Building a Console-Based Library Management System in C++

Alfie Atkinson
University of Lincoln
25715017@students.lincoln.ac.uk

***Abstract*—This paper presents the design and implementation of a console-based library management system developed in C++. The system encompasses core functionalities such as book and user management, facilitating the addition, removal, and searching of library resources. It incorporates advanced features including borrowing and returning operations, data persistence through a database, multi-threading for efficient concurrency, and networking support for remote access. Additionally, there is potential for future enhancements such as advanced search capabilities, enhanced user management, and improved data visualisation. Through a modular and well-structured architecture, the Library Management System ensures seamless interaction between its components, providing a robust solution for managing library resources effectively.**

I. Introduction

The purpose of this report is to present the design and implementation of a console-based Library Management System (LMS) developed in C++. This system aims to manage a library's operations efficiently, including cataloging books, managing user accounts, and handling the borrowing and returning of books. The project demonstrates the use of various programming concepts such as object-oriented programming (OOP), low-level memory management, input/output operations, multi-threading, and networking.

The LMS is designed to function as an offline application, with a potential for future extensions to support network and multi-threading functionalities. The system's core features include a user-friendly console interface, persistent data storage, and robust concurrency management.

The codebase for this project can be found on GitHub [1].

A. Project Overview

The Library Management System provides a comprehensive solution for managing library resources and user interactions. Key functionalities include:

- CRUD Operations: Manage books, users, and transactions.
- Borrow and return books: Tracking the lending status of books.
- Remote Access: Server/client architecture for remote access using sockets.
- Multi-threading: Threaded actions for concurrent user access.
- Data persistence: JSON for storing and retrieving data.

A more comprehensive overview can be seen in the project timeline (see Figure 1 in the appendix).

II. System Design

This section details the system architecture and design choices, illustrated by the diagrams in the appendix.

A. UML Diagrams

The system design is visually represented through UML diagrams to provide a clear understanding of the structure and interactions within the LMS. The class diagram (see Figure 2) outlines the main classes and their relationships. The use case diagram (see Figure 3) illustrates the interactions between the system and its actors (users, librarians, and admins).

B. Design Description

The architecture of the Library Management System (LMS) adopts a modular approach, emphasizing separation of concerns to enhance maintainability, scalability, and flexibility. Each component of the system is designed to handle specific responsibilities, ensuring a clear division between various aspects of functionality. Unit tests and Test-Driven Development were used to validate the core functionality of the system.

Each class collaborates with others to perform the required functionalities, ensuring a cohesive and robust system. The detailed interactions between these classes are visually represented in the class diagram (see Figure 2).

III. Implementation

This section outlines the coding and development process for the Library Management System, focusing on its core classes, advanced features, and implementation strategies.

A. Core Classes and Functionalities

The core classes implemented in the system include:

- **Application**: Manages the application's main flow, including user authentication, menu navigation, and book, user, and transaction management.
- **Book**: Represents a book with attributes like title, author, ISBN, year published, and availability. Includes methods for borrowing and returning books.
- **Database**: Manages the storage and retrieval of books, users, and transactions. Provides methods for CRUD operations and data persistence.
- **LibraryManager**: Acts as the main interface between the application and the database, handling book borrowing/returning, user authentication, and data queries.

- **Menu**: Handles the display and interaction of menu options, including pagination and user input management.
- **PersistenceManager**: Manages saving and loading the database to and from a file, ensuring data persistence.
- **Server**: Manages networking by handling client connections and communication with the application, enabling remote access.
- **ThreadManager**: Manages background tasks such as periodic data saving and client threading for concurrent operations.
- **Transaction**: Represents a book transaction with details like type (borrow/return), status, and associated book and user. Includes methods for executing and canceling transactions.
- **User**: Represents a library user with attributes like username, forename, surname, email, phone number, and password. Includes methods for borrowing and returning books and checking out status.

The classes were implemented using object-oriented principles and low-level memory management. For example, `std::shared_ptr` was used for efficient memory allocation and to manage book and user objects dynamically.

B. Advanced Features

The system incorporates advanced features such as multi-threading and networking to enable concurrent access and remote operations.

The system includes a `Server` class that manages client connections and communicates with the `Application` class. This allows multiple clients to interact with the system remotely. It uses the `std::thread` library to allow multiple clients to connect and interact simultaneously. Synchronisation mechanisms, such as mutexes, are used to ensure thread safety and prevent race conditions.

Thread-safe operations were implemented for shared resources, such as accessing the database, to maintain data integrity in a multi-user environment.

IV. Scalability & Future Work

The system is designed for small- to medium-scale use, but scalability challenges may arise as concurrent users and the size of the database increase. A growing dataset of books, users, and transactions could lead to slower query response times, especially for complex operations like searching or updating multiple records.

As the number of concurrent users rises, server performance may degrade due to the increased demand for processing power, memory, and network resources. High volumes of client requests can lead to bottlenecks, particularly in single-threaded or poorly optimised server environments.

Additionally, the current JSON-based persistence system may struggle to handle large datasets efficiently, resulting in higher latency and increased chances of data corruption under heavy load.

A. Proposed Solutions

To address these challenges:

- **Load Balancing**: Distribute client requests across multiple servers to prevent bottlenecks and ensure consistent performance.
- **Database Optimisation**: Use indexing on frequently queried fields, such as ISBN or user credentials, to improve query efficiency. Streamline the database schema to minimise redundancy.
- **Caching**: Implement caching mechanisms (e.g., Redis) to reduce database load and improve response times for frequently accessed data.
- **Refined Search Method**: As the library grows, implement features like fuzzy search, keyword ranking, and advanced filters (e.g., by genre, author, or availability) to improve user experience and minimise unnecessary database queries.
- **Transition to SQL**: Replace JSON-based file persistence with a scalable SQL database like PostgreSQL, which supports large datasets and efficient queries.

B. Security: Password Encryption

Secure user authentication is achieved by hashing passwords with algorithms like bcrypt or Argon2. To enhance security:

- **Salting**: Add unique salts to passwords before hashing to prevent duplicate hashes.
- **Secure Storage**: Store hashed passwords securely in an encrypted database.
- **Key Rotation**: Periodically rotate encryption keys to reduce the impact of potential breaches.

These measures ensure user data remains secure even if the database is compromised.

C. Client-Side Application Logic

Offloading some application logic to the client can reduce server load and improve scalability:

- **Input Validation**: Perform basic validation on the client side to minimise unnecessary server requests.
- **Local Caching**: Use browser storage options like IndexedDB or LocalStorage for session data and frequently accessed content.
- **Web APIs**: Use RESTful APIs to shift data processing and rendering to the client, reducing server-side overhead.

Delegating non-critical tasks to the client improves scalability and enhances user experience with faster responses and lower latency.

References

[1] A. Atkinson, *Library management system*, 2025. [Online]. Available: https://github.com/alfieatkinson/Library-Management-System.

Appendix

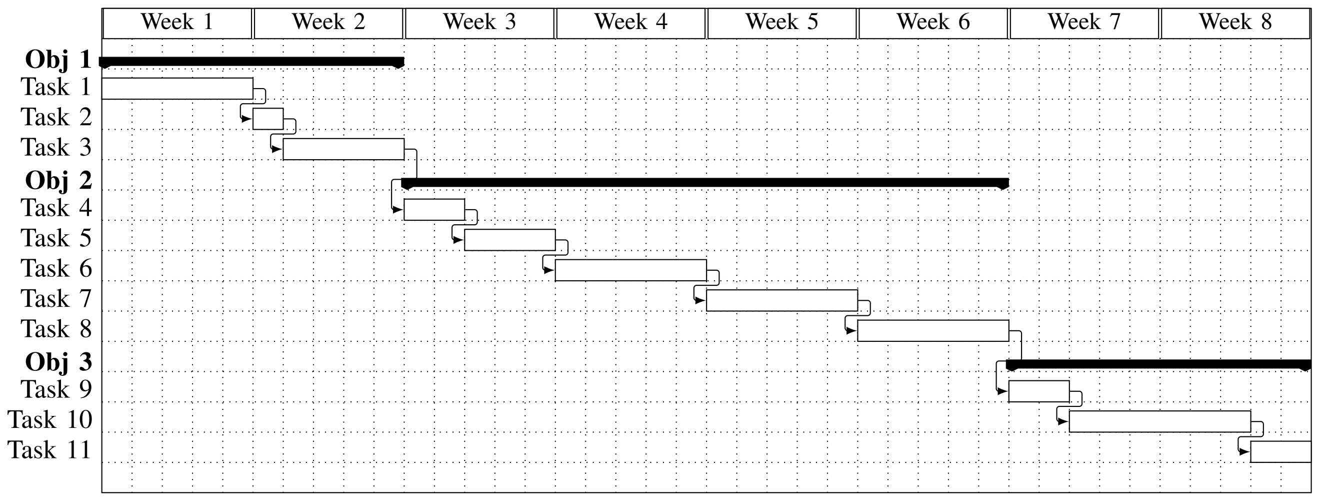

Fig. 1. Gantt Chart for Project Timeline and Task Dependencies

Task	Description	Task	Description	Task	Description
Obj 1: Planning, Design & Test Writing		**Obj 2: Core Functionality Development**		**Obj 3: Final Testing & Documentation**	
1	System Design (UML Diagrams)	4	Book & User Management	9	Final Testing & Debugging
2	Set Up Development Environment	5	Borrow & Return Books	0	Report Writing & Final Touches
3	Write Tests (TDD)	6	Persistence for Saving & Loading Data	11	Proof Reading & Improving
		7	Multi-Threading for Concurrent Ops		
		8	Networking for Remote Access		

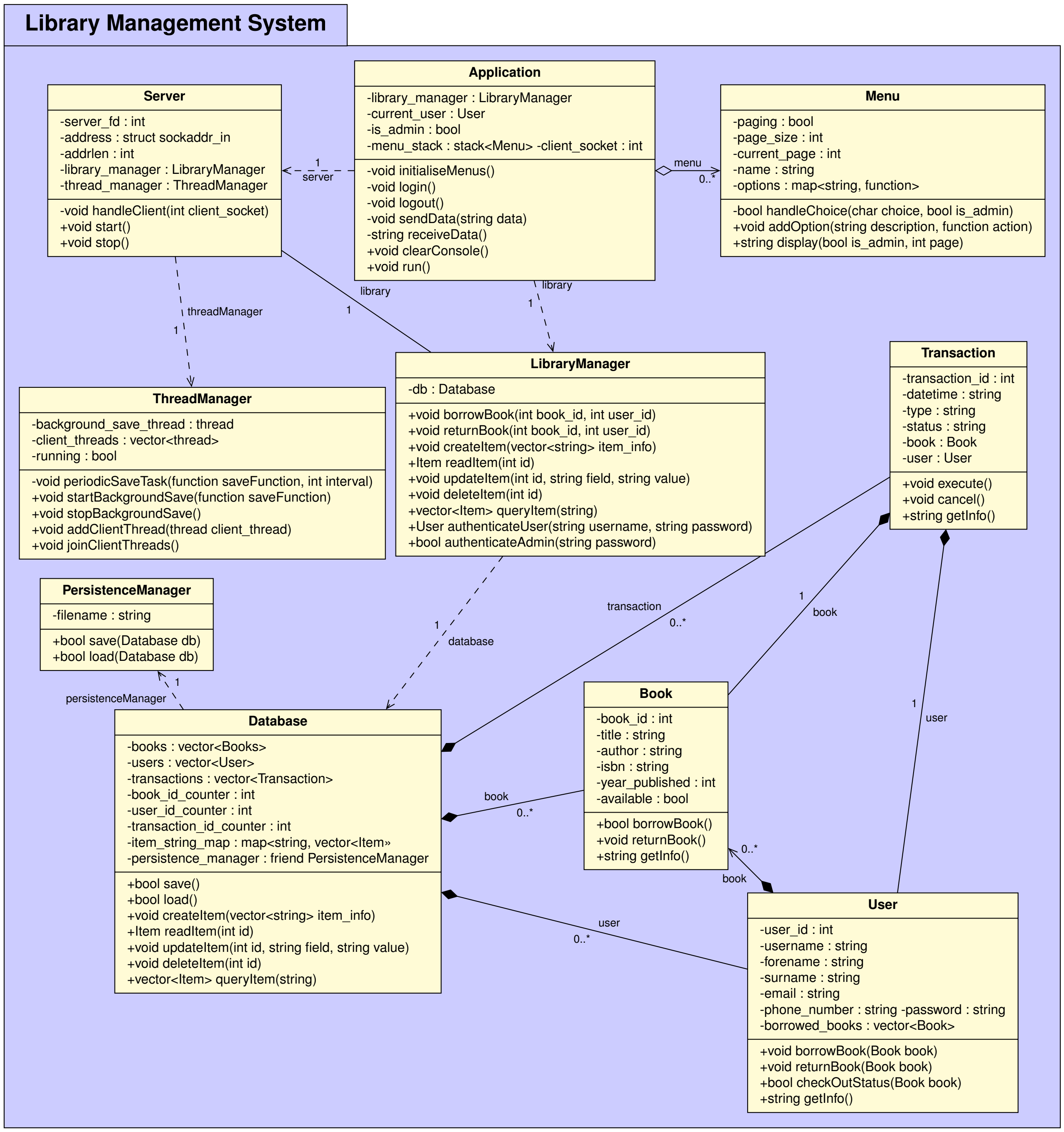
Library Management System
Application
-library_manager : LibraryManager
-current_user : User
-is_admin : bool
-menu_stack : stack<Menu> -client_socket : int
-void initialiseMenus()
-void login()
-void logout()
-void sendData(string data)
-string receiveData()
+void clearConsole()
+void run()
Server
-server_fd : int
-address : struct sockaddr_in
-addrlen : int
-library_manager : LibraryManager
-thread_manager : ThreadManager
-void handleClient(int client_socket)
+void start()
+void stop()
Menu
-paging : bool
-page_size : int
-current_page : int
-name : string
-options : map<string, function>
-bool handleChoice(char choice, bool is_admin)
+void addOption(string description, function action)
+string display(bool is_admin, int page)
1
server
menu
0..*
threadManager
1
library
1
library
1
ThreadManager
-background_save_thread : thread
-client_threads : vector<thread>
-running : bool
-void periodicSaveTask(function saveFunction, int interval)
+void startBackgroundSave(function saveFunction)
+void stopBackgroundSave()
+void addClientThread(thread client_thread)
+void joinClientThreads()
LibraryManager
-db : Database
+void borrowBook(int book_id, int user_id)
+void returnBook(int book_id, int user_id)
+void createItem(vector<string> item_info)
+Item readItem(int id)
+void updateItem(int id, string field, string value)
+void deleteItem(int id)
+vector<Item> queryItem(string)
+User authenticateUser(string username, string password)
+bool authenticateAdmin(string password)
Transaction
-transaction_id : int
-datetime : string
-type : string
-status : string
-book : Book
-user : User
+void execute()
+void cancel()
+string getInfo()
PersistenceManager
-filename : string
+bool save(Database db)
+bool load(Database db)
transaction
0..*
1
book
1
database
1
user
1
persistenceManager
Database
-books : vector<Books>
-users : vector<User>
-transactions : vector<Transaction>
-book_id_counter : int
-user_id_counter : int
-transaction_id_counter : int
-item_string_map : map<string, vector<Item»
-persistence_manager : friend PersistenceManager
+bool save()
+bool load()
+void createItem(vector<string> item_info)
+Item readItem(int id)
+void updateItem(int id, string field, string value)
+void deleteItem(int id)
+vector<Item> queryItem(string)
Book
-book_id : int
-title : string
-author : string
-isbn : string
-year_published : int
-available : bool
+bool borrowBook()
+void returnBook()
+string getInfo()
book
0..*
0..*
book
user
0..*
User
-user_id : int
-username : string
-forename : string
-surname : string
-email : string
-phone_number : string -password : string
-borrowed_books : vector<Book>
+void borrowBook(Book book)
+void returnBook(Book book)
+bool checkOutStatus(Book book)
+string getInfo()

Fig. 2. Class diagram for the Library Management System, including key classes for multithreading, networking, and persistence.

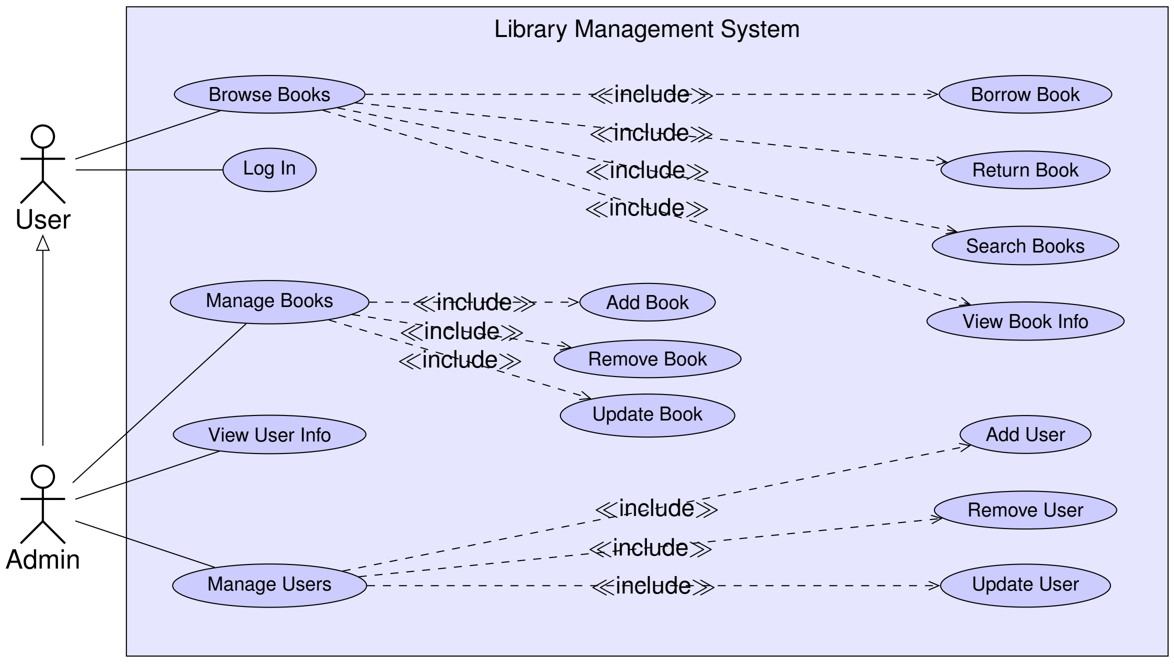

Fig. 3. Use Case Diagram for the Library Management System showing the interactions between Users and Admins