Department of Biomedical Informatics
Emory University School of Medicine, Atlanta

Securing Linux Storage with ACLs: An Open-Source Web Management Interface for Enhanced Data Protection

Mentor(s): Robert Tweedy and Mahmoud Zeydabadinezhad, PhD

GSoC Project Proposal

Presented to Department of Biomedical Informatics of Emory University School of Medicine for *Google Summer of Code 2025*

by

Aditya Patil

from

Mumbai (India)

About the Contributor

Personal Information

Project Title:	Securing Linux Storage ACLs
Name of Contributor:	Aditya Patil
Academic Studies:	B.Tech Electrical Engineering
Year of Graduation:	2026
Enrolled University:	National Institute of Technology Hamirpur

Skill Set

Languages:	Golang, Rust, Python, Javascript, C, Shell Scripting (POSIX)
Frameworks:	NextJS, SvelteKit, Flask, Django, Actix, Sphinx
Databases:	Redis, OpenLDAP, MySQL, PostgreSQL, MinIO, MongoDB
Dev Tools:	NeoVim, Git & GitHub, GitWeb, Makefile, Kitty, Tmux
Operating Systems:	Arch Linux, Debian Linux & RedHat based Distros
Deployment:	Nginx, Linux Servers, Docker, Kubernetes

**Note: Relevant skills for the proposed project have been mentioned, for more information, please check CV document*

Professional Work Experience

Athena Blockchain Consulting LTD. Dubai

- Solely worked on system design for cloud deployment under the guidance of senior developers for Eumlet (UAE) (financial application built on NextJs) on AWS with Debian EC2 instances and Load Balancer and NGINX.
- Worked on setting build automation pipelines on GitHub Actions and connected with EC2 manually.
- Managed a team of 4 developers in a constraint environment of 2 highly valued clients, namely Lunarspace and Concordium, and wrote legal developer handbooks for new recruits.
- Worked with BGTrade from China for security audits and deployments of financial platforms with teams across the globe.

Technical GTM Freelancer (Tel Aviv, Israel)

- Worked for **Application Security** for writing highly technical SCA tool documentation of 50+ pages and fumadocs.
- Ghost Writing for CTO of security company on **Reachability Analysis** in their Software Composition Analysis Tool.
- Successfully set up flipt on the cloud for video production for **Feature Flagging Platform Provider** and fixed deployment issues on different Linux servers by writing POSIX-compliant installation script.
- Worked for a **Proxy Provider** to create scraping tools for Real Estate Platforms and write articles on them quickly.

Chuku LLC. (Texas, USA)

- Worked on **TP-Link WR-720N Router** for Security Research on **UPNP Protocol** by Firmware Dumping.
- Gathered Information from the Web, Studied Security Issues in UPNP, and created a report for the company's reference.

Research Publications

IEEE 17th International Conference on Communications and Technology (COMSNETS 2025)

- Lead Authored and presented a research paper at an esteemed conference in Bengaluru in communications and technology. Guided by **Dr. Sreeram T.S.** (Assistant Professor at Electrical Engineering Department, NIT Hamirpur.)
- This institute-funded research was based on laying the design for a novel encryption adapter for securing legacy Modbus devices and designing protocols to secure critical systems utilizing ARM-based processors.

Previous Open Source Contributions

Propeller - CRED Club (Pull Request)

- Tools: gRPC, Golang
- Implemented Server Reflection in gRPC in Golang for better debugging and updated docs.

Flipt - Feature Flag Management Tool (Pull Request)

- Tools: POSIX Compliant Shell Scripting
- Fixed installation script by upgrading it from bash script to POSIX-complaint shell script, expanding support for every POSIX shell available on Linux and MacOS while preventing crashes while running installation script on different shells.

Hacktricks (Multiple Pull Requests)

- Tools: Hardware Hacking and ICS Writing
- Worked on researching protocols like UART, SPI, EEPROM, I2C, and Firmware Dumping with reference to the router.
- Initiated the Industrial Control Systems Hacking section and contributed a document on SCADA terminologies.

Kitty Terminal - IV Kitten + Docs (Pull Request)

- Tools: Golang
- Project for **FOSS 2024 Hackathon**, creating IV kitten (recursively display images in a grid) MVP in Golang in 2 days with three others under the guidance of **Kovid Goyal** (maintainer). Merged PR in Docs for CMD configurations.
- Listed in the **"Special Mentions for impactful contributions to projects by maintainer"** in FOSS Hack 2024.

For more information, visit here for a detailed CV document.

Links for References

GitHub:	https://github.com/PythonHacker24
LinkedIn:	https://www.linkedin.com/in/aditya-patil-260a631b2/
Portfolio:	https://minimalistbook.com
Google Scholars:	https://scholar.google.com/citations?user=RXNnvOEAAAAJ
Curriculum Vitae:	https://bit.ly/41zPVyA
Blog Posts:	https://medium.com/@adityapatil24680
Email:	adityapatil24680@gmail.com

Problem Statement

While the traditional POSIX permissions used by nearly all common Linux filesystems allow for simple data permissions management based on group membership, this can become complicated to manage in a large research environment where a project's directory tree may not be efficiently structured to permit access via the simple group management achievable via POSIX permissions, especially when a research PI would like to grant different levels of access to a file to users who are otherwise in the same group. Linux ACLs, while a de-facto standard rather than a formally defined one like POSIX, can be used to resolve these types of situations but have a higher learning curve and far fewer management tools available for a large-scale storage system than tools managing POSIX permissions, relying mainly on the command line tools "setfacl" and "getfacl" [8] to view any details. While there is a GUI tool known as "eiciel" that can adjust Linux ACLs, its audience and feature set is aimed towards an individual user on their personal machine and thus is not suitable for large-scale storage system management. This project aims to develop an application with a web-based GUI that can be deployed on a research network's storage system to provide PIs with a graphical overview of all their research data and allow management of the Linux ACLs to grant appropriate levels of access to the PI's research team members.

Expected Outcomes

A self-contained application (i.e. the application should not require external APIs, javascript, etc. to be queried by the end-user's web browser at runtime and should have any relevant scripts packaged with it; the underlying back-end should only access local network resources and not need external Internet connectivity) that allows an end-user to view and manage the access permissions of a large research storage network at a fine-grained level via a web GUI. Both the back-end logic and front-end web interface are in the scope of this project.

- The application should support authentication via different modules, with a minimum of LDAP-based authentication.
- The application should support deployment behind a reverse proxy running on Apache or Nginx.
- The application must be Linux distribution agnostic as much as possible, but at a minimum should support both running on both RedHat Enterprise Linux (or free derivatives like Rocky Linux) and Debian Linux (or other derivatives like Ubuntu Server) as a SystemD service.
- Documentation explaining a basic overview of application usage and installation steps for use by research network system administrators.
- The code and documentation for the application will be made publicly available on a platform such as GitHub and licensed under an Open-Source license such as GPLv3.

Contents

Chapter 1

Introduction

Institutional departments, like the Biomedical Informatics Department of Emory University School of Medicine, manage vast amounts of data, often reaching petabytes of scales across multiple Linux-based storage servers. This data comprises a wide range of critical resources, including research results, experimental records, videos, image samples, and other materials researchers produce.

Permission management is one of the most significant challenges in managing such extensive data at an organizational level. It is crucial to ensure that sensitive information is accessible only to authorized personnel. Additionally, granting or revoking user access must be streamlined to accommodate dynamic research needs.

While traditional POSIX [4] permissions — widely used across common Linux filesystems — provide a straightforward way to manage data access based on group membership, they become difficult to manage in large research environments. In scenarios where a project's directory structure is complex or when a Principal Investigator (PI) wishes to assign different access levels to users within the same group, POSIX permissions may not suffice.

To address these limitations, Linux Access Control Lists (ACLs) offer a more flexible alternative. However, although widely accepted, Linux ACLs are less standardized than POSIX permissions and have a steeper learning curve. They also lack robust management tools, relying primarily on command-line utilities such as "setfacl" [9] and "getfacl." [8] While the GUI tool "eiciel" [5] can adjust Linux ACLs, its design is intended for individual users rather than large-scale storage system administration.

To overcome these challenges, this project aims to develop a web-based GUI application that can be deployed on a research network's storage system. This tool will provide PIs with a clear graphical overview of their research data and enable them to manage Linux ACLs effectively, granting appropriate access levels to their team members.

Currently, BMI employs a manual approach to data management. Researchers are responsible for uploading and organizing data within their designated directories, and the IT team is responsible for granting or modifying access permissions. This manual process can be time-consuming, prone to errors, and inefficient, particu-

larly as data volume and user demands continue to grow.

The project aims to build a system that provides non-technical users with a customization front from which they can store, retrieve, delete, and manage permissions to files and directories. It is intended to be kept open source with a GPLv3 License for use beyond BMI Infrastructure. Beyond the timeline of GSoC [10], it will continue to be developed and improved through community contributions.

Chapter 2

Project Specification

The approach of this project starts from the current implementations in BMI's internal infrastructure. Fig. 2.1 shows the current architecture of BMI's internal file management system. As per the project specifications in BMI's GitHub [1], the project's design must strictly consider components in the network.

BMI Research Network Architecture Overview

Here is the breakdown of components in the current system design and their specifications:

System Components

- **End-Users**
  - Clients (Desktop/Workstation)
  - Secure connection via SSH and/or HTTPS
- **Firewall**
  - Serves as a security barrier between End-Users and the Internal BMI Network
- **Internal BMI Network**
  - **Computational Nodes**
    - Slurm-managed compute jobs only
    - No direct SSH access
  - **Computational Cluster Job**
    - Runs QOS Web-based GUI or graphical interfaces
    - Supports SSH services for CLI interactions
  - **Authentication Service (FreeIPA)**
    - Supports standard LDAP lookup tool

    * Provides secure user authentication
  - **Filesystem Mounts**
    * NFS Mount (via Ethernet)
    * BeeGFS Mount (via Ethernet + Infiniband)
  - **Storage Cluster**
    * Centralized data repository for user data, project files, etc.

The project's design takes inspiration from the underlying system design but not the application running on it. Sufficient computation power would be allocated for the project's deployment inside BMI's premises for testing purposes.

System Requirements

The functional and non-functional requirements listed here are per BMI's GitHub project listings. The author has not made any interpretations here.

Functional Requirements

- **Web GUI for Permissions Management**
  - Provide an intuitive web-based interface for end-users to view and manage access permissions.
  - Support fine-grained control over permissions across a large research storage network.
- **Authentication Support**
  - Implement modular authentication mechanisms.
  - Must support LDAP-based authentication at a minimum.
- **Self-Contained Application**
  - Application must bundle all required scripts to operate independently.
  - No reliance on external APIs or JavaScript libraries that require internet access during runtime.
  - Backend logic must only access local network resources.
- **Deployment Support**
  - The application must support deployment behind a reverse proxy running on:
    * Apache
    * Nginx
- **SystemD Service Integration**

  - The application must run as a SystemD service for both:
    * RedHat Enterprise Linux (and derivatives like Rocky Linux)
    * Debian Linux (and derivatives like Ubuntu Server)
  - The application must operate under a limited service account (not as the root user).

- **Service Account Permissions**
  - : These must be clearly documented if the application requires elevated permissions (e.g., AmbientCapabilities in the SystemD script).

- **Documentation**
  - Include clear instructions for:
    * Application usage
    * Installation steps for research network system administrators

- **Open-Source Licensing**
  - The code and documentation must be publicly available on a platform like GitHub.
  - The application must be licensed under an Open-Source license, such as GPLv3.

Non-Functional Requirements

- **Performance**
  - The application must efficiently handle large-scale storage networks and numerous permission entries.

- **Reliability**
  - The application must maintain uptime and consistent behavior across supported Linux distributions.

- **Security**
  - The application must adhere to best security practices, especially when handling user authentication and file system permission changes.

- **Scalability**
  - The system should be scalable to accommodate growing data volumes and expanding user bases.

- **Portability**
  - The application should be Linux distribution agnostic, supporting both RHEL-based and Debian-based systems.

- **Maintainability**
  - The code should be clean, well-documented, and modular to ensure long-term maintainability.
- **User Experience**
  - The web GUI must provide a clear, intuitive interface for easy permission management.

Further discussions of the project would strictly adhere to satisfying the given requirements.

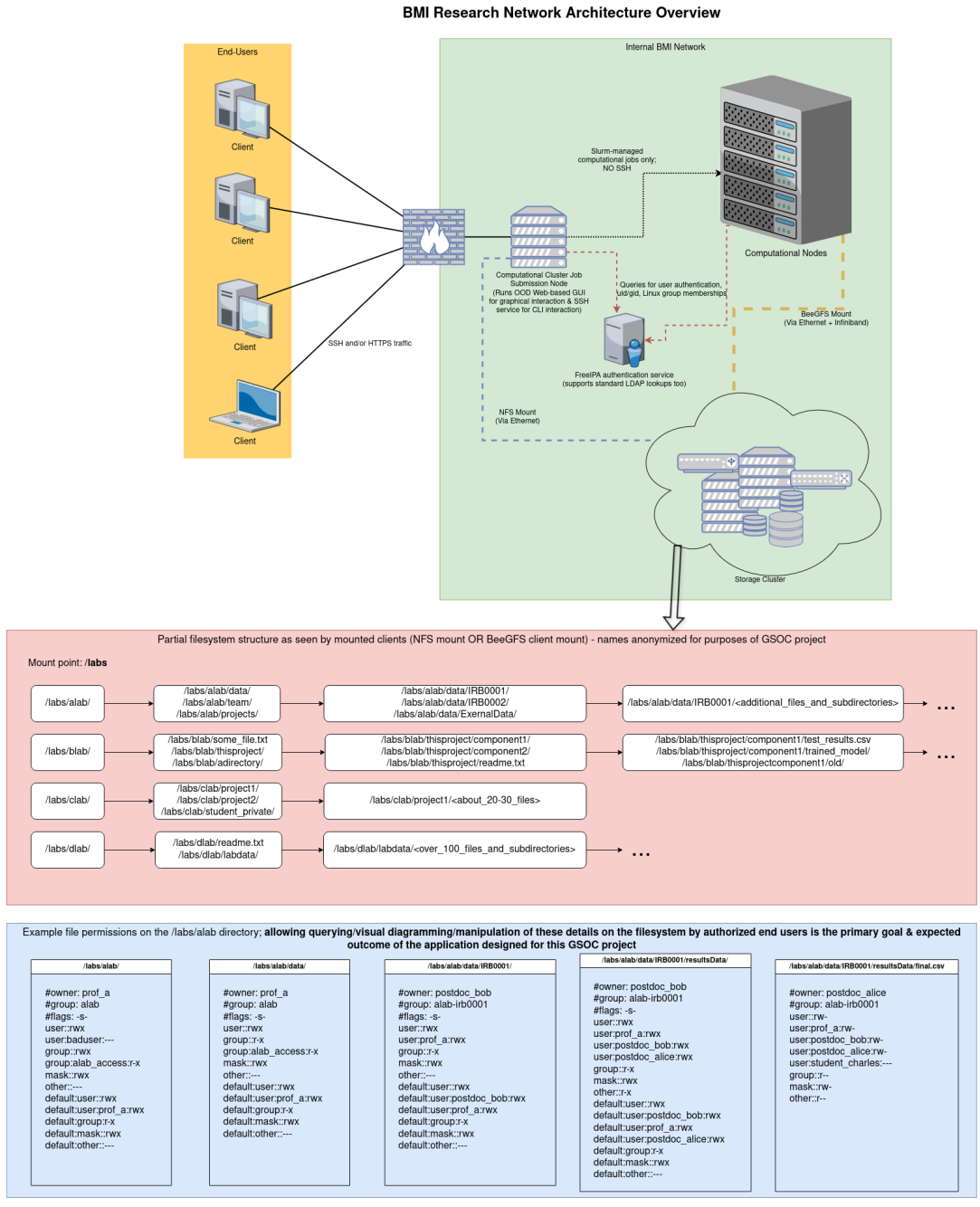

Figure 2.1: BMI's Current Architecture for File Storage

Chapter 3

Tech Stack

The tech stack was decided during the first meeting with mentors. Following is the decided tech stack approved by the mentors:

Golang for the back-end server, APIs, daemons, and data-intensive applications like logs streaming

Reason: Performance, Simplicity, Developer Productivity, Safety, Stability, Powerful Standard Libraries, Support from Google, Best in Class Concurrency Support [Author's Proficient Language] [7].

Python for Data Analytics and IT Operation Backends (only if required)

Reason: Simplicity, Huge Support from Analytics GUIs and Data Processing Libraries [Author's First Language and Most Experienced]

C or Rust for Lower Level Operations (only if required under special circumstances)

Reason: While interacting with filesystems, we might need lower-level access for managing permissions and for security purposes, optimizations

Javascript for Frontend Development

Reason: Since we need to keep our front-ends lightweight, we would prefer using Vanilla JS or, in case of higher requirements, NextJS/SvelteKit due to its performance and developer base.

SQL Databases, Redis for Caching, and LDAP for Authentication

Reason: This depends on how we use it and how fast we need it to be. This can be decided later since I am comfortable with any of them, and each has its own

pros and cons. Specifically, SQL can be done with PostgreSQL [6], caching with Redis, etc. LDAP is assumed to have already been configured in the system.

Tarball for Installation, Docker as Second Priority (as instructed)

Reason: Docker would be the second installation priority. BMI prefers deployment with tarballs with source code for security reasons.

Sphinx for Documentation

Reason: Sphinx Docs is a widely used documentation framework known for its simplicity, elegance, and lightweight nature, making it ideal for this application.

The documentation will be divided into two sections: User Docs and Code Docs.

Product Docs - This is for non-technical users who want to learn how to use this tool from the front-end side if they have any issues. This is like the manual researchers use if they are having an issue. This also contains deployment instructions for IT Teams.

Code Docs - This is for IT Teams who want to use APIs provided in the Backend and want to develop our product.

GitHub for Distribution, Git for Version Control

Reason: Most Open Source Projects are hosted in GitHub and have a vast user base. Functionality-wise, it's sufficient. The project will be distributed under the GPLv3 license. Git is an industry standard for version control.

Chapter 4

High-Level System Design

A considerable amount of time was dedicated to meticulously curating the project's High-Level System Design under Robert Tweedy's guidance.

In this section, we will explore the project's finalized high-level system design, supporting the low-level system design decisions made in the proceeding sections.

This design holds hands with current implementations in BMI's internal network, as mentioned in Fig. 2.1. Efforts have been made to keep the design as modular as possible due to its open-source nature and to keep it accessible to a wide range of users with different requirements.

Upon discussions and curation, Fig. 4.1 is designed and approved by Robert Tweedy.

Explanation

Frontend

The front-end is responsible for interacting with the user. It sends requests to the back-end server through decoupled APIs. The front-end user interface handles user input, displays data, and manages interactions efficiently.

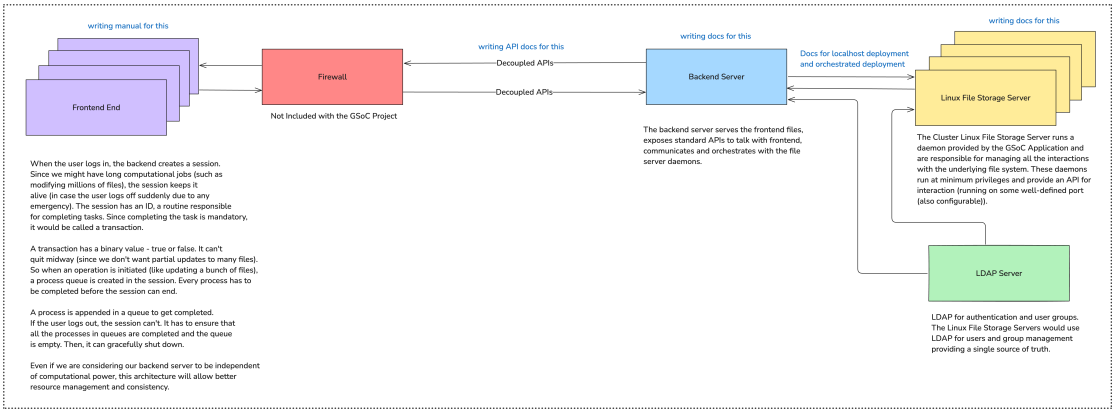

Figure 4.1: Proposed High-Level System Design

Since this is an Open-Source project, various organizations will use this tool. Hence, decoupling the APIs is essential since many might want to configure their own or modify the design. This project will be shipped with a default front-end decoupled from the back-end and used by BMI.

Firewall

The firewall acts as a security layer between the front-end and back-end servers. It ensures that only authorized and adequately formatted requests reach the back-end server.

The firewall is not included in the GSoC project and is assumed to be installed by the organization. However, it has been well-established in BMI's network.

Backend Server

The back-end server serves the front-end files, exposes APIs to communicate with the front-end, and orchestrates tasks with the file server daemons and manages locally mounted filesystems like BeeGFS. It is responsible for creating user sessions, managing transactions, and completing processes in order.

Linux File Storage Server

The Cluster Linux File Storage Server manages interactions with the underlying file system. Servers with filesystems like NFS where local command execution for **setfacl** is required will be executed by a daemon provided by the GSoC application that operates at minimal privileges. These daemons expose APIs for interaction and are configurable via a defined port. In filesystems like BeeGFS, the filesystem will be mounted on the backend server in the specified directory in the configuration file, and a daemon will not be required for such filesystems.

Upon discussion with Robert Tweedy, we must provide options for running the daemon and back-end server on the same server in case the user lacks resources and requirements for managing clusters. In this case, the daemon communication would be confined to localhost (via Unix Sockets). In the case of clustered deployment, like in the case of BMI, network protocols (like gRPC [11]) will be used with the back-end running dedicated hardware and orchestrating multiple Linux file storage servers.

LDAP Server

The LDAP Server manages authentication and user groups. The Linux File Storage Servers use the LDAP server to provide a single source of truth for user identities and group permissions.

The LDAP server is assumed to be installed by the user to manage users across the Linux file storage servers. The back-end would be connected with the LDAP

server for authentication purposes. It cannot make any changes (it might do with plugins, but it is currently a low priority for the GSoC project).

Deployment and Documentation Scope

- Writing manual for the Frontend End.
- Writing API documentation for the Backend Server.
- Writing deployment and orchestration documentation for the Linux File Storage Server.

Additional Ideas during Emails with Mentors

During the design phase and conversation with Robert Tweedy, we came up with various lower-level ideas like session management of users, transactions, process queues, and reverting changes during unexpected faults like server crashes, power failure in servers, etc. More on this in the Low-Level System Design Chapter.

Chapter 5

Low-Level System Design

The application has three components: front-end, back-end, and daemon (for Linux Storage Cluster like NFS). Each is intended to be kept loosely coupled for flexible usage and deployment. This chapter contains a detailed layout of low-level design components like endpoints, session management strategies, data structures, orchestration algorithms, etc. These designs are formulated through discussions and strategic planning but may undergo minor modifications during development.

Backend Server

Frontend Facing Endpoints

The front-end-facing endpoints will follow REST API architecture. These standardized APIs will be configurable by any client following the standard, making it possible for users to create their own clients apart from the default front-end that ships with the project.

Authentication

When users go to the specified URL, they must log in with credentials. Upon successful login, a session is created. When no processes are queued, the timeout counter jumps in, which ends the session if no read/write/update/delete operation for permissions is queued. This session timeout parameter can be configured. This serves as a safety mechanism and closure of the session in case of idle workload. When the session is closed, the user must log in again to access the resources.

Transactions and Session Management

A transaction is an operation that includes actions such as read/write/update/delete on single/multiple files and directories. Upon the execution of the transaction, the result is sent to the user: completed successfully, incomplete, or failed. These transactions are added to a queue of the process scheduler. The timeout counter is reset and paused when active transactions are in the queue. Even if the front-end is disconnected for any reason, the process queue remains intact, and the session is

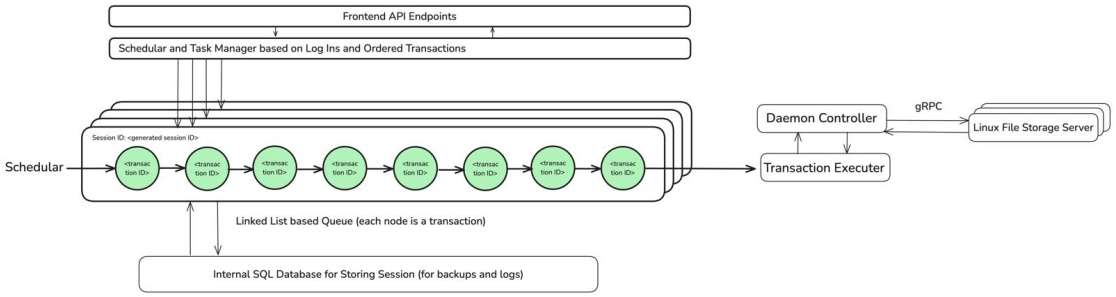

Figure 5.1: Transactions Architecture

alive until all the transactions are executed. Once the process queue is empty, the timeout counter is started, and the session is closed if there is no user interaction. If the front-end has already logged out, there is no requirement for the timeout counter, and the session can be closed safely.

NFS vs BeeGFS Filesystems

Filesystems like NFS behave differently than BeeGFS in terms of ACL management. Since NFS is a remote-mounted filesystem, ACL changes must be executed on the NFS server itself. If **setfacl** is executed on an NFS client, changes won't apply unless the NFSv4 ACL support is enabled and mapped correctly. So, in this case, **setfacl** must be executed on the file server itself through a daemon that the backend server will contact. In this case, the user provides a file path relative to /mnt of the back-end server, and the backend needs to resolve the respective file server's path and contact the daemon to make appropriate changes.

In filesystems like BeeGFS, ACL modifications done on the client side are perfectly valid and propagated through the whole network of filesystems. This allows the back-end server to apply ACL modifications on /mnt mounts and requires no daemon interaction. Other filesystems like BeeGFS that follow similar ACL modifications are CephFS, GlusterFS, etc.

This concept is better illustrated in Fig. 5.2

Daemon Interaction and Orchestration

The daemons running on Linux File Storage Servers would be connected with gRPC, which is faster than the REST framework of APIs and follows a stricter schema for interaction. During installation, the configuration would be done using YAML [12] files to specify the hosts and ports on both sides, with access tokens for security purposes. gRPC supports SSL/TLS for enhanced security, which can be utilized in this case.

The daemons must be deployed only on the servers serving file systems where modifying operations on mounted paths on the backend doesn't propagate through the network. For example, NFS (using setfacl on /mnt/nfs-mount) doesn't make any change on the network level, so we need to use **setfacl** commands on the file

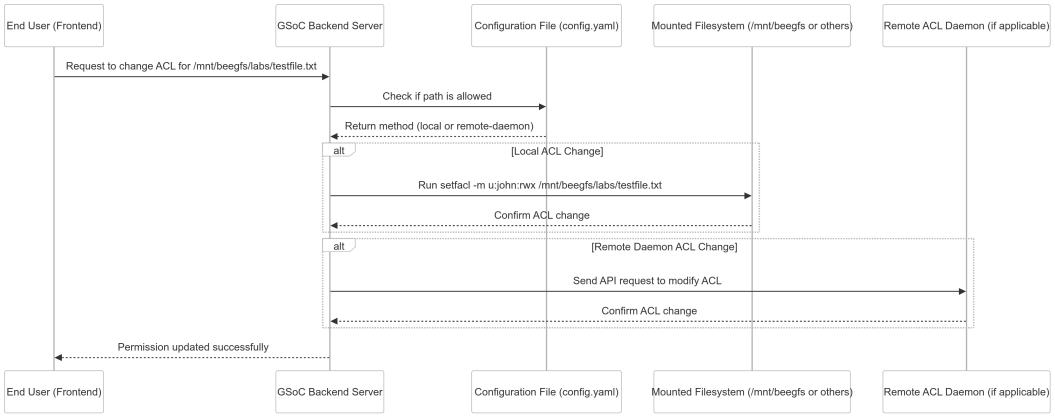

Figure 5.2: BeeGFS vs NFS Interaction for Permission Management

servers locally through daemons. In the case of filesystems like BeeGFS, daemons are not required, and the backend server makes the changes through mounted paths (like /mnt/beefs).

In localhost deployments (discussed in the Deployments Chapter), Unix Sockets would be used to interact with the underlying daemon.

Frontend Server

The front-end server is a Web UI that is exposed to application users. The user should perform all interactions via the front-end, which would be loosely coupled with the back-end server and follow asynchronous communication (as much as possible).

Whenever a user logs into the front-end, they are authenticated by the back-end, and a session is created (as explained above). In an idle state, these sessions have a timeout, after which the user has to log in again. This is done for safety and to close the session if the user leaves the front-end application stale.

When transactions are scheduled to run on the back-end server, a session has a queue of transactions it has to process and will not be closed even if there is no front-end interaction. This ensures that time-consuming transactions are processed in the background even if the front-end loses its connection with the back-end. A conceptual representation of the front-end design has been illustrated in Fig. 5.3. Note that it's for ideation purposes, and the final design will have more features and a modern UI.

Permission Management

Users would be able to list different users on the front-end and assign/retract/update permissions to their own files/directories. Only the permissions provided by the users themselves can be seen by them on other users and nothing else.
Since users are not changing very frequently, we can make use of caching databases

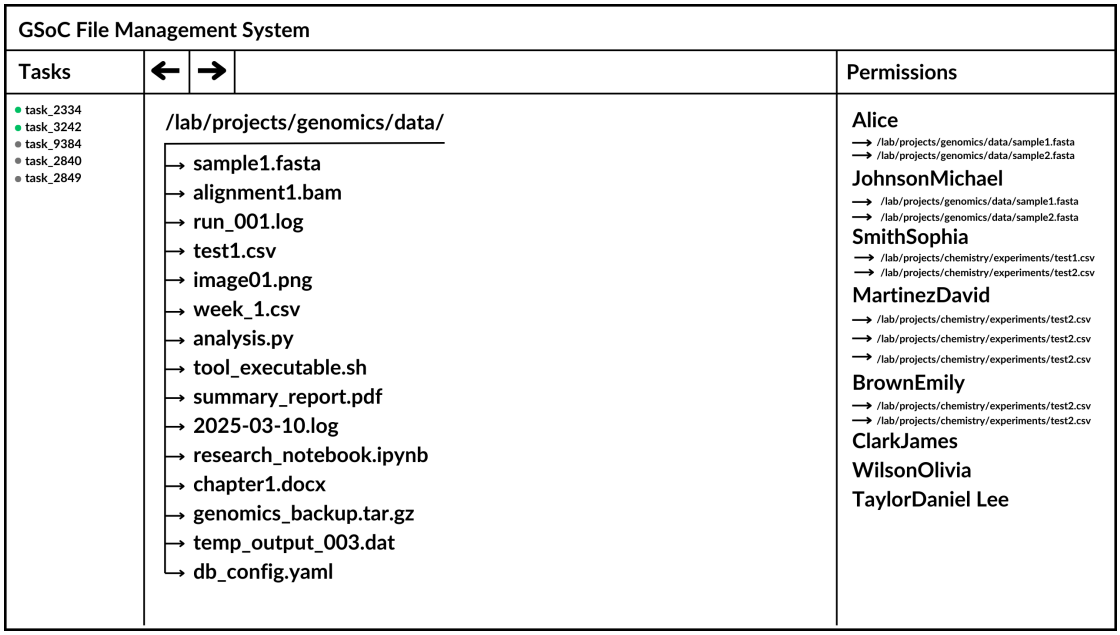

Figure 5.3: Frontend Conceptual Design

like Redis to provide information about the users. Updated permissions would be automatically translated into transactions that will be added to the process queue of the session to get executed with **setfacl** by the Linux File System Daemons.

Linux Storage Server Daemons

The Linux Storage Server Daemons are installed on the Linux File Servers where the files are stored. These daemons read, write, delete, and update permissions on the Linux Servers. They communicate with the back-end server using gRPC, and configurations are made using the YAML files.

These daemons follow the least privilege principle for security reasons. They interact directly with the underlying file system and use **getfacl** and **setfacl** for ACL management as per the requests from the back-end server.

Each operation executed by these daemons is named a transaction (discussed in earlier subsections). Currently, only one transaction is executed at a time. Since the back-end is responsible for process management, these daemons only execute transactions as instructed. Upon completion, the result of the execution is given back to the user.

LDAP Server for Testing Purposes

A formal definition: The Lightweight Directory Access Protocol (LDAP) is an open, vendor-neutral protocol used for accessing and managing directory information over a network. It enables applications to authenticate users, retrieve

organizational details, and enforce access control policies. LDAP organizes data hierarchically in a tree structure called the Directory Information Tree (DIT), where entries like users, groups, and devices are stored. Each entry is uniquely identified by a Distinguished Name (DN) and consists of attributes such as cn (Common Name), uid (User ID), and mail (Email). LDAP supports simple authentication (username/password) and SASL-based authentication for stronger security. It operates over TCP/IP, typically on port 389 (unencrypted) or 636 (with SSL/TLS).

During this project's development, we will use localhost deployment of OpenLDAP [3] **(OpenLDAP Public License)**, an Open Source and widely used LDAP server. Since LDAP is a vendor-neutral protocol, any LDAP server can work with the back-end after development. During the network testing phase, the OpenLDAP server would be hosted on a different network (like on a cloud service) and tested with the back-end during the final stages of testing.

Particularly for the development process, we will use the Docker images https://hub.docker.com/r/osixia/openldap for its hosting.

The docker container can be started on localhost with:

```
docker run -p 389:389 -p 636:636 --name openldap \
    -e LDAP_ORGANISATION="Example Corp" \
    -e LDAP_DOMAIN="example.org" \
    -e LDAP_ADMIN_PASSWORD="admin" \
    -d osixia/openldap
```

For GUI interaction with the LDAP server, we are going to use phpLDAPadmin **(GPL-2.0 License)** https://github.com/leenooks/phpLDAPadmin, which is commonly known as "PLA". PLA is designed to comply with LDAP RFCs, enabling it to be used with any LDAP server.

Particularly for the development process, we will use the Docker images https://github.com/osixia/docker-phpLDAPadmin for its hosting.

phpLDAPadmin can be deployed with a docker container with the following command:

```
docker run -d \
  --name phpldapadmin \
  --link openldap:ldap-host \
  -p 8080:80 \
  -e PHPLDAPADMIN_LDAP_HOSTS=ldap-host \
  -e PHPLDAPADMIN_HTTPS=false \
  osixia/phpldapadmin:latest
```

Chapter 6

Progress on Prototype

In the pre-GSoC period, I worked on building the project prototype, which incorporated all the design decisions proposed in this proposal. This prototype was kept open-source and hosted on GitHub. During the prototyping phase, many new ideas emerged to tackle new problems, which would be incorporated into the final building phase of the project.

Note that the codebase is currently written to create an MVP and prove the design decisions made in this proposal. Hence, I haven't incorporated the coding patterns and practices that would be decided when the project would be in the building phase and open source. This allowed me to speed up the MVP building process and prove my design decisions. However, I have kept the code as modular as possible, allowing me to reuse the final project's components and speed up the development process.

During the prototyping phase, I decided to avoid writing anything for the front end since it has no significant design choices and includes a standard development approach - thanks to the loosely coupled APIs the backend provides.

All the code is maintained in GitHub Repository: linux-acl-management

As per the suggestions of Mahmoud Zeydabadinezhad, the process of building the is documented at https://pythonhacker24.github.io/linux-acl-management/

The Changelog for the whole codebase can be found here: CHANGELOG. Currently, a Python script has been written that automatically fetches details from commits in the local repository and updates the markdown file for the Changelog. A more automated and robust Changelog system will be developed in the final project.

The prototype went through numerous revisions as directed by Robert and Mahmound. All the decisions and updates, date by date, are recorded in the progress report here: Progress Report.

Backend

Codebase: GitHub Link for Backend Code

Since the concept of transactions and session management is novel in its design, explaining its flow through illustration is necessary before processing the implementations. The flow of logging into the backend to issue a transaction is illustrated in Fig. 6.1. Details about each component are discussed in the following sections.

Configuration File

This is an example of the backend.yaml configuration file:

```
deployment-config:
  - host: "localhost"
    port: 9999

transaction-logs-redis:
  - address: "localhost:6379"
    password: ""
    db: 0

basePath: "/mnt"

servers:
  - path: "/beegfs/labs"
    method: "local"

  - path: "/administration"
    method: "local"

  - path: "/labs-data"
    method: "remote"
    remote:
      host: "192.168.1.100"
      port: 5001

  - path: "/admin-data"
    method: "remote"
    remote:
      host: "192.168.1.101"
      port: 5002
```

Important Parameters in YAML Configuration File

- **deployment-config:** Configurations related to the application deployment itself.
- **transaction-log-redis:** Configurations about the redis databases where the transactions results would be stored.

- **basePath:** The base path where all the filesystems are mounted. In case the basePath is not set or set to "" or "/", the backend will consider the "path" parameter in the servers section as the absolute path. This configuration is useful when all the filesystems are mounted in a single directory.
- **servers:** All the filesystems, including BeeGFS, NFS, etc., and configurations about their nature (remote or local) and, if remote, their hosting information. If the basePath is not provided, the user must provide the absolute path to the directory where individual filesystems are mounted.

Endpoints and Handlers

Below are the handler definitions for the prototype until the proposal is written.

The upcoming sections provide detailed explanations of all these handlers. This includes Curl commands, which are necessary and sufficient to demonstrate the use of each endpoint and can be adapted for interaction with any language.

```
mux.Handle("/health", middleware.LoggingMiddleware(
    http.HandlerFunc(
        handlers.HealthHandler
    )
))

mux.Handle("/login", middleware.LoggingMiddleware(
    http.HandlerFunc(
        handlers.LoginHandler
    )
))

mux.Handle("GET /current-working-directory",
    middleware.LoggingMiddleware(
        middleware.AuthenticationMiddleware(
            handlers.GetCurrentWorkingDir
        )
    )
)

mux.Handle("POST /current-working-directory",
    middleware.LoggingMiddleware(
        middleware.AuthenticationMiddleware(
            handlers.SetCurrentWorkingDir
        )
    )
)
```

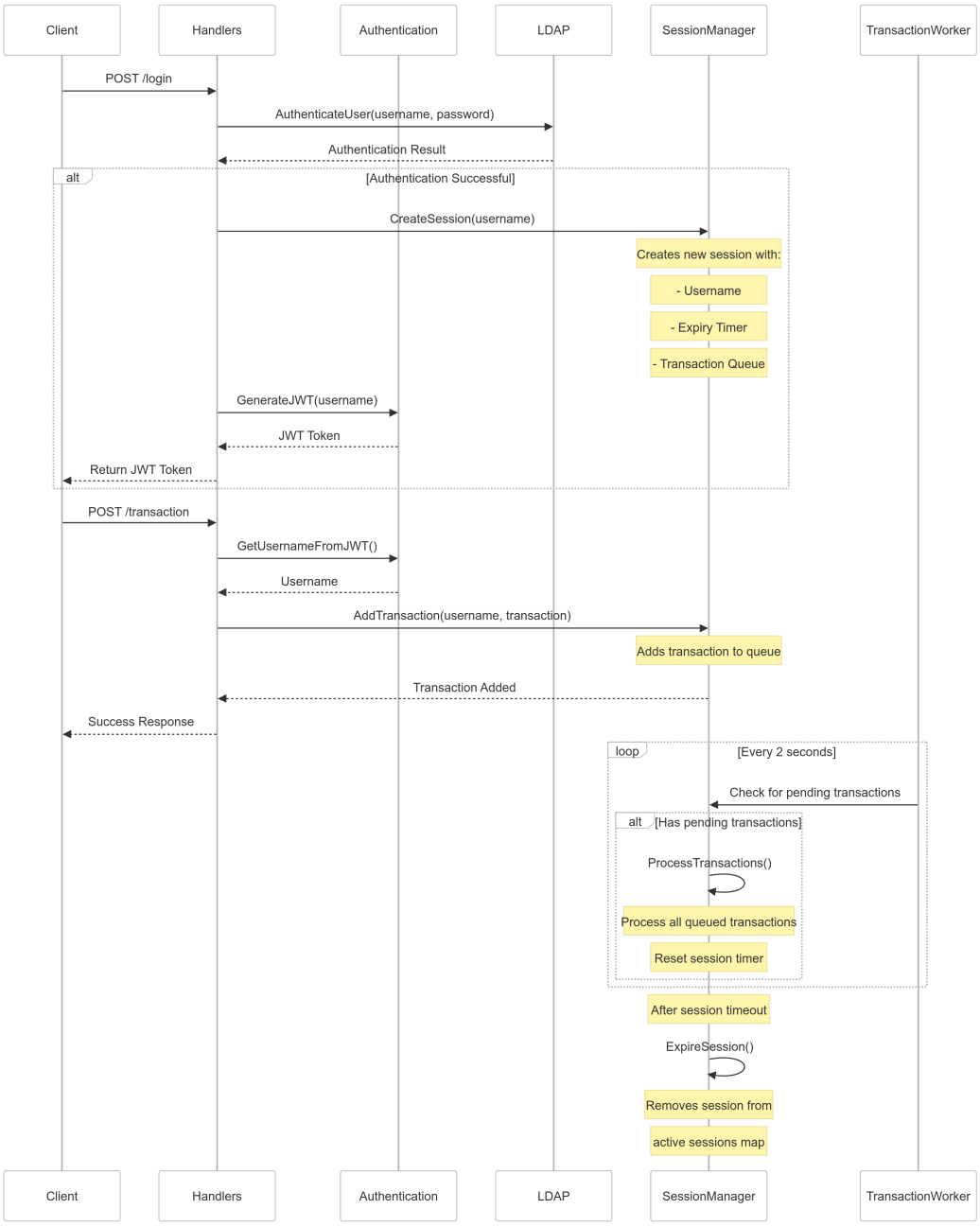

Figure 6.1: Backend Prototype Architecture

```
mux.Handle("GET /list-files",
    middleware.LoggingMiddleware(
        middleware.AuthenticationMiddleware(
            handlers.ListFilesInDir
        )
    )
)
```

Health Handler

The first handler is the **/health** handler, which returns the health of the backend server. This endpoint has no authentication since it's a read-only handler providing a health check on the back-end server. The Linux File Storage daemons and the front-end will use this handler to check if the backend server is healthy enough to handle automatic debugging.

In the final project, the health handler would return more information than responding to an acknowledgment message. It would be used to monitor the health of the backend server, resource utilization, loads, potential issues, etc., which would be used by the IT teams to debug the backend server remotely. In the case of detailed health reports, authentication would be introduced into /health endpoints, too.

Login Handler

The login handler is the starting point of any backend endpoint (except the /health handler). It authenticates the user using the LDAP server and provides a JWT Token [13], which the user uses to perform any operations with API endpoints (for example, scheduling transactions).

A JSON Web Token (JWT) is a compact, URL-safe token format used for securely transmitting information between parties as a JSON object. It consists of three parts—header, payload, and signature—encoded in Base64 and digitally signed, ensuring data integrity and authenticity.

This JWT token must be included in all the requests in an Authentication Header ("Authentication: Bearer JWT-Token").

In the current implementation, the JWT token expires in 24 hours, which can be modified as per the requirements. The user will specify the JWT token expiration time in the YAML configuration file in the final project.

A session is created when the user authenticates with the /login endpoint (discussed in the low-level system design chapter). As discussed, this session has a timeout counter and would expire after a set time (set by the user in the YAML configuration file). After logging in, the user can schedule transactions in the session, and they would be queued for execution.

Example Curl Command for Login Endpoint:

```
curl -X POST https://<ENDPOINT>/login \
```

```
    -d '{
          "username": "USERNAME",
          "password": "PASSWORD"
        }' \
    -H "Content-Type: application/json"
```

Example Response from Login Endpoint if Authentication Successful:

```
{
    "token": "JWT-TOKEN"
}
```

Current Directory Handler

The current directory handler is responsible for showing and updating the user's current directory in a session. This handler reads and modifies the CurrentWorkingDir parameter of the Session struct shown below.

Following the REST principles, GET requests to this endpoint fetch the current working directory of the user session.

Example Curl Command for GET Request to /current-working-directory endpoint:

```
curl -X GET 'https://<ENDPOINT>/current-working-directory' \
-H 'Authorization: Bearer <JWT-TOKEN>'
```

Example Response from the /current-working-directory endpoint for GET Request:

```
{
    "currentDir": "mnt"
}
```

Using the POST Request method, the current directory can be changed.

Example Curl Command for POST Request to /current-working-directory endpoint:

```
curl -X POST 'https://<ENDPOINT>/current-working-directory' \
  -H 'Authorization: Bearer <JWT-TOKEN>' \
  -H 'Content-Type: application/json' \
  -d '{
    "setWorkingDir": "beegfs/"
  }'
```

Example Response from the /current-working-directory endpoint for POST Request:

```
{
    'currentDir': 'mnt/beegfs'
}
```

List Files Handler

This handler returns the content of the current working directory, which is used to display its content and for navigation purposes through the file system. This handler interacts directly with underlying mounted filesystems and provides a structured JSON Schema.

Example Curl Command for /list-files Handler:

```
curl -X GET 'https://<ENDPOINT>/list-files' \
-H 'Authorization: Bearer <JWT-TOKEN>'
```

Example Response from the /list-files Handler:


```
[
  {
    "name": "mri-results",
    "size": 672,
    "permissions": "drwxr-xr-x",
    "user": "adityapatil",
    "group": "staff",
    "mod_time": "2025-04-02T12:50:48.790995956+05:30",
    "is_directory": true
  },
  {
    "name": "ct-scan-results.pdf",
    "size": 132447,
    "permissions": "-rw-r--r--",
    "user": "adityapatil",
    "group": "staff",
    "mod_time": "2025-04-02T12:20:36.233738164+05:30",
    "is_directory": false
  }
]
```


This information can be used to display file and directory information on the front end, including current permissions, user, group, etc. The front end uses the "name" field to navigate through the file system through the /current-working-directory endpoint.

Middleware Implementation

Handlers are wrapped with two middleware (except /health, which only has one middleware for logging): 1. Logging Middleware, which logs all the requests and time taken for their execution. 2. Authentication Middleware, which checks if the JWT token is valid. All the handlers behind the Authentication Middleware are protected handlers, as shown in Fig. 6.2.

The code for the middleware is implemented here: middleware.

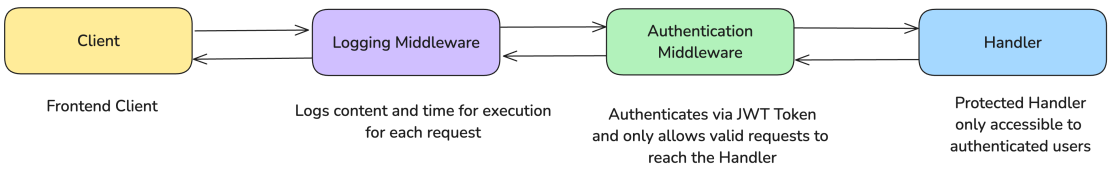

Figure 6.2: Structure of Nested Middlewares

Process Scheduler

The process scheduler was one of the novel designs proposed in this proposal. Hence, I considered dedicating a separate module in the prototype for this, which would be upgraded in the final project. As discussed in the prior chapters, the process scheduler's job is to process transactions issued by the users and get them executed one by one by the transaction executor as per the resource availability.

A session is automatically created when a user authenticates with the /login endpoint. Information about the session and transaction queue is defined in the following struct:

```
type Session struct {
    Username            string
    CurrentWorkingDir   string
    Expiry              time.Time
    Timer               *time.Timer
    TransactionQueue    *list.List
    Mutex               sync.Mutex
}
```

A transaction queue is a queue data structure where new transactions are pushed, and the transactions that are ready to be executed are fetched. This allows the backend to execute transactions systematically and according to resource availability. It also facilitates the logging of the transactions.

Each session has its own transaction queue, and each transaction is executed by the transaction executor. It works independently of the transaction queue manager as a separate goroutine and doesn't interfere with scheduling. In the prototype, this executor is called **TransactionWorker()** and its implementation looks like this:

```
func TransactionWorker() {
    slog.Info("Transaction worker started")
    for {
        sessionMutex.Lock()
        // Create a local copy of sessions to process
        sessionsToProcess := make(map[string]*models.Session)
        for username, session := range sessions {
            if session.TransactionQueue.Len() > 0 {
                sessionsToProcess[username] = session
            }
```

```
        }

        sessionMutex.Unlock()

        // Process transactions for each session without holding the
        // global lock
        for username, session := range sessionsToProcess {
            ProcessTransactions(username, session)
        }

        slog.Info("Transaction worker finished all the transactions")
        time.Sleep(2 * time.Second)
    }
}
```

This function is executed as a goroutine in the **main.go file**. It looks for the transaction queues of the sessions and ensures they are executed. If all of them are executed, it waits two seconds and again checks if there is a new process (the time would be managed by the user through YAML configuration. In the future, time would be replaced by an event-driven mechanism that starts the worker as soon as a new transaction is detected). If there are no transactions, the cycle repeats until new transactions occur.

In **main.go** file, it is called in the following way:

```
go sessionmanager.TransactionWorker()
```

As a prototype, advanced error handling is not implemented here and will be done in the final project.

Also, goroutines can be leveraged more advancedly here. Depending on the availability of cores on the back-end server, varying numbers of goroutines can be spawned to make optimal use of the underlying hardware.

Managing Asynchronous Execution of Transactions and Polling with Redis

When a new transaction is issued, it is first stored in a Redis database as pending. Redis is an in-memory database providing high read/write performance. Each transaction stored in Redis has a TxnID (transaction ID). The front-end will attempt to fetch the transaction result by polling the /get-transaction-result. During the phase where the transaction is still in the queue and not executed, the front-end will get the "PENDING" status result. When a transaction is executed, the Redis entry is updated, and the front-end receives the desired result of the operation.

This mechanism is visualized in Fig. 6.3

The number of times the front end polls the backend server depends upon the configurations made on the front end. This mechanism allows the endpoints to work independently of the session manager and just read/write the state of the backend while the session manager updates the state by its own logic.

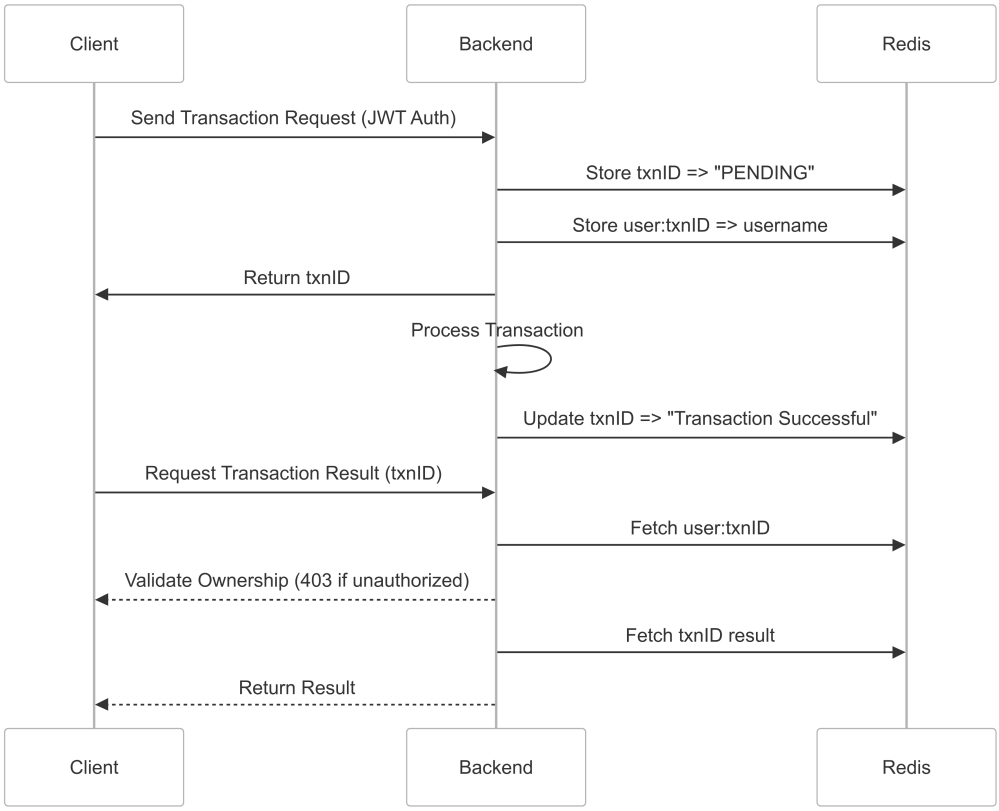

Figure 6.3: Transactions and Redis Database Interaction

Linux File Server Daemon

The primary testing the Linux File Server Daemon prototype had to do was related to gRPC connections, performance evaluation with multiple server clusters, and connecting it with the backend prototype.

So far, a health check RPC function has been implemented to check the daemons' status. This health-check handler would be paired with advanced health checks in the final project - like disk space, CPU utilization, etc.- for advanced monitoring. Currently, it's out of scope for the prototype since the implementation is straightforward and not a core feature.

Frontend

Since the APIs themselves can better simulate everything the front-end can do, the prototype doesn't include the front-end development part unless there is time left before the final project, in which case the foundations of the front-end can be built for testing purposes.

Chapter 7

Deployment Strategies

Setup Strategies

Due to the open-source nature of this project, various users have varying requirements and available resources. Large organizations have multiple Linux Servers for Storage and dedicated hardware for orchestrating them. Others might have lesser requirements and a single server to manage all the files. Hence, this project has flexible deployment methods. The project supports two primary deployment methods:

1. **Orchestrated Deployment:** Multiple Linux Storage Servers must be managed, and the back-end server has dedicated hardware for deployment.

2. **Single Instance Deployment:** Single Linux Server will store files and host the back-end. The back-end server and File server daemon will run on the same hardware in this case.

Orchestrated Deployment

For orchestrated deployment, the user should install the back-end package onto the dedicated back-end server inside the network. This server has access to the Linux File Storage servers via the preferred medium and the front-end devices.

Each Linux file storage daemon must be installed individually and configured to connect with the back-end server. In the future, we can provide Ansible playbooks to configure multiple servers simultaneously as a deployment option. The manual docs constantly instruct manual options for high-security deployments.

Single Instance Deployment

For a single-instance deployment, the user should first install and configure the back-end components for localhost connections. The Linux File Server daemon would be installed on the same system, and a similar configuration must be made (which includes not allowing external communications with the daemon and only accepting localhost connections via Unix sockets).

Here, a single package installation option can be provided, which installs the back-end server and initializes the daemon independently.

Installation

The installation procedure needs to be flexible for this project due to the wide variety of users. Some users prefer ease of maintenance, while a few focus on security. Tools like Docker are preferred for the first category of users, while later users prefer building through source code and deploying manually.

In the first meeting of this project, Robert Tweedy suggested that BMI prefers security and Docker might have security concerns. This project will prioritize source code installation and building over Docker containers. After the former is successfully tested, the latter will be worked on.

Installation through Source Code

Tarball

On GitHub, we can make various releases with versioning. In that case, the source code would be bundled as a **tarball**, which users can **untar** and build the binaries in their premises.

Upon unpacking the source code, users can run commands like **make build** to build the binaries in appropriate directories (for example, build/). For a more manual approach, we can provide step-by-step build commands in the documentation to build everything from scratch.

GitHub Repository Cloning

Since GitHub is our distribution platform, cloning the repository is one of the most popular methods. Users can clone our repository into their local systems and build through the source. This method is suitable for systems with internet connection and for those who need seamless updates through commands like **git pull**.

Deploying through Docker Images

To deploy through docker containers, we would maintain **Dockerfile** for the front-end, back-end, and the file system daemon, which can be pulled and updated from the docker registry. This form of deployment will be suitable for users who need fast deployment with default configurations and have systems with internet access. It is ideal for developers testing out test deployments during development.

As mentioned, dockerization is a low-priority task and will be done at the end of the project's development.

Chapter 8

Documentation

Documentation is a crucial part of any open-source project. For this GSoC project, the process of writing documentation will parallel the development process. Particularly, two types of documentation would be written and hosted:

1. **User docs:** This is like a user manual for non-technical users. It would mostly focus on front-end applications and explain high-level processes for user understanding. This portion of the docs would be written at the end of the front-end development process.

2. **Code Docs:** This is a technical manual for developers and technical users. It will have a detailed listing of all the components in the project, including API endpoints, deployment strategies, configurations, extensions, etc. These portions of the docs will be written in parallel with the project's development.

During the project's first meeting, **Sphinx docs** [2] was approved. Sphinx is an open-source documentation tool **(BSD License)** which converts reStructured-Text (reST) files into HTML, PDF, and other formats. It supports extensions, automatic indexing, and integration with tools like ReadTheDocs for efficient documentation management.

Particularly for this project, I proposed the **Furo (MIT License)** theme. It's an aesthetically pleasing and simple theme for development and reading purposes (as shown in Fig. 8.1)

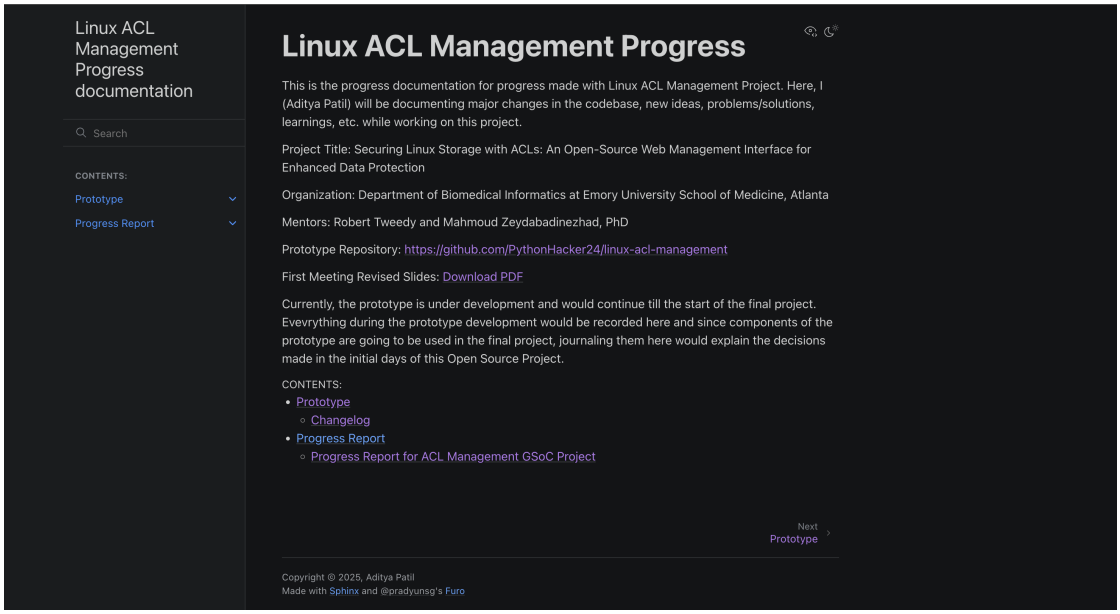

Figure 8.1: Sphinx Docs (Furo Theme)

Chapter 9

Motivation

Impact of the Project

The project started its roots in solving the problem of optimizing the file permission management system of BMI. The project has been planned to be kept open source for other organizations to modify, use, and distribute. This would allow them to optimize their own systems and speed up the pace of development, leading to improvements in their field - which is a rewarding experience. Due to its open-source nature, it would be available to everyone worldwide for free of cost, regardless of geography.

Why am I the right person to work on this?

I have been working on related projects since an early age, making the tech stack very familiar to me. My CV says a lot about my work experience and the work I have done in these 5 years, from cracking JEE with Physics, Chemistry, and Mathematics to working with people around the world to solve practical problems, from web3 to electronics security, from developing solutions to deploying production-level back-ends. I am strongly motivated to work on Open Source projects, especially those that enhance researcher productivity. As a researcher, it is rewarding to contribute to the betterment of the community.

Commitments During GSoC

During summer vacation (May 15, 2025, to July 15, 2025), I would be on summer vacation and have a complete day to work on the project.

In university working days, I would work on the project daily at routine times and be available for meetings anytime throughout the day. This is possible due to my flexible work schedule, which allows me to work consistently on long-term projects. My university classes span for 4-5 hours maximum (2 hours on good days), after which I work on my own stuff. On weekends, I would dedicate more time (Saturdays and Sundays are holidays). During the GSoC period, I will have 1-2 exams (which take 1 - 2 weeks to end), during which I will work lightly.

Post GSoC Commitments

My mentors will decide on my role after I have completed my GSoC. From my side, I will be more than happy to keep working on the project and become one of the project's maintainers, solving issues and managing community contributions.

Chapter 10

Acknowledgments

A special thanks to Mahmoud Zeydabadinezhad for steering me in the right direction, helping me navigate the complexities of this project, and arranging crucial meetings that familiarized me with the process. His mentorship has been instrumental in shaping my approach and execution.

I am immensely grateful to Robert Tweedy, whose patience and dedication shone through in his countless emails, ensuring I clearly understood the project. His willingness to address every query, no matter how small, has been the backbone of this proposal.

Also, to all the people who work countless hours on open-source projects for the betterment of society.

Glossary

Access Control Lists A mechanism for defining permissions more granularly than standard POSIX permissions.

Application Programming Interface A set of routines, protocols, and tools for building software and applications.

Berkeley Software Distribution (BSD) The BSD License is a permissive free software license that allows users to freely use, modify, and distribute software with minimal restrictions, while requiring attribution to the original authors.

GNU General Public License version 3 GPLv3 is a free software license that guarantees end users the freedom to run, study, share, and modify the software.

Graphical User Interface A user interface that includes graphical elements, such as windows, icons, and buttons.

Lightweight Directory Access Protocol A protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network.

Massachusetts Institute of Technology License The MIT License is a permissive free software license that allows users to use, copy, modify, merge, publish, distribute, sublicense, and sell copies of the software with minimal restrictions.

Portable Operating System Interface POSIX is a family of standards specified by the IEEE for maintaining compatibility between operating systems.

Principal Investigator Principal Investigator is the lead researcher for a particular well-defined research project.

System and Service Manager Systemd is a system and service manager for Linux operating systems.

Acronyms

ACLs Access Control Lists.

API Application Programming Interface.

BMI Biomedical Informatics.

GPLv3 GNU General Public License version 3.

GSoC Google Summer of Code.

GUI Graphical User Interface.

JSON JavaScript Object Notation.

JWT JSON Web Token.

LDAP Lightweight Directory Access Protocol.

MVP Minimum Viable Product.

PI Principal Investigator.

POSIX Portable Operating System Interface.

RHEL Red Hat Enterprise Linux.

SOM School of Medicine.

SystemD System and Service Manager.

Bibliography

[1] NISYSLAB. *Emory-BMI-GSoC*. 2024. Available at: https://github.com/NISYSLAB/Emory-BMI-GSoC (Accessed: 10 March 2025).

[2] Sphinx Documentation Team. *Sphinx: Python Documentation Generator*. Available at: https://www.sphinx-doc.org/en/master/ (Accessed: 10 March 2025).

[3] go-ldap. *LDAP Library for Go*. Available at: https://github.com/go-ldap/ldap (Accessed: 10 March 2025).

[4] IEEE and The Open Group. *POSIX.1-2017 — Standard for Information Technology — Portable Operating System Interface (POSIX®)*. Available at: https://pubs.opengroup.org/onlinepubs/9699919799/ (Accessed: 10 March 2025).

[5] Eiciel. *Graphical ACL editor for GNU/Linux*. Available at: https://rofi.roger-ferrer.org/eiciel/ (Accessed: 18 March 2025).

[6] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Available at: https://www.postgresql.org (Accessed: 10 March 2025).

[7] The Go Programming Language. *An open-source programming language*. Available at: https://go.dev/ (Accessed: 18 March 2025).

[8] *getfacl(1) - Linux Manual Page*. Available at: https://linux.die.net/man/1/getfacl (Accessed: 10 March 2025).

[9] *setfacl(1) - Linux Manual Page*. Available at: https://linux.die.net/man/1/setfacl (Accessed: 10 March 2025).

[10] Google Summer of Code. *Program Timeline*. Available at: https://developers.google.com/open-source/gsoc/timeline (Accessed: 18 March 2025).

[11] gRPC. *A high-performance, open-source universal RPC framework*. Available at: https://grpc.io/ (Accessed: 18 March 2025).

[12] YAML. *YAML Ain't Markup Language (YAML™) Specification*. Available at: https://yaml.org/ (Accessed: 18 March 2025).

[13] JWT. *JSON Web Tokens*. Available at: https://jwt.io/ (Accessed: 18 March 2025).