

LDAP subtree merging proxy daemon

Project for the Distributed Systems course - A.Y. 2023-2024

Michele Ravaioli

`michele.ravaioli3@studio.unibo.it`

Eugenio Tampieri

`eugenio.tampieri@studio.unibo.it`

October 17, 2025

This project presents the design and implementation of an Lightweight Directory Access Protocol (LDAP) [5] proxying service deployed within a containerized environment. The service accepts incoming LDAP requests and forwards them to one or more target servers, aggregating all the responses of the single servers into one.

Its primary use case is the consolidation of multiple Active Directory domains [1], allowing authentication queries to be executed from a single entry point while transparently returning results as if all users resided within the same domain. While remaining online, the service can be configured through a control panel web app accessible to the administrators, which will be able to add and remove servers and clients.

Contents

1	Analysis	3
1.1	Functional requirements	3
1.2	Non-functional requirements	3
1.3	Usage scenarios	3
2	Design	3
2.1	Architecture	5
3	Implementation	5
3.1	Database	5
3.2	Control panel	7
3.2.1	Backend	7
3.2.2	Frontend	7
3.3	LDAP Merger	7
4	Validation	8
5	Deployment	8
5.1	How to deploy	8
5.1.1	Docker	9
5.1.2	Kubernetes	9
6	Conclusions	10
6.1	Future works	10

1 Analysis

The goal of this project is to design and deploy a proxying service for *LDAP* servers capable of handling requests efficiently. It should be able to achieve its goal while remaining always available, even when its configuration is changed while online.

1.1 Functional requirements

The service should be able to:

1. **Request Forwarding:** Route incoming *LDAP* requests to the appropriate servers registered within the system. In order to use the service, the requester must be registered in the system.
2. **Multi-Target Handling:** In cases where multiple servers are valid targets for a request, forward the request to all relevant servers and aggregate their responses into a single unified result (proxy merger).
3. **Consistency:** If even a single server of the ones it should forward a request is unavailable, then the whole request should fail.

1.2 Non-functional requirements

The system should fulfill the following requirements:

1. **Runtime Configurability:** Allow servers, clients, and settings to be updated without downtime or interruption/restart of the service. The service configuration can only be changed by an administrator of the system.
2. **Containerization:** The service should be deployed as a container.

1.3 Usage scenarios

2 Design

Given the functional and non-functional requirements, we opted to provide a cloud-native service composed of three main components: the **LDAP merger**, the **database** and the **control panel**. The proxy merger is in charge of redirecting all incoming *LDAP* requests based on the settings stored in the database. The database information can be updated by the control panel, which allows administrators to modify the list of servers and clients.

The system can encounter these failures:

1. *LDAP* servers are unreachable or their connection times out
2. The merger cannot reach the database

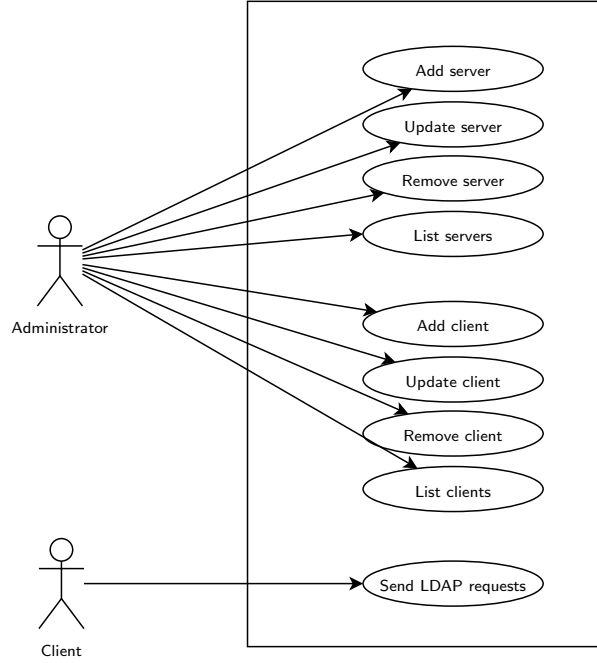


Figure 1: Use case diagram of the *LDAP* proxy service

3. The control panel cannot reach the database
4. The control panel backend is unreachable
5. The control panel frontend is unreachable

The first two failures are the most severe, because they can impair the core function of the system, while the last three only impair the ability to update the configuration at runtime. It must be noted, however, that existing connections are not affected by the unavailability of the database.

In particular, when an *LDAP* server is unreachable it could be because of a network partition or because the server is not available. For what concerns this system, the two failures can be considered a single one and, according to the CAP theorem [3], we will have to choose whether to sacrifice *consistency* (and return data from the available servers) or to sacrifice *availability* (and to return an error, even though a partial response could be constructed).

Lastly, the second failure is critical and poses no choice other than sacrificing *availability*, because the system will not know which servers to contact.

The merger will return a failure when at least one server is unreachable or if the database is unreachable to ensure *consistency*.

To handle the authentication (Fig. 2), we will assume that the system is configured with a username and password for each merged server. The system will then authenticate the clients against a list of credentials configured by an administrator and, if the

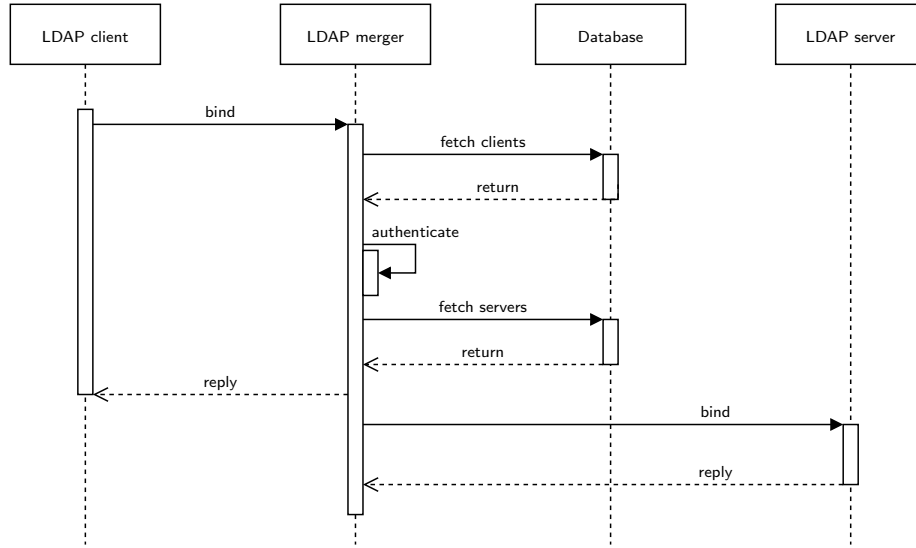


Figure 2: Sequence diagram of a successful authentication with only one merged server

authentication succeeds, it will send a bind request to each backend using the credentials obtained from the database.

Through the control panel, an existing (and privileged) administrator can create other administrators, choosing whether they have the privileges to add other administrators or not.

2.1 Architecture

We opted for a microservices architecture, where each service can be scaled independently to match the load.

We have divided the system into three main services (Fig. 3):

LDAP Merger This component manages all incoming *LDAP* requests, forwards them to the right servers, and then sends the aggregated replies back to the correct client.

Database This component stores *LDAP* servers, clients and administrative users.

Control panel This component allows the system to be configured via a web interface.

3 Implementation

3.1 Database

We opted to use *MongoDB* because of its high availability and replication features. More specifically, we chose MongoDB replica sets [4] to support fault tolerance and high availability.

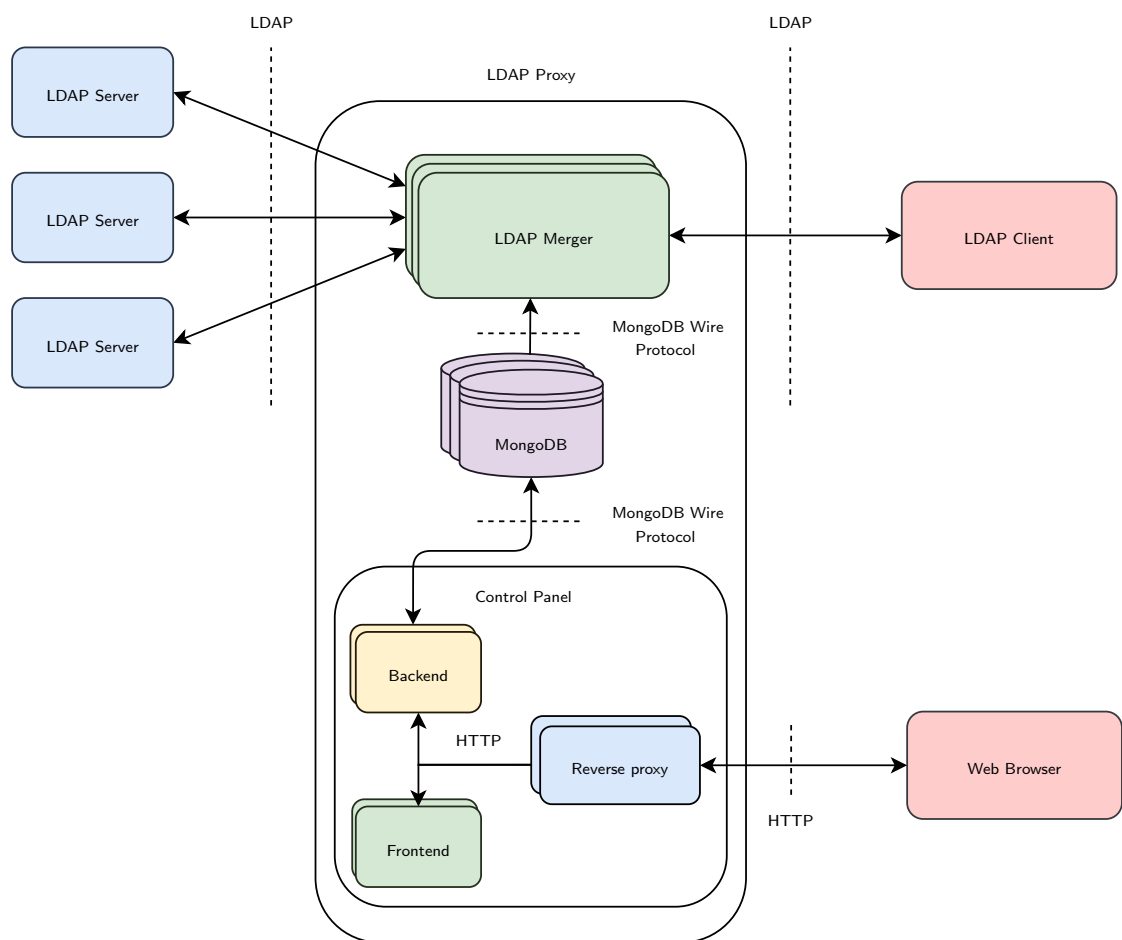


Figure 3: Architectural diagram

A *MongoDB* replica set entails the replication of all the data across multiple nodes. This means that if a node fails (and the quorum is respected), the data is still accessible. *MongoDB* is a *BASE* [2] system, meaning that the replicas obey *eventual consistency*, ensuring that data can be accessed from any node after a small amount of time. This is not a problem for our use case, because the number of writes is dwarfed by the number of reads.

The schema is made up of three collections:

1. **Client**: this collection stores the bind DN and the hashed password for each client.
2. **Server**: this collection stores the servers' bind DN, password, base DN and connection informations.
3. **User**: this collection stores the username, the password and the privileges of each administrator.

The schema is generated from a JSON Schema [7][8] ¹ file to keep it consistent with the rest of the system.

3.2 Control panel

This component is structured as a *Vue.js* Single Page Application that communicates with an *express* backend, which in turn reads and modifies data from the database. The authentication is handled through JSON Web Tokens (JWTs).

3.2.1 Backend

The backend provides a set of JWT authenticated REST APIs that allow Create, Retrieve, Update and Delete access to the three domain entities.

We opted to implement the backend using Typescript for the enhanced type checking and to share types between the frontend and the backend.

3.2.2 Frontend

The frontend is a set of simple views optimised for both desktop and mobile that use Bootstrap components and allow to create and modify clients, servers. Privileged administrators can also create and modify administrators.

3.3 LDAP Merger

The proxy merger component was implemented in Python using the *ldaptor* library [6], which provides a lightweight and reliable framework for building *LDAP* servers. *ldaptor* enables the creation of LDAP Mergers that establish multiple connections in parallel, automatically aggregate responses, and offer high configurability.

¹The second references refers to the most recent specification, while the first refers to what we are using in this project

The merger is designed as a read-only service: it supports only `bind` and `search` requests, while rejecting all write or modification operations. A request is processed only if the client is registered in the database. Once validated, the merger forwards the request to all relevant servers (also stored in the database) and aggregates their responses (Fig. 2). If any server returns an error (e.g., due to unavailability or lack of authorisation), the entire procedure fails, and the client receives an error response.

Thanks to the underlying reactor pattern provided by *ldaptor*, the system achieves high responsiveness and supports multiple concurrent LDAP Merger instances. This design allows the service to scale dynamically with the incoming request load.

4 Validation

To ensure both the performance and the accuracy of the system, automated testing was implemented for the **LDAP Merger** and **Control Panel** services. In addition, the system is continuously validated through integration workflows on GitHub.

The **LDAP Merger** service was tested using *Twisted Trial*, which enables the creation of a custom testing infrastructure. This was necessary because *ldaptor* does not provide a dedicated testing framework. To verify the correct behavior of the merger, mock components were developed for servers, clients, and the database, allowing controlled and reproducible test scenarios.

To ensure that all microservices have the same definition of the data, we used JSON Schema to define the main entities. Those are used to generate the *MongoDB* schema and the Typescript definition files. The schema is also used to validate the JSON serialised Python classes in CI.

5 Deployment

Each component of the system is containerised separately, with its own Docker image. Services can be deployed independently using platforms such as *Docker* or *Kubernetes*, which also support automatic scaling of pods to handle varying request loads. Both the LDAP Merger and the Control Panel are stateless services, making them straightforward to scale horizontally without additional concerns. The Database, however, requires special attention due to its stateful nature, which complicates scaling. This challenge is addressed by *MongoDB* through the use of Replica Sets, which distribute database replicas across multiple instances while preserving consistency and reliability of the system. In order to have a simpler database configuration, we decided to make the number of replicas fixed.

5.1 How to deploy

The system can be deployed using Docker Swarm or Kubernetes.

The repo contains a `docker-compose.yaml` file and the in the `k8s` directory.

5.1.1 Docker

1. If the *Docker Swarm* is configured, otherwise run `docker swarm init`
2. Deploy the stack: `docker stack deploy -c ./docker-compose.yaml ldapProxy`
3. Retrieve the ID of a *MongoDB* container with `docker ps`
4. Launch a *Mongo shell*: `docker exec -it <container_id> mongosh`
5. Initialise the *replica set*:

```
1  rs.initiate({
2      _id: "rs0", members: [
3          {_id: 0, host: "mongodb1"},
4          {_id: 1, host: "mongodb2"},
5          {_id: 2, host: "mongodb3"}
6      ]
7  });
8  rs.status();
```

6. The previous command should return the *replica set*'s status. Check that it is healthy
7. The application is now running on port 8080.

5.1.2 Kubernetes

1. The provided manifests deploy the application under the `ldap-proxy` namespace. If that is not what you want, the you will have to modify all the manifests.
2. Set the domain name in the control panel's *Ingress* by modifying `k8s/ingress.yaml`
3. Deploy with `kubectl apply -f k8s`
4. Attach to a *MongoDB* pod and run:
`kubectl -n ldap-proxy exec -it statefulset/mongodb -- mongosh`
5. Initialise the *replica set*:

```
1  rs.initiate({
2      _id: "rs0", members: [
3          {_id: 0, host: "mongodb-0.mongodb"},
4          {_id: 1, host: "mongodb-1.mongodb"},
5          {_id: 2, host: "mongodb-2.mongodb"}
6      ]
7  });
8  rs.status();
```

6. The previous command should return the *replica set*'s status. Check that it is healthy

6 Conclusions

In this project we designed, implemented, and validated an *LDAP* proxying service aimed at consolidating multiple *Active Directory* domains under a single access point. The system successfully forwards and aggregates *LDAP* requests, ensuring transparent authentication across domains while remaining configurable at runtime through a dedicated control panel.

By adopting a microservices architecture and containerized deployment, the solution achieves scalability, availability, and ease of integration with cloud-native platforms. Stateless services such as the LDAP Merger and the Control Panel can be scaled without complications, while database consistency is ensured through *MongoDB* Replica Sets.

Validation through automated tests and continuous integration confirmed the correctness and reliability of the core components. Overall, the project demonstrates a flexible and fault-tolerant approach to *LDAP* proxying, capable of supporting complex multi-domain environments in a transparent and maintainable way.

6.1 Future works

According to our initial report, we should have built a test environment. However, we felt that it was out of scope for our project, because we are already simulating a set of possible fault conditions using mocks. This area of experimentation is interesting, however, and merits further exploration.

References

- [1] *[MS-ADTS]: Active Directory Technical Specification*. 2024. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-adts/d2435927-0999-4c62-8c6d-13ba31a52e1a (visited on 09/22/2025).
- [2] Armando Fox et al. “Cluster-based scalable network services”. In: *SIGOPS Oper. Syst. Rev.* 31.5 (Oct. 1997), pp. 78–91. ISSN: 0163-5980. DOI: 10.1145/269005.266662. URL: <https://doi.org/10.1145/269005.266662>.
- [3] Seth Gilbert and Nancy Lynch. “Perspectives on the CAP Theorem”. In: *Computer* 45.2 (2012), pp. 30–36. DOI: 10.1109/MC.2011.389.
- [4] The MongoDB Project. *Replication - Database Manual*. 2025. URL: www.mongodb.com/docs/manual/replication/ (visited on 09/22/2025).
- [5] Jim Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511. June 2006. DOI: 10.17487/RFC4511. URL: <https://www.rfc-editor.org/info/rfc4511>.
- [6] Twisted. *Ldaptor library*. 2024. URL: <https://github.com/twisted/ldaptor> (visited on 09/18/2025).

- [7] Austin Wright and Henry Andrews. *JSON Schema: A Media Type for Describing JSON Documents*. Internet-Draft draft-handrews-json-schema-01. Work in Progress. Internet Engineering Task Force, Mar. 2018. 25 pp. URL: <https://datatracker.ietf.org/doc/draft-handrews-json-schema/01/>.
- [8] Austin Wright et al. *JSON Schema: A Media Type for Describing JSON Documents*. Internet-Draft draft-bhutton-json-schema-01. Work in Progress. Internet Engineering Task Force, June 2022. 78 pp. URL: <https://datatracker.ietf.org/doc/draft-bhutton-json-schema/01/>.