Chapter 3 - System Architecture

The overall application has been deployed in a distributed way. The system is composed of three different nodes: the frontend node, the backend node and the database node. These nodes are briefly described in what follows.

3.1 - Frontend node

The frontend node of the system is the node providing the frontend service, and therefore communicating directly with the end user: in particular, when the user asks for a page of the website through an HTTP request, such a request must be sent to the port at which the HTTP server is listening. The frontend node provides all the web pages to the end user, and it is also able to communicate with the backend server when necessary, in order to present to the end user all the data (s)he asked for, and more in general to manage the user input.
More in detail, the frontend node is constituted by a Unix virtual machine, providing a React 3 development server, that is based on NodeJS 4. When a user asks for a page of the web application through an HTTP request, such a request is sent to the port 3000, at which the server is listening: the server will then provide the requested page, as well as the data that are necessary to let the application work in a proper way. The frontend node is also able to start a connection with the socket server implemented in the backend node, so that each user is able to receive notifications.
We have chosen not to replicate the frontend node, mainly because it was a matter of time. It could have been replicated at least one or two times, and it could have been added an NGINX load balancer that would have been able to balance the amount of requests made by the client to all the frontend replicas.
Since the frontend node is not replicated, it is a single point of failure. Therefore, if it crashes, all the system will go down; that's why we have tried to limit the damage by configuring the single node with the following restart policy: `restart: unless-stopped`. In this way, we ensure that the frontend container is always running, even if it crashes: indeed, the container is set up to automatically restart in case of failure, so that the frontend service will be always available. Since this has been done for all the containers of the system, we are in a crash and recovery failure model.

3.2 - Backend node

The backend node of the system is the node providing the REST web services that are necessary for implementing all the logic of the application.
Within the routes that can be reached when issuing a REST API, there is a function implementing the specific logic for providing that particular functionality. For instance, if the user wants to login into the application, a REST API will be issued, and the corresponding function will handle all the logic that is behind the login feature. For this reason, the backend node is able to communicate with the database server, in order to retrieve, insert or update the necessary data; the backend node is also able to communicate with the front-end node, in order to permit the presentation of the results of the application logic. More in detail, the backend node is constituted by a Unix virtual machine providing two services: a REST web service and a socket server.

The REST web service has been implemented by using Flask 6, a python-based web framework that provides tools, libraries and technologies to build a web application. It works on port 5000, and it allows managing REST APIs. In practice, when the frontend asks for a specific REST API, the server searches for the correct handler, and then it executes the corresponding function, implementing the logic of the functionality that has been requested. Almost all the handlers that have been defined communicate with the database node of the system, a PostgreSQL database, by means of the psycopg 2.7 python library.

The socket server has been implemented by using Flask-SocketIO 8, a python library that is able to let the Flask server behave as if it was a socket server. Therefore, the port 5000 accepts not only the traffic of the REST APIs, but also the traffic of sockets that are used for implementing the logic for asynchronous messaging. This is true in particular in the case of users belonging to a team: if two users belong to the same team, they can chat to each other, and they can receive notifications that are specific for the team which they belong to. This is how we have implemented a Topic-based Publish/Subscribe communication paradigm: if a user accepts the invitation to belong to a team, (s)he is subscribing for receiving all the notifications about events that are related to that specific team. The publisher is a REST API that, after having created a new event, broadcasts the notification message to all the members belonging to that specific team.

We have chosen to replicate the backend node, in order to increase fault tolerance and the availability of the system. The second replica maps the port 5001 to the container port 5000, so that both servers can continue to listen on port 5000, without the need to modify the code. Moreover, we have chosen to configure both replicas (the primary and the backup ones) with the following restart policy: `restart: unless-stopped`.
In this way, we ensure that both replicas are always running, even if they crash: indeed, both the containers are set up to automatically restart in case of failure, so that the backend service will be always available. Additionally, the backend logic is also tolerant to one physical fault, as there is more than one replica: if the default endpoint is a (physically) failed replica, then the request is forwarded to the other one.

3.3 - Database node

The database node of the system is a node providing a database management system, that is able to manage all the data that are necessary for the correct behavior of the application. The database node is only able to communicate with the backend node of the system, in order to provide the requested data.
More in detail, the database node is constituted by a Unix virtual machine, providing a PostgreSQL 9 database management system. At the beginning of each deployment, the database will be initialized by the means of an SQL script that will be executed only if there is no data in the database (i.e., if no previous deployment has been made).
We have chosen to replicate the database node, in order to increase fault tolerance and the availability of the system. The second replica maps the port 5433 to the container port 5432, so that all the requests to the database node can be done by asking port 5432, without the need to modify the code. Moreover, we have chosen to configure both replicas (the primary and the backup ones) with the following restart policy: `restart: unless-stopped`. In this way, we ensure that both replicas are always running, even if they crash: indeed, both the containers are set up to automatically restart in case of failure, so that the DBMS service will be always available. Additionally, the database node is also tolerant to one physical fault, as there is more than one replica.
In the case of the database node, we have implemented the so-called primary-backup technique, for ensuring software replication: the original replica acts like a primary, and the replicated one acts like the backup. Indeed, the

primary replica receives invocations from the backend logic, and it sends back the correct answer. The backup replica will simply store the state of the computation: indeed, the primary replica will update the state of the database, that is shared between the primary and the backup. If the primary replica fails, after a while the backup replica is able to detect that the primary replica has crashed, and therefore it is asked to become the new primary replica, at least until the original primary replica is physically restored. The only problem with this approach is that the backend always tries to connect with the primary replica; after a timeout, it tries to connect with the backup one. This delay is experienced by the end user, who feels that the system is not working or is not responding. However, after a few seconds, the system will become responsive again.

Obviously, the database node does not tolerate two physical faults: if both replicas are physically shut down or destroyed, the overall database node would go down, and the entire application would stop working.