choice come from the fact that after a new election, nodes could change role (persistence nodes could become input and vice versa). So from a stabiliy viewpoint, this was the better approach.

2.3 Data Plane

The Data Plane consists of those components that handle Data Messages, which are messages from input nodes to persistence ones and vice versa. The DataPlaneManager is agnostic towards the Control Plane, and is only interested in 'Data' Scope messages. Before starting, it needs to be configured appropriately with permissions, especially for the persistence (R/RW). This also manager implements the Forwarder interface and is responsible for forwarding the request to the appropriare Reader, or Writer.
The reason that Data Plana Manager implements Forwarder, is that requests and replies

reguarding user-server interaction are Data Messages, so a responsibility of the Data Plane Manager.

All application-level traffic is orchestrated by the DataPlaneManager (being exchanged in form of Data scope Messages). This manager has its own instance of Connection Manager, being a different one than the one used by the Control Plane. This prevented Head-of-Line blocking, ensuring that large data transfers don't disrupt the Control Plane's Messages, specifically Election and Tree messages. In order to follow the component's "On-Demand" activation nature, also the Connection Manager utilizes a Lazy approach when sending messages. That is, the connections between tree parent (upstream) and children (downstreams) are closed by default, and are only opened for the first request, remaining open after that. Unlike the Control Plane Manager, who computes the topology, the Data Plane Manager acts as a consumer, relying on the Spanning Tree structure built buy the Control Plane Manager after the election, using the Tree Manager.
The Spanning tree represents an overlay network on top of the classic, Control Plane, topology, where communication happens using a next hop strategy, sending requests Upstream, from the leaves to the root, and responses Downstream, from root to a leaf, using the the correct next hops.
The Data Plane primarily forwards the requests from the input nodes to the persistence roles. These requests come from the Service layer. However, the main problem was that the messages, once they came down from the Service layer, to the Data Plane layer, were not able to make it back up to the service layer, rendering it impossible to complete the request. To solve this problem a mechanism to register callbacks inside the Data Plane Manager was developed. Since Cluster Node has access to all components, being the orchestrator. It is able to execute the Data Plane's logic of unmarshaling the request, executing the service, and retriving the response.
Even though this design is rigid, repetitive and hard to maintain, it was one of the simplest solutions that came to mind. The cluster node registered a callback for all 14 types of data message, inside the Data Plane Manager, which is actually the one who calls the callbacks.
The nodes use the following criteria to forward the requests, considering that the logical network of the Data Plane is the spanning tree constructed in the Control Plane:

- Requests come from input nodes and are headed towards persistence nodes, meaning that requests are send Upstream, to the tree parent;
- Responses have to go from persistence nodes to the original input node. This is achived thanks to the next hop map what was build after the spanning tree construction.

An interesting design choice is that read-only nodes can choose to intercept requests and handle them instead of forwarding to the only writer. This however, can only happen for READ messages that are consistent with the reader. This topic is explained later, in entities, but to put it briefly, since there are multiple databases, the writer updates a counter, called epoch, each time some data is changed on the DB, allowing the partial ordering requests in

time (each record has also store the epoch the db was when it was created).

When a reader receives a request for the leader, and it is a READ operation, it checks if it consistent enough. If the local DB has an epoch X, it means that the data inside the DB is at most, consistent with the X-th write; if the request has an epoch Y, the following can happed:

- $Y \leq X$, meaning the request is up to older writes, meaning it can be handled;
- $Y > X$, the requests is for an epoch greater, the local DB cannot possibly have the requested data, forwarding up.

However, this also requires that the writer announces the updates, so that the replicas can update themselves. This is done using a Data Message, of type SyncOne, which forwards the response of the request to the persistence node, basically asking it to replicate the operation. This is done after every write operation in the writer. If the system were to sustain heavy traffic, this would probably cause performance of latency issues.

The final dilemma about persistence syncrhonization was the following: what happens when an election terminates, the leader already has a DB of epoch ¿ 0, and the new persistence nodes have it empty? They forward the requests, until a WRITE arrives, this makes the leader send a Sync, the persistence node replicates the update, updating its epoch to the new value, and now this it is up to date.

This synchronization problem has been challenging, but in the end, a non, properly, scalable approach was taken. After the roles are self assigned, the writer sends a whole copy of the database, using JSON and a Data Message of type SyncAll. While it is not the most performant solution, it ensured, most of the time, that persistence nodes are up to date, and then correctly replicate new updates.

2.3.1 Input layer

The input layer is managed by the Input Manager, which is tasked to handle client interactions via an HTTP server, allowing it to interact with the chat server. The HTTP server side of the project was designed for an hybrid webpage/REST usage. In other words, some requests return HTML (text), for a browser to render, while some others return JSON data. The reason is that some requests are just sent via a browser, with either a `GET` or `POST` request, and needed to show a different page, returning HTML with `http.Redirect` or rendering a template. On the other hand, some requests used `DELETE`, unfortunately some browsers could ignore `DELETE` if it was used as method in a standard HTML form, replacing it with `GET`. Because of this, the `DELETE` requests were performed with a Javascript function that sent the delete request and awaited for the response; for these requests, the return type was JSON.

On a second thougth, the chat server could be seen as an hybrid between a web app and a REST api, by mixing API endpoints and View endpoints.

2.3.2 Web chat request handling

Apart from the presentation, UI, part, HTTP requests on the Input nodes are handled with Gorillamux's router, since they offer routing also for paths with variables.
Each request is handler by an object called Handler. Specifically, there are three handlers, one for each main entity. As a matter of fact, request handling is implemented using the Service pattern, by having an Handler that retrieves the necessary parameters from the HTTP request, and delegating the actual work to the Service.
Before actually reaching the handler, every request passes through two middlewares, apart from the authentication ones, which just pass through one. The two middleware have different purpose, one to protect the server from unauthenticated users, the other,to protect the server while not entirely ready.

- The first middleware each request goes through is a Pause Middleware. Basically, each input node has a flag `paused` (`atomic.Bool`) that can be used to block request handling. This middleware just checks the flag, and if set, returns a `ServiceUnavaiable` http error. This ensures that no request is received if, for example, the underlaying services are not ready;

- The second one, only for post-authentication routes, is an Authentication Middleware. Instead of needing to login each request, the login handler stores a session cookie on the client's browser, so that following requests are automatically authenticated.

The service, on the other hand, maneges the business logic, it receives the data from the handler, runs necessary checks, and asks a repository to store or retrieve data.
The repository was used to abstract the database. As a matter of fact, method could be used to store data, as long as it offers an interface compliant with the repository's. Its task is to access the persistence storage to retrieve and store data. For the scope of this application, Repositories were implemented using SQLite, thanks to its zero-configuration nature, since nodes could gain, and lose, the persistence role multiple times during their lives.

2.3.3 Service layer

The main problem with a standard layered architecture, was that input nodes, don't have a database, so the Services cannot just delegate information storing and retrieval to repositories, they need to forward the request to a persistence node, which can actually handle it.
For this reason, the Services were designed as interfaces, which abstract the forwarding logic from the Handler.

Two implementations of each service was realised, a Local one and a Proxy one. A Local service has access to repotitories, and it is the one used by persistence nodes.
The other impelementation, Proxy Service, is the one used by input nodes, which just have an instance of a forwarder interface. As explained in the following subsubsection, a forwarder's job is to get a request from the handler, forward it towards a persistence node, wait for the response, and give the result to the handler, as if the procedure was all local.

Furthermore, the Proxy Services use JSON to both marshal and unmarshal request parameters before delegating it to the forwarder (which creates an appropriare Data Message before sending it into the Data Plane).

2.3.4 Forwarder

The idea of abstracting a Data (or Request) Forwarder came from the fact that input nodes are guaranteed to not be persistence nodes. This implies that the usuale Services that just take and store data using the repository could not be applied.
Since the Service layer is the one managing business logic, it was choosen to be the layer that would handle the request by forwarding it to nodes that actually have a repository, and collecting the response.

Briefly, for an input node the HTTP request handling workflow constists of retrieving the parameters using in the Handler, which calls the appropriate Service function.
The input nodes use the Proxy Services implementation, meaning that when they get a request, they marshal it and passing it to the Forwarder, waiting for its response.
The forwarder has a single ExecuteRemote function. A concrete implementation is done by the Data Plane Manager, which alredy has all the connection infrastructure ready. Once the forwarder returns a response, the Proxy Service extracts the data to return and gives it to the handler as if it executed it locally.

2.3.5 Persistence layer

The persistence layer is really just a nice wrapper to keep all the repositories togeher. Internally, when created, either reads the system epoch, or created the record if none is present. Also this object holds a copy of the epoch in memory, so that each time it is requested, there is no need for a separate DB query.

2.3.6 Entities

The modelled entities for this project are very simple and not big in numbers. The are explained more carefully in the documentation. There are just the following:

- **User** just have a UUID, and also an User, Tag combination that is unique (old-Discord style), a CreatedAt, DeletedAt time and a list of group chats.

- **UserSecret** contains the hashes of the users, to avoid storing the hashes in the same table as the users;
- **Group chats** just have a UUID, a name, CreatedAt, DeletedAt time, and a list of members;
- **Message** have a UUID, a Chat identifier, the content, CreatedAt time, and the UUIDs of sender and receiver.
- **SystemState**: This

Also, all entities (but secret) have an epoch field, which is a timestamp of when the record was last modified. The CurrentEpoch in system state is actually the number of writes to the database, that is, operations that changed the database.
The epochs are used in almost every remote request (by proxy services) to ensure a consistent data retrieval. The reason is that persistence nodes work by intercepting requests before sending them to the leader, only if the request is a read.
However, this posed the question, how can the replicas be consistent, because if a replica is not consistent with the writer, all the input nodes that would reach the leader through this replica would just (partial) fail.
Epochs serve this purpose, since they are the number of writes in the database, they are used by persistence nodes to understand if the request they are looking at can be intercepted or not:

- A request that has an epoch between (0, localEpoch], implies that the request is surely for data that has been created before the localEpoch, so the persistence node can safely respond;
- A request with epoch bigger, means it is for newer data, that the persitence node still does not have, so it forwards it to the leader;
- Epoch 0 is tricky, because it means that the client is new, and has no knowledge of the type of the database. This request is always forwarded.

Clients store the lastest epoch in the session cookie, so when they make a request the handler extracts such epoch and forwards it with the request.

2.4 Cluster Node

The Cluster Node is the orchestrator of the various mini-orchestrators in the system. It is the one that actually creates the necessary components, given an initial minimal configuration, and injects all the necessary dependencies into objects.
Its role is also to start all the goroutines, and orchestrates the interaction between the Control Plane Manager, Data Plane Manager, Input Manager and the Storage Manager.

It injects a channel into the Control Plane Manager to listen and wait for objects of type DataPlaneEnv, which we will later use to configure the Data Plane Manager, start it, as well as the other modules (depending on the roles).

Furthermore, ir registers into the Data Plane Manager all the necessary callbacks for handling forwarder requests and sync operations on the read-only persistence nodes. It also creates the appropriate services (either the proxy or local implementation) to give to each node.
Finally, the last things he handles is the entire lifetime of the processes. As a matter of fact, a custom context can be set for the cluster node, with a custom cancel function (for example in the binary it waits for SIGINT). Cluster Node injects the context into any component that runs goroutines, to ensure that when the Cluster Node stops, all the internal components can safely stop and, for instance, deallocating resources.

2.5 Bootstrap and Nameserver

Bootstrap and Nameserver are central servers that were added to facilitate Nodes' jobs.
The Bootstrap node was designed to make sure that nodes could find neighbors without previously knowing their IPs. It has a gRPC server to which nodes can connect after boot-up to gather information about the topology, in particular, retrieving their topology neighbors. The Cluster Nodes connect to the gRPC server to register themselves. The Bootstrap receives the request, extracts the ID, IP and Control Plane port of the node, and stores it both in memory and on a configuration JSON file. The neighbors are assigned using a fixe, deterministic, pattern based on a sequence array of number of neighbors for a node. This index is updated with Round Robin, while neighbors are chosen from the active nodes list, using a decrementing, wrapping, round robin. This created a topology that, while being deterministic, it offered a way implement *Yo-Yo* over an arbitrary topology.
The Bootstrap also responds with a map of neighbors, that is a map of NodeId to Address (Host, Port). The Cluster doesn't connect to them instantly, but rather gives them to the Control Plane Manager, who will try to connect when ready. Since the bootstrap also states the ID of the neighbors, the firs telection step, the exchanging ID one, could be skipped.

The Nameserver, on the other hand, is much simpler. It was designed to facilitate connecting to the chat server Input nodes. Briefly, it's not quite a proxy, but rather a redirecter. It solved the problem of not knowing all the input nodes that are active. As discussed up until now, multiple nodes in the topology can be Input nodes and, since users should be able to use the chat system even though they do not know the single active nodes, a simple nameserver was deployed.
Nodes, after acquiring the Input role, contact the nameserver, which has a gRPC server, to register themselves. They give the nameserver their IP and port used for HTTP, and the nameserver adds them to a circular queue (a slice iterated with Round Robin). The nameserver also hosts an HTTP server, which just takes one of the nodes from the circular

queue and redirect the client there.
Even though this does not really abstract, or hide, the redirection or the presence of multiple nodes, it was designed this way for simplicity. It was not one of the main goals of the project and was realized for convience in a rather, small, amount of time.

3 Development

In the following section information about development are listed. That is, what environment was used to write the application and test it. Moreover, for each of the various technologies and external libraries used, there is a brief explaination of why it was chosen how it was used.

3.1 Development environment

The project is written using **Golang** `1.25.5` using VS-Code as text editor and all the used modules were up to date (reference date 23.01.2026) . It was developed on Ubuntu 24.04 LTS, using the Windows Subsystem for Linux (WSL). This was mainly due to `ZeroMQ` requiring both a `C` compiler to compile and runtime dependencies to run. On Linux (at least on Debian based distributions) both of these are easily solvable using the package Manager.

Note regard testing: Testing of the final application refers to running multiple nodes to see if they correctly terminate the election, and then interacting, as a client, with the chat server, to see if requests are correctly calculated and data stored. The final application was tested on three different machines: Ubuntu 24.04 (WSL), Void linux and a Docker container (Debian based). The application was also tested while development, particularly when there was only the Control Plane, to see how nodes react to neighors' turning OFF and ON during and post election. During-development testing was done primarily on Ubuntu WSL and sporadically on the other two machines. On both occasions, best results where obtained on Ubuntu WSL.

3.2 Technologies and tools

Different tools and technologies were integrated in this project to facilitate multiple aspects of the application.

3.2.1 Node registration and neighbor discovery: `gRPC`

Nodes need to register themselves before entering the Topology, they do that by connecting to a central *bootstrap* node and announcing their IP and ports used. After that, the *bootstrap* responds with a list of neighbors for the node (in this exchange, the node is the client and the bootstrap is the server).

This communication is implemented using gRPC, using Protocol Buffer Language to define the registration function, the request and response message.
gRPC was chosen mainly for its cross-language nature. Since the contract is written in protocol buffer language, it allows to integrate the both the client's and server's side of the call in different languages.
The bootstrap node is implemented in Go as well, but this is mainly for convenience and, if in the future, the same bootstrap had to be implemented in another language, the .proto file can be used to generate the appropriate code automatically, ensuring correct parameter and return types.

3.2.2 Control and Data Plane communication: `ZeroMQ`

Inter-node communication was realized using the `ZeroMQ` framework, both for managing the Topology on the Control Plane, and forwarding requests and responses on the data Plane.
For both, the communication requirements were that a node should be able to send a message asynchronously and always be prepared to receive one. However, in both cases, a node should not block its execution to wait for a response.

`ZeroMQ` was chosen for its brokerless and asynchronous nature. It offers a *fire and forget* message-oriented communication mechanism, where each node can send a message to the socket and resume its task. Moreover, ZeroMQ offers automatic reconnection (when peers disconnect the socket is not closed, but it keeps trying to reconnect) and message buffering (even if the receiver is offline, messages are kept in a queue and not lost). Both of these characteristics were perfect in this scenario.
Specifically, `ROUTER`-type sockets were used, in a `ROUTER-ROUTER` pairing, where each node had a connection to each of its neighbors in the Topology.
`ROUTER-ROUTER` was chosen because it allows bidirectional communication over a single connection.
Moreover, since `ROUTER` sockets require a unique identity, election messages routing was simplified: by using node IDs as socket identities, the system enables deterministic neighbor addressing without additional lookup logic.

The ZeroMQ module used is github.com/pebbe/zmq4, this is a wrapper around the C zmq library, and requires both a C compiler (like gcc, to compile underlying C code) and the dinamic libzmq library.

3.2.3 Webpage routing: Gorillamux

The clients can interact with the chat system using an hybrid REST api. There are both *view* endpoints, that return HTML, and *API* endpoints, that return JSON. Some of these routes contain variables in the URL (such as /users/{username}/{tag}) and should only be reached using certain HTTP verbs. For this reason Gorillamux was used to handle the various routes, rather than the standard net/http package.

3.2.4 Data persistence: SQLite

Implementing a chat system also means storing the users, the groups and the all the messages that are sent.
SQLite was chosen for multiple reasons, primarily for its zero configuration nature. Contrary to a standard MySQL database, they don't require additional setup, which was really useful since *persistence* nodes start the database only when they understand their role.
Furthermore, it also avoided deploying a single, central, database, allowing each persistence node to have its own copy.

3.2.5 Password security: Bcrypy

User passwords were not stored directly, in clear, in the database, as they are previously hashed. Here BCrypt was chosen because it has integrated password salting and adjustable hashing cost (though in the application the default one is used). Since BCrypt uses salting by default, it automatically protects passwords from rainbow table attacks (finding a password from an hash, by calculating the hash of a set of passwords, and comparing the hash).

4 Theoretical References

In the following section are listed the various theory concepts used to develop the project. For each one, there is also a brief explaination of why it was chosen, how it's used and, if alternatives came to mind before development, why they were not used.

4.1 Message-Oriented Middleware

One of the first requirements of this application, after a first analysis, was that nodes should not pause, or block, their execution when communicating with other nodes. That is because an election, a spanning tree construction and even periodic heartbeat messages generate a small, but consistent, traffic. Handling communication in a blocking way would be detrimental, if not leading to a dead-lock. This lead to considering asynchronous communcation, because it would've allow nodes to send messages into a queue and (try) to read from the queue without blocking, each at its pace. Adding message brokers would have posed a serious challenge: a single message broker would be both a bottleneck and a single point of failure, while adding more message brokers would require handling another distributed system of its own, while nodes would need to know multiple brokers.
In the end ZeroMQ was chosen for this reason, as stated in the **Technology and Tools** section.

4.2 Marshalling and Unmarshalling

Every information exchange between nodes is node via message passing. To achieve an efficient (and simple to use) way to so, JSON was heavily used. This only required to define structures that could be serialized and deserialized.
The most prominent example of such structs is the `protocol.Message` interface, which is the type used at both the Control and Data Plane of the Cluster node. These are, in fact, the messages used for joining the Topology, to carry out an election, build the spanning tree and even forward data requests.
This made communication much simpler, because each node, at high level, knows that it needs to send a `protocol.Message` and it should receive objects of type `protocol.Message`.

4.3 Dependency Injection

Apart from bootstrap and nameserver, which cover a smaller role in the system, the main partecipant is Cluser node. This structure was developed with Separation of Concernes, Loose coupling and composition in mind. Since there are lots of structures that use, or depent to, other structures and interfaces, most are passed during, or after, construction to grant flexibility and interchangeability, while keeping at minimum the objects created internally. Examples of this are:

- Topology Manager: Expects a ConnectionManager to handle the actual neighbor connection. The actual given object is a ZeroMQConnectionManager, which implements the interface using ZeroMQ sockets. Ideally, the cluster node could also give another implementation during construction;
- Control Plane Manager: Almost all components are passed after construction; all loggers, each Manager (Topology, tree, election, heartbeat) or even the channel used to send the data Plane environment configuration;
- This also allowed sharing the same resource to multiple objects, such as the clock. ClusterNode is that holds a RWMutex guarded clock, but it also injects it into the logger, the Control Plane Manager and Data Plane Manager, to make sure each component is synchronized, and not using a clock of its own.

As a matter of fact, Control Plane Manager is able to work on its own: if we were to connect a Control Plane Manager to a Topology it would be able to partecipate in an election and the construction of spanning tree. The ClusterNode is a *shell* that holds together the various components, it has a Control Plane Manager so that it can use its result to correctly start the Data Plane Manager and, optionally, the Input and Storage Manager.

4.4 Synchronization mechanisms

The continuos flow of messages in the Topology, paired with the fact that each node could experience different execution speed or network latency meant that even a simple synchro-

nization mechanism was needed to avoid distributed time races (for example a node that receives two proposals during the election, one from this round and the other for the previous one, but can't tell the old one is obsolete).
Synchronization was implemented using a simple lamport clock, abstracted in the LogicalClock structure. It has a simple counter (uint64) guarded by a RWMutex (to avoid data races within goroutines). Each node increments the clock by one before sending a message, storing the new value inside a Timestamp field in the message header. After receiving a message, the sender's clock is retrieved from the message header and the local clock is updated using Lamport's rule, ReceiverClock = MaxReceiverClock, SenderClock + 1.
Vector clocks were in mind at first as well, however they would've become hard to maintain as the system, or well, the numbers of neighbors, increase: sending the whole vector inside each message would make traffic heavier. Furthermore, both election and spanning tree construction algorithms do not require total ordering:

- Election algorithms require that all nodes end up considering the same one as leader, a simple lamport clock is sufficient to determine if a message is old, regarding which arrives first. Even if more messages arrive simultaneously, the one with stronger ID is accepted (ID must be unique);
- Spanning tree construction algorithms work similarly, the clock is just used to understand if a message is for an event that already happened or not. Even if two nodes send each other a Q (asks the other to become tree-child), they will, deterministically, not create a cycle (to send a message a node is either ROOT or already has a parent, meaning they always respond with NO).

Either way, for the scope of the application, a simple lamport clock was sufficient.

4.5 Election

One of the main goals of this project was to address how election algorithms work in practice, and how close they are to their theoretical implementation.
Since the actual Topology construction (which node has which neighbors) is delegated to the bootstrap, it cannot be assumed that it will be a totally connected graph (n nodes and $\frac{n(n-1)}{2}$), consequently, the Yo-Yo algorithm was used to elect a leader. For this implementation, the cretaria was '*biggest clock, smallest ID*'.
As stated before, Yo-Yo was used instead of Bully since there is guarantee of a global network view. Pruning was also performed to ensure that the algorithm correctly terminates, and to speed up its convergence. Another factor that sped up the election is that the nodes already know their, and their neighbors', ID, allowing each node to skip the first phase of ID discovery.

4.6 Spanning tree construction

After the election, the system starts the construction of the spanning tree of the Topology. This is started by the Leader, once it realized it won the election. As a matter of fact, protocol SHOUT is used by the leader to broadcast the LeaderAnnouncement message: this ensures that each node acknowledges the leader and builds (its local view of) the spanning tree.
Also the actual logic of each node follows protocol SHOUT, the leader starts by sending LeaderAnnouncement messages, which are considered a 'Q' message in protocol SHOUT. Each node that receives a LeaderAnnouncement responds with a LeaderResponse, acknowledging the leader and responding to the 'Q' message (by setting a flag in the message).

4.7 Layered architecture

Multiple components of ClusterNode present a layered architecture structure, both on the Control, and data Plane.
The first example is the Control Plane Manager itself, and the way it handles messages in general, consider a scenario where node 'A' sends a message to 'B' and 'B' receives it (it basically extends the TCP/IP stack):

- Node A prepares the message, marshalls it into a byte payload and gives {B, payload} to its Topology Manager;
- A's Topology Manager receives {B, payload} and forwards {identity-B, payload}to the connection Manager (identity is the name set to the ZeroMQ socket 'node-¡nodeId¿');
- A's connection Manager receives the payload, and uses identity-B to send the payload to it using its ROUTER socket;

If they previously connected with one another, the ZeroMQ socket will receive the message and store it in its internal buffer. When B read a message (calls Recv()) the following will happen:

- B's Connection Manager will retrieve the message from ZeroMQ, passing the raw byte payload to B's Topology Manager;
- B's Topology Manager will extract the identifier of the message (since ZeroMQ writes a ROUTER socket identity in the first frame before sending), and give node B A, payload;
- At the Control Plane Manager level, B will unmarshall the message and will finally be able to read it.

A visual representation of this workflow is given with Figure 1 Other examples of layered architecture can be found on the data Plane:

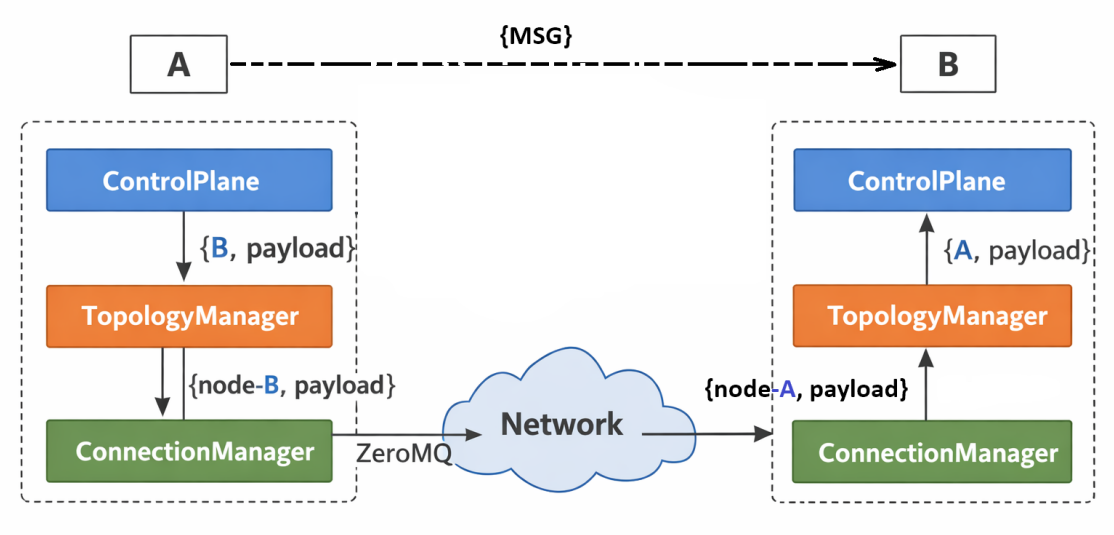

Figure 1: Workflow of sending a message {MSG} between two nodes

- A client request passes from the Handler to the Service and finally to the Repository (this happens on a node that has a local DB);
- On a node that does not have a local database, a client request passes from the Handler, the Service, gets to the data layer, gets forwarded to a node with a database. The response takes the same path in reverse.

4.8 Other concepts and patterns used

There are multiple other theory concepts that were used to develop the application that, while not covering a big chunk of the project, still were useful. First of all Remote Procedure Calling, specifically using gRPC. Its purpose was mainly to abstract the calls between ClusterNode and both Bootstrap and Nameservice. This was with the possibility in mind that, these two services, could be integrated into other application, written in different languages. Protocol buffer language would help in such scenario, being language agnostic. Finally, both GORM and SQLite were used to handle data persistence. Particularly, GORM was used to facilitate mapping objects to database records; the underlying engine was SQLite, mainly to deploy a zero-configuration database on just the nodes that needed it.

5 Tipical use case

Tipical use case of the application would require both to start the Bootstrap and Nameserver servers, and then starting a series on nodes. A very simple way to build and execute the application is given by the makefile in the project directory. Two sides of use case would be

presented, one for the server (whole nodes setup) side one for the client (http connection) side.
Note: For this execution nodes will be run as different process on a single WSL machine just for presentation reasons, execution would not differ on remote machines.

5.1 Server side

Execution on the server side is basically entirely automated, the only thing to do is to build, and run the Bootstrap, Nameserver and Node, as shown in Figure 2. After that, all the

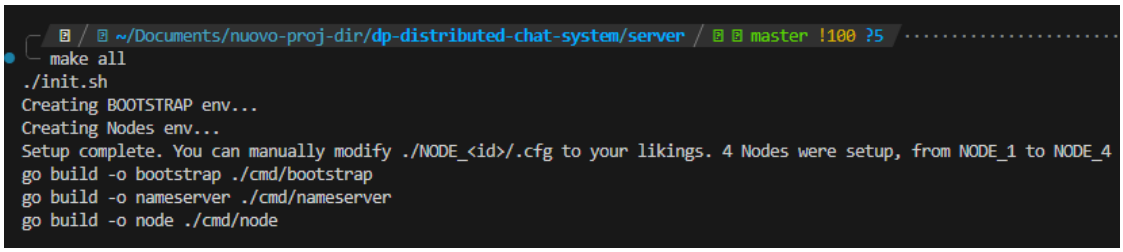

Figure 2: Building binaries with makefile

binaries should be run, as shown in Figure 3. There is not much more that can be done

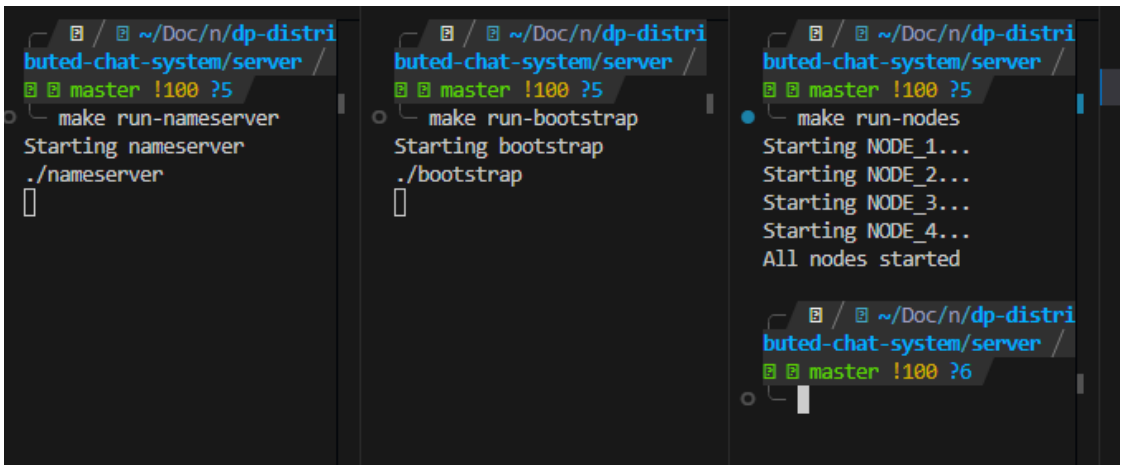

Figure 3: Running binaries with makefile

on the server, but it can be veriefied that the nameserver works, as shown in Figure 4. A local address was shown since the application was running as multiple processes inside WSL, however, the nodes correctly open their ports, same goes for the HTTP port aswell, as shown in figure 5. Apart from manually stopping nodes to test re-election or tree heal tries, on the server side, the only real interaction is stopping the binaries. This can be done by sending SIGINT (CTRL-C or kill -2) to the bootstrap and nameserver, and by using the makefile for the nodes, as shown in figure 6

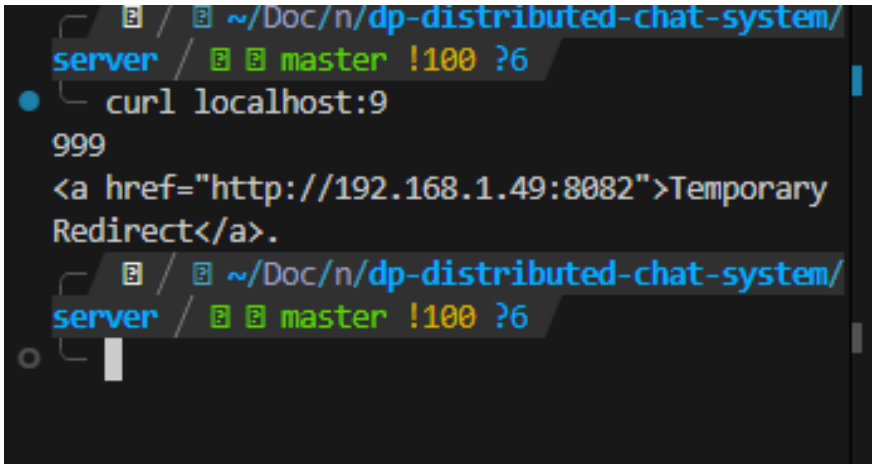

Figure 4: Redirect with nameserver

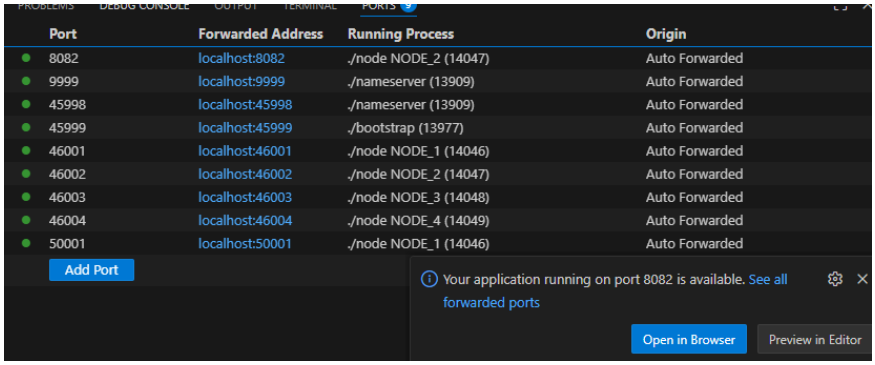

Figure 5: Open ports on both Control and Data plane

5.2 Client side

Most of the interaction actually comes from the clients side, for example, when connecting to a the server, while still having a valid session cookie, they are authenticated and a main page is shown, like in figure 7 From there most interaction is done from the URL searchbar. For example, to enter a group chat, the group must first be retireved, this is done with a GET request on the route /groups/{uuid}, as shown in figure 8. As a matter of fact, only the users in a group can access it, for example, by trying to access with the other user, I would get the result shown in figure 9. Finally, if I were to add both users to the group they could access the group chat. A screenshot does not truly show the persistence, but it shows the, almost minimal, user interface, as shown in figure 10.

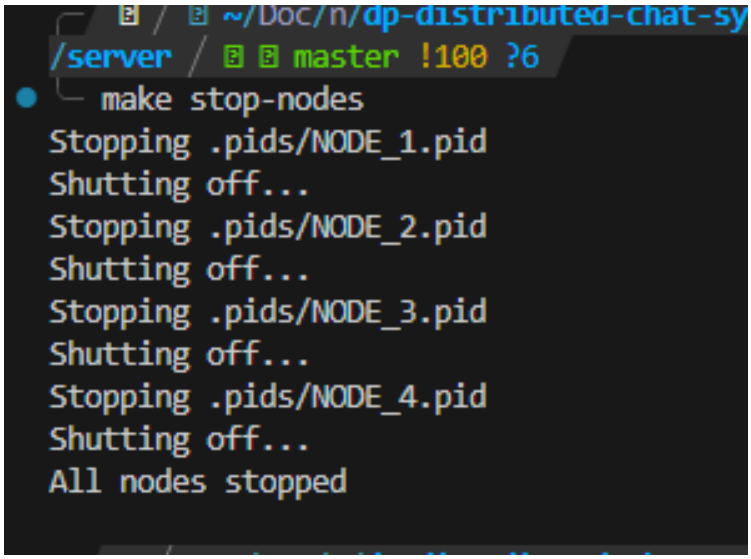

Figure 6: Open ports on both Control and Data plane

6 Additional information

6.1 Self critiques

The bootstrap node and nameserver actually are a Single point of Failure. If the bootstrap failed, nodes would not be able to connect to one another (to connect, someone should write a little program creating a cluster node and not connecting it to the bootstrap, but directly to a neighbor. On the other hand, if the nameserver were to crash no problem would actually arise on the server, it would just make client's attempts to connect harder (since they would now need to know each input node's IP and port).

Initially I though about pausing the InputManager to hot-swap the services, something that could happen when a node switches from input to persistence (the Proxy Service should be replaced with a Local one). However in the end I was not able to implement this in time, however this should not pose a problem, since persitence nodes close their input server, if they had it.

6.2 Differences from the project proposal

In the end, the final application has changed a bit from the initial project proposal, ranging from what the application offers to clients, to the technologies used.

For instance, in the initial proposal, users were supposed to be able to send friend requests to each other. In the final application however, there are only users and group chats, messages

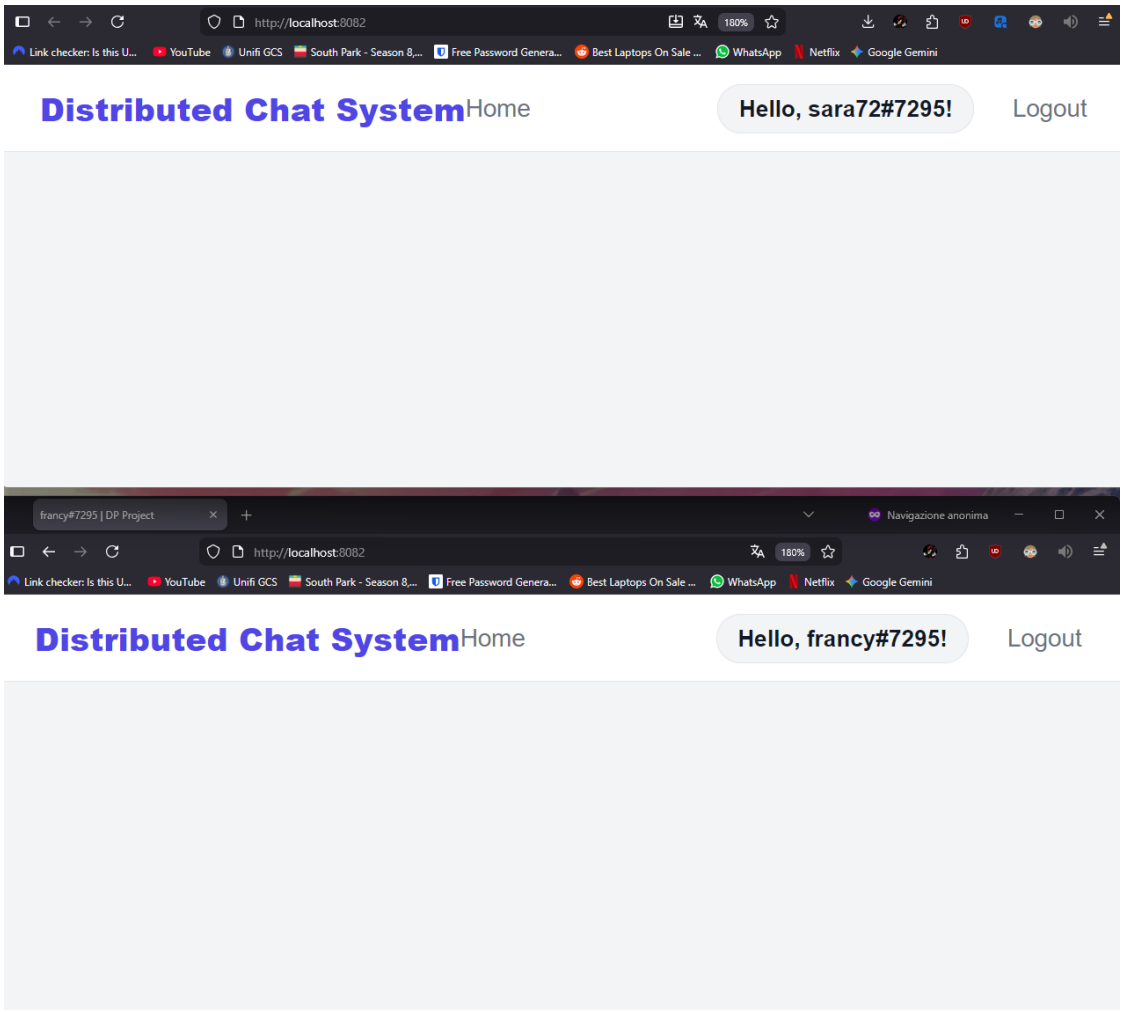

Figure 7: Main page of the application

can be sent between two users or to a group chat:

- Every user can send messages to any other user, he just needs to know his username and tag, then search `<chat-system-ip-addr>/users/{username}/{tag}/chat`;
- Users can send a message to a group only if they are added to said group by another member.

. Moreover, also file transfer was not implemented in the end, because once the infrastructure to send inter-node messages, store data and handle requests by users is set, adding multiple entities is just a matter of writing an appropriate handler and/or add more complex queries; I preferred to concentrate more on the *distributed* part of the project.

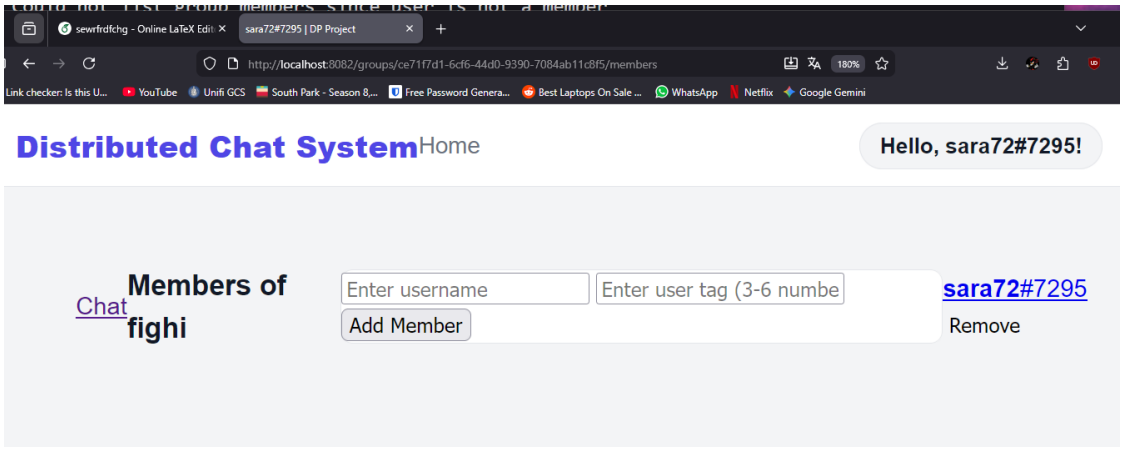

Figure 8: Member of a group can access its member list

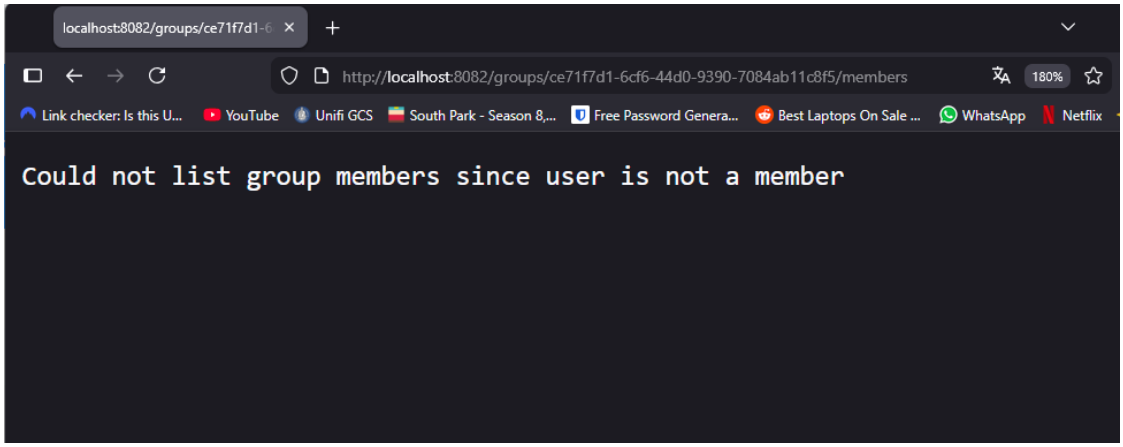

Figure 9: Non member of a group cannot access its member list

Another thing that is different is the way users can reach both the view and API endpoints (i.e. the input nodes). Initially there was supposed to be a central nameserver that acted like a *reverse proxy*, gathering user input and making the request on their behalf to the server. This would have allowed the user to be completely agnostic with regards to which input node they were interfacing (always showing a static IP address). On the other end, this would have both implied adding a possible bottleneck to the system; that's why in the end, the nameserver's task is to accept the client's connection and instantly redirect them onto one of the currently active input nodes.

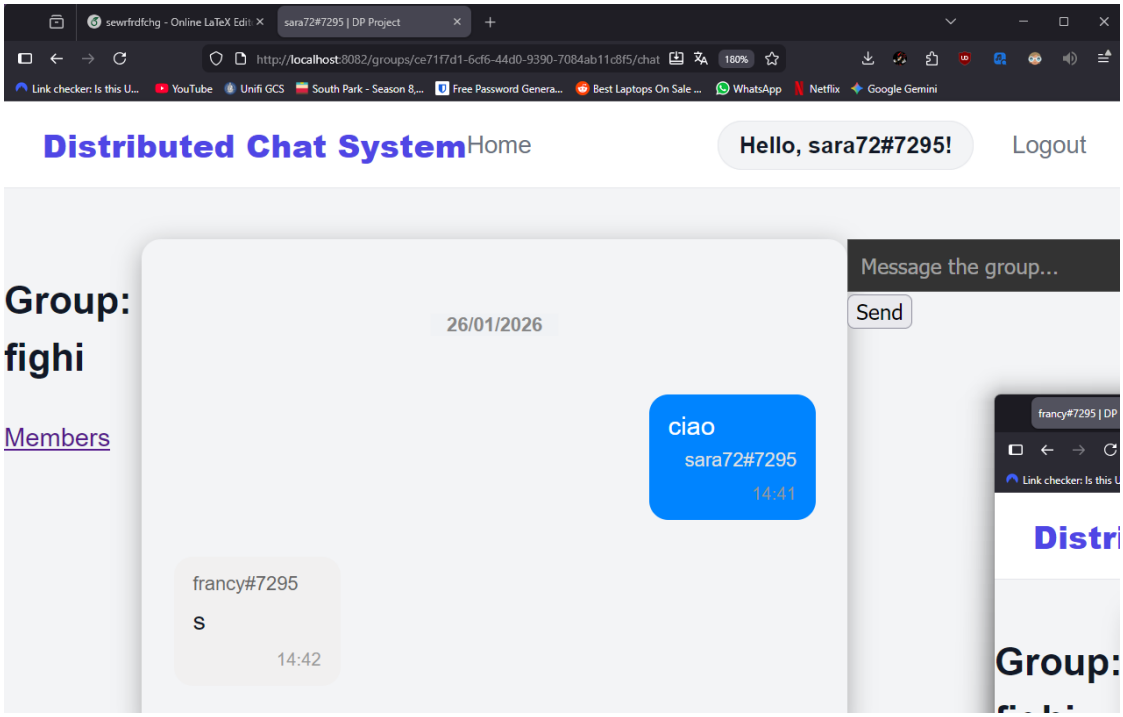

Figure 10: Example of user interface