SOFTWARE ENGINEERING PROJECT REPORT

for

Academia Course Registration Portal

Prepared by:

1. Akshat Batra (IMT2023025)
2. Harshita Bansal (IMT2023035)

Github Link:
https://github.com/AkiBatra25/Academia_Portal

Submitted to:

Prof. Sujit Kumar Chakrabarti
Professor, IIIT Bangalore

December 7, 2025

Contents

1. Introduction

Course registration is a critical academic process in any university. In many institutes it is still handled using spreadsheets, emails or partially manual methods. Such approaches quickly become difficult to manage as the number of students and courses grows. Typical problems include clashing registrations, inconsistent data across different files, and a lack of proper access control. There is also a high dependency on administrative staff for routine operations which ideally could be automated.

To address these issues, we designed and implemented a **Client–Server Course Registration Portal**, called **Academia**. The system supports three distinct roles: **Admin**, **Faculty** and **Student**. Each role has its own set of permissions and operations. The project is implemented entirely in the C programming language and intentionally uses concepts from both the Operating Systems and Software Engineering courses. On the Operating Systems side, it uses TCP sockets, multi-threading using `pthreads`, file locking and binary file I/O. On the Software Engineering side, it follows a phase-wise lifecycle: requirements analysis, design, construction and testing.

This report consolidates the work done across all four phases:

- **Phase I – SRS:** Capturing functional and non-functional requirements.
- **Phase II – Design:** High level and detailed design using diagrams and module decomposition.
- **Phase III – Construction:** Implementation details and build/run instructions.
- **Phase IV – Testing:** White-box, black-box, unit, integration and system testing, including branch and path coverage.

The aim of this report is to give a clear picture of how the system evolved from a problem statement into a working, tested software artefact.

2. Phase I: Software Requirements Specification (SRS)

2.1 Purpose

The Software Requirements Specification (SRS) defines what the Academia portal is expected to do, and the constraints under which it must operate. It serves as a contract between the stakeholders (faculty, students, admin staff and evaluators) and the development team. The SRS is used as the primary reference for design, implementation and testing activities.

2.2 Scope of the System

The Academia portal aims to support a typical university course registration scenario. The main goals of the system are:

- To provide secure, role-based access for Admin, Faculty and Students.
- To centralize information about users, courses and enrollments.
- To allow Faculty to offer courses and manage course details.
- To allow Students to browse available courses, enroll and unenroll.
- To help the Admin manage user accounts and control access to the system.

The system is implemented as a terminal-based client that connects to a server over TCP. All persistent data is stored in binary files on the server side. The target environment is a Linux system with the GCC compiler.

2.3 User Characteristics

Admin: An administrative user who is responsible for creating and managing Student and Faculty accounts. The Admin is assumed to be technically comfortable with using command-line interfaces.

Faculty: Teaching staff who offer courses and manage course details. Faculty users are assumed to be moderately familiar with basic system operations.

Student: End users who register for courses. Students may not have deep technical knowledge, so the interface needs to be simple and menu-driven.

2.4 Functional Requirements

The core functional requirements are organized according to user role.

Admin Requirements

A1. The system shall allow the Admin to add new Student accounts by specifying a username and password.
A2. The system shall allow the Admin to add new Faculty accounts.
A3. The system shall provide options to view the list of all Students and all Faculty.
A4. The system shall allow the Admin to modify details of an existing Student or Faculty, such as username or password.
A5. The system shall support blocking and activating Student accounts. A blocked student must not be allowed to log in.
A6. The system shall allow the Admin to logout and return to the main login menu.

Faculty Requirements

F1. The system shall allow a Faculty member to view all courses currently offered by them.
F2. The system shall allow a Faculty member to create a new course by providing a course name and maximum number of seats.
F3. The system shall allow a Faculty member to update details of an existing course, such as the name or maximum seats.
F4. The system shall allow a Faculty member to remove a course from the catalog.
F5. The system shall allow a Faculty member to change their own password.
F6. The system shall allow the Faculty member to logout and return to the main login menu.

Student Requirements

S1. The system shall allow a Student to view all available courses in the catalog, along with current and maximum seat counts.
S2. The system shall allow a Student to enroll in a course by providing a valid course ID, subject to seat availability.

S3. The system shall prevent a Student from enrolling in the same course more than once.
S4. The system shall allow a Student to view the list of courses they are currently enrolled in.
S5. The system shall allow a Student to unenroll from a course in which they are currently enrolled.
S6. The system shall allow a Student to change their login password.
S7. The system shall allow the Student to logout and return to the main login menu.

Authentication Requirements

- The system shall authenticate users based on username, password and role (Admin, Faculty or Student).
- The system shall display an informative error message when the credentials or role are incorrect.
- A blocked Student must not be allowed to log in until the Admin activates the account again.

2.5 Non–Functional Requirements

Usability

The system must provide a clear, menu-driven interface that can be operated using simple numeric choices. Prompts and error messages should be concise and unambiguous.

Performance

Under normal load (a few concurrent clients), each operation such as login, viewing courses or enrolling must complete within a few seconds. Data access is optimized by using binary files.

Security

Security is handled mainly through role-based access control and authentication. Only Admin users can create or modify user accounts. Students and Faculty can modify only their own passwords.

Reliability

The system must maintain consistent data even when multiple clients are connected simultaneously. File locking is used to avoid race conditions and data corruption during updates.

Portability

The application targets Linux-based systems and requires only a C compiler and standard libraries. No external database or web server is required, which makes the system easy to deploy in lab environments.

3. Phase II: System Design

3.1 Overall Architecture

The system follows a modular client–server architecture. The client process provides the user interface, while the server process contains all business logic and manages access to the data files. The server listens on a fixed TCP port and accepts incoming connections. For each connected client, the server creates a new thread that handles the complete interaction with that client.

Data is stored in binary files on the server side. This design keeps the client thin and stateless, while the server acts as a centralized controller and data manager.

3.2 Module Decomposition

The implementation is divided into the following logical modules:

- **Client module:** Responsible for connecting to the server, sending user input, and displaying responses.
- **Server main module:** Creates the listening socket, accepts client connections and spawns handler threads.
- **Authentication module:** Implements the `authenticate()` function that validates username, password and role.
- **Admin module:** Contains functions for adding, viewing, modifying and blocking/activating users.
- **Faculty module:** Contains functions for course creation, update and deletion.
- **Student module:** Contains functions related to viewing courses, enrollment and unenrollment.
- **File management module:** Provides helper functions to read and update binary files with proper locking.

Each module communicates with others using well-defined function interfaces, which makes the code easier to maintain and extend.

3.3 Data Structures

The core entities are represented using C `structs`, for example:

- **User:** stores ID, username, password, role and active/blocked status.
- **Course:** stores course ID, name, faculty ID, maximum seats and current enrollment count.
- **Enrollment:** stores a pair of student ID and course ID.

These structures are written to and read from binary files, making the system lightweight and independent of any external database.

3.4 Design Decisions

Several design choices were made intentionally:

- The system uses a plain-text terminal interface instead of a graphical UI, in order to focus on OS-level and SE-level concepts rather than front-end design.
- Binary files are used instead of SQL databases to keep deployment simple and to provide practice with file operations and locking.
- A thread-per-client model is used because it is straightforward to implement and clearly demonstrates concurrency.
- Role-based menus are implemented on the server side so that basic security policies are enforced centrally.

4. Phase III: Construction and Implementation

4.1 Technologies Used

The project is implemented in the C programming language on a Linux platform. Key technologies and libraries include:

- **TCP Sockets:** for network communication between client and server.
- **pthreads:** for handling multiple clients concurrently.
- **Standard C file I/O:** for reading and writing binary data files.
- **fcntl-based file locking:** to provide mutual exclusion when updating shared files.

4.2 Directory Structure

A simplified view of the project directory is:

```
Academia_Portal/
    client/      # Client-side C program
    server/      # Server-side C program and modules
    include/     # Header files
    init/        # Data initialization utility
    tests/       # Automated tests and unit test code
    README.md
    Testing.md
```

4.3 Build and Execution Instructions

To make the project easy to reproduce, the following commands can be used.

Build the Server

```
gcc -Iinclude -o Server/server Server/server.c Server/common.c
```

Build the Client

```
gcc -Iinclude -o Client/client Client/client.c
```

Initialize Sample Data (Optional)

```
gcc -include -o init/init_data init/init_data.c
./init/init_data
```

Run the Server

```
./server/server
```

Run the Client

In a different terminal:

```
./client/client
```

The client will connect to the server and display the main login menu. From this point, the user can log in as Admin, Faculty or Student and use the respective functionalities.

5. Phase IV: Testing Phase

Testing is a crucial part of the Software Engineering lifecycle. For this project, we applied a combination of white-box and black-box techniques and covered multiple levels of testing: unit, module, integration and system. In addition, we explicitly considered branch coverage and path coverage for some key functions.

5.1 Testing Objectives

The testing phase had the following objectives:

- Verify that each major function behaves correctly for valid inputs.
- Ensure that invalid inputs and corner cases are handled gracefully.
- Confirm that separate modules work together correctly when integrated.
- Check that the system as a whole supports realistic usage scenarios.
- Achieve good branch coverage and basic path coverage for critical logic such as authentication and enrollment.

5.2 Types of Testing

5.2.1 White-Box Testing

White-box testing was mainly applied to the functions `authenticate()`, `enroll_course()` and `unenroll_course()`. Since the internal code was available, we identified all major decision points and designed test cases such that each branch of an `if` statement was taken at least once across all tests.

For example, in `authenticate()`, the decision to accept or reject a login depends on three checks: whether the username matches, whether the password matches and whether the role matches. We created separate test cases for each combination where one of these checks fails, and one case where all three succeed. This ensured that all true and false outcomes of the internal conditions were exercised.

5.2.2 Black-Box Testing

Black-box testing focused on the external behaviour of the system. We treated the client–server application as a black box and designed tests based only on the requirements, without looking at the code. Typical scenarios included:

- Logging in with correct and incorrect credentials.
- Students enrolling in available courses.
- Students attempting to enroll in a full course.
- Admin trying to block and activate a student.
- Faculty adding a course and observing its visibility on the student side.

5.2.3 Unit Testing

Unit tests were written for the `authenticate()` function in a separate file `tests/test_authenticate.c`. The unit test calls `authenticate()` directly with various usernames, passwords and roles, and checks whether the returned result matches the expectation. This test does not involve sockets or menus and therefore isolates the logic being tested.

5.2.4 Module Testing

Module testing was performed after unit testing and focused on groups of related functions. For example, all Admin-related functions were tested together by executing a sequence of operations: adding a student, viewing the student list, modifying the student's details and verifying that the login still works with the updated credentials.

Similarly, the Student module was tested by running through the sequence: view all courses, enroll in a course, view enrolled courses and then unenroll. We monitored the changes in the data files to ensure that the internal state remained consistent.

5.2.5 Integration Testing

Integration testing examined the interactions between the Admin, Faculty and Student modules and the shared data files. We designed scenarios such as:

- Admin adds a new student, and then that student logs in and enrolls in a course.
- Faculty adds a new course, and a student subsequently enrolls in that course.
- A student enrolls in multiple courses and then the Admin blocks the student; after blocking, the student should no longer be able to log in.

These tests confirmed that data added by one role is visible and usable by the others and that restrictions like blocking are properly enforced.

5.2.6 System Testing

System testing treated the entire client–server application as a single unit. We ran the server and one or more clients, then executed complete workflows from logging in to logging out. We also tested edge cases such as entering an invalid menu choice, logging out suddenly and restarting the client while the server is still running.

5.3 Test Case Summary

Table 5.1 summarises representative test cases used during the testing phase.

Test Case ID	**Description**	**Input**	**Expected Output**
TC-Auth-01	Valid Admin login	Username = admin1, Password = admin1pwd, Role = admin	Login successful, Admin menu displayed
TC-Auth-02	Wrong password	Username = admin1, Password = wrong, Role = admin	Error message "Invalid credentials" and return to login menu
TC-Auth-03	Wrong role	Username = admin1, Password = admin1pwd, Role = student	Error message and login denied
TC-Stu-01	Student enrolls in valid course	Student selects "Enroll" and enters ID of an existing course with free seats	Enrollment succeeds, and the course appears in the list of enrolled courses
TC-Stu-02	Duplicate enrollment	Student tries to enroll in the same course twice	Second attempt is rejected with an "Already enrolled" message
TC-Stu-03	Unenrollment	Student chooses "Unenroll" and enters ID of a course they are enrolled in	Enrollment record is removed and course disappears from enrolled list
TC-Fac-01	Faculty adds course	Faculty selects "Add New Course" and enters a valid course name and seat count	New course appears in "View Offering Courses" and is visible to students

TC-Adm-01	Admin adds student	Admin enters new student's username and password	Student record is created and the new student can log in using those credentials

Table 5.1: Representative test cases for the Academia portal

5.4 Branch and Path Coverage

authenticate()

For the `authenticate()` function, we identified the following major branches:

- Branch 1: user record found vs. user record not found.
- Branch 2: password matches vs. password does not match.
- Branch 3: role matches vs. role does not match.

By designing test cases for each combination, we ensured that:

- There is at least one test case where the username is not present in the file.
- There is a test case where the username exists but the password is wrong.
- There is a test case where both username and password match but the role is different.
- There is a test case where all three fields match and the function returns success.

This gives us near-complete branch coverage and good path coverage for this function.

enroll_course()

For `enroll_course()`, the important branches are:

- The course ID exists in the course file vs. it does not exist.
- The student is already enrolled vs. not yet enrolled.
- The course has reached maximum capacity vs. seats are available.

We created separate test cases to cover each of these conditions:

- Enrolling in a valid course with free seats (success path).

- Attempting to enroll in a course with no remaining seats (course full path).
- Attempting to enroll again in a course the student is already enrolled in (duplicate path).
- Entering a non-existent course ID (course not found path).

unenroll_course()

In `unenroll_course()`, we distinguish between the case where a matching enrollment record is found and the case where the student is not enrolled in the specified course. We also ensure that the course seat count is decremented only when the unenrollment actually happens. Test cases were written for both success and failure scenarios.

5.5 Automated Test Scripts

To make testing repeatable, a set of shell scripts was created in the `tests/` directory. These scripts connect to the running server and pipe predetermined input into the client program, simulating user behaviour. For example:

- `run_demo_test.sh` builds the project, initializes data, starts the server and runs a demo client interaction.
- `test_invalid_login.sh` automatically performs a login attempt with incorrect credentials and checks the error message.
- `test_student_enroll_unenroll.sh` walks through a complete student workflow: login, view courses, enroll, unenroll and logout.
- The C unit test `test_authenticate.c` is compiled into a small executable that prints PASS/FAIL for each authentication case.

These scripts demonstrate that testing is not only manual but also automated, which is an important aspect of Software Engineering practice.

5.6 How to Run Tests

This section provides complete instructions for compiling the project and running all automated, unit, and manual test cases. All commands shown below must be executed from the project root directory:

```
Academia_Portal$
```

—

5.6.1 Building the Server and Client

Before running any tests, the server and client must be compiled.

Build the Server

```
gcc -Iinclude -o Server/server Server/server.c Server/common.c
```

Build the Client

```
gcc -Iinclude -o Client/client Client/client.c
```

(Optional) Initialize Sample Data

This step populates the data files with example Admin, Faculty, Student, and Course entries.

```
gcc -Iinclude -o init/init_data init/init_data.c
./init/init_data
```

—

5.6.2 Demo Test Script (End-to-End System Test)

A demo script is provided to automatically build the project, initialize default data, start the server, and run a sample client interaction.

Run Demo Test

```
chmod +x tests/run_demo_test.sh
./tests/run_demo_test.sh
```

This demonstrates a full client–server execution without requiring manual input.

—

5.6.3 Running Black-Box Automated Test Scripts

To execute the black-box test scripts, start the server in one terminal and run test scripts from another terminal.

Step 1: Start the Server (Terminal 1)

```
./Server/server
```

You should see:

```
Server listening on port 8080...
```

Step 2: Run Test Scripts (Terminal 2)

(a) Invalid Login Test

```
chmod +x tests/test_invalid_login.sh
./tests/test_invalid_login.sh
```

This script attempts a login using incorrect credentials and verifies that the server returns an appropriate error.

(b) Student Enrollment + Unenrollment Workflow

```
chmod +x tests/test_student_enroll_unenroll.sh
./tests/test_student_enroll_unenroll.sh
```

This test automates:

- Student login,
- Viewing available courses,
- Enrolling in a course,
- Unenrolling from that course,
- Logging out.

(c) Admin Adds a New Student

```
chmod +x tests/test_admin_add_student.sh
./tests/test_admin_add_student.sh
```

This verifies:

- Correct admin authentication,
- Adding a new student record,
- Displaying updated student list.

—

5.6.4 Unit Testing of authenticate() (White-Box Testing)

Unit-level white-box testing is implemented using a dedicated C test file.

Compile the Unit Test

```
gcc -Iinclude -o tests/test_authenticate tests/test_authenticate.c Server/common.c
```

Run the Unit Test

```
./tests/test_authenticate
```

The program prints individual PASS/FAIL outcomes for:

- Valid Admin login,
- Invalid password,
- Wrong role selection,
- Valid Student login.

This ensures branch coverage and basic path coverage for the `authenticate()` function.

—

5.6.5 Manual Testing for Remaining Menu Features

Some menu features require manual validation due to interactive workflow.

Procedure

1. Start the server:

   ```
   ./Server/server
   ```

2. Start the client:

   ```
   ./Client/client
   ```

3. Log in (Admin / Faculty / Student).
4. Execute menu operations such as:
   - Modify Student / Faculty details,
   - Update course information,
   - Remove a course,
   - Activate/Block a student.
5. Verify correctness through:
   - On-screen success/error messages,
   - Updated entries in `users.dat`, `courses.dat`, and `enrollments.dat`.

All manual test steps are documented in the `Testing.md` file under the "Manual Test Cases" section.

6. Conclusion and Future Work

In this project, we implemented a complete course registration portal using a client–server architecture. From a Software Engineering perspective, we followed a structured lifecycle with clearly separated phases: SRS, design, construction and testing. From an Operating Systems perspective, we worked with sockets, threads, file management and synchronization.

The system successfully supports Admin, Faculty and Student roles and demonstrates key features such as authentication, course offering and registration, and data persistence across sessions. Thorough testing, including branch and path coverage for critical functions, gives reasonable confidence in the correctness and robustness of the system.

There are several directions in which the project could be extended:

- Replacing binary files with a relational database for more complex queries.
- Adding a web-based or graphical front end while keeping the existing server logic.

Overall, the project provided practical experience in designing, implementing and testing a non-trivial software system using concepts taught in both Operating Systems and Software Engineering courses.