ICE: Intelligent Course notes Executive

CSC3002 Group Project
Baicheng Chen, Danyang Chen, Shuhan Zhang, Weiqi Mao, Yuetong Chen

1. Introduction

Motivation

In recent years, students have relied heavily on various software tools to study. For example, they have to download materials from platforms like BlackBoard, take notes using applications like Goodnotes, ask Chat GPT school-related questions, or discuss them in messaging apps like Slack or WeChat. Such reliance on multiple applications significantly increases the complexity of the study.

Goal

To save students' time and enhance their learning experiences, we propose an all-in-one learning application ICE : Intelligent Course notes Executive. It will have the following features:

(1) comprehensive course search and management

(2) resources sharing and downloading

(3) a chat room supporting text, likes and dislikes interactions

(4) AI assistant

Limitations of related work

While numerous educational websites and applications are available, most focus on distinct features. What's more, few all-in-on applications cater to the diverse needs of students. Although the Chinese University of Hong Kong, Shenzhen, has proposed "Fries Artificial Intelligence" with similar goals to ours, it is unstable and may encounter issues such as failing to load files or the AI being unable to function properly. What's more, "Fries Artificial Intelligence"'s ai, chatgpt 4o, can only one file at one time, while our ai support reading mutilple files and giving respond to them, facilitating generating answer for the hole course with mutil-files. Our application ICE, developed in C++, is faster in performance speed. It also supports customizing users' workspace and manage their own files.

Significance

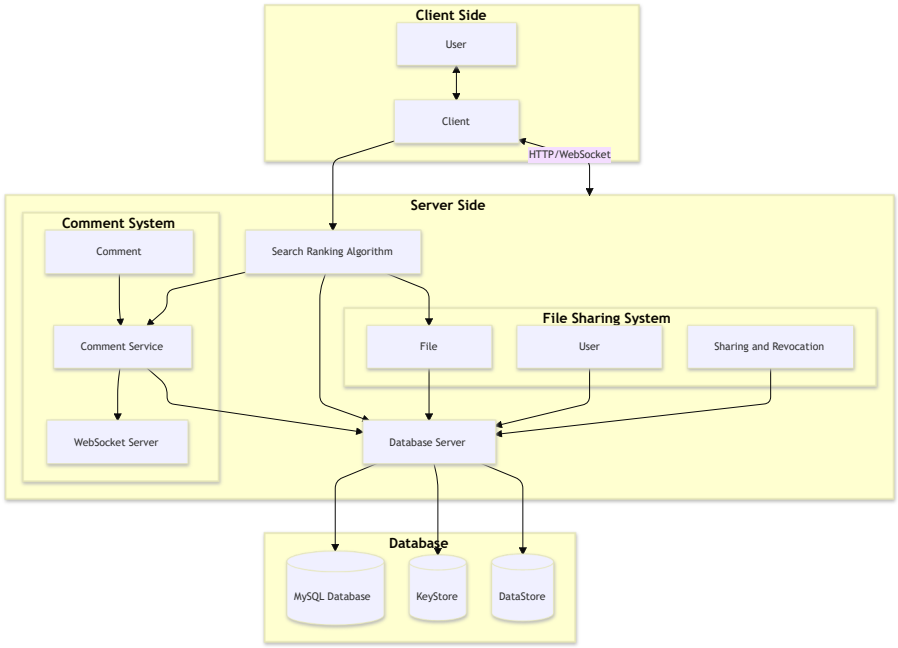
The significance of our project lies in its user-centered design, optimizing resource search, convenient file-exchanging process, and question resolution. It dramatically simplifies students' learning journeys and conserves their time. Students will no longer need to toggle between multiple websites to acquire knowledge. Our application will become an intelligent companion in the academic process, making learning more accessible and enjoyable for all students.

2. Related Work

> 1. *Fries Artificial Intelligence*
>
> *Fries Artificial Intelligence is a website that focuses on helping students at the Chinese University of Hong Kong, Shenzhen, to read PPT. It uses AI to translate every PPT to help the students to understand it. However, it does not support creating your workspace and managing related files. It can only search on a course basis, while our project supports searching for file names and adding those files into the users' workspace for a specific subject. Our project and Fries Artificial Intelligence are both all-in-one study platforms. Fries Artificial Intelligence focuses on PPT translation, while our project focuses on file exchanging.*
>
> 2. *CS161 (Computer Security) Project 2 from UC Berkeley*
>
> *This project aims to use some provided cryptographic library functions to design a secure file-sharing system using the programming language **Go**. The essential functions are to log in, store files, and share files with others while in the presence of attackers.*

3. Our work Components

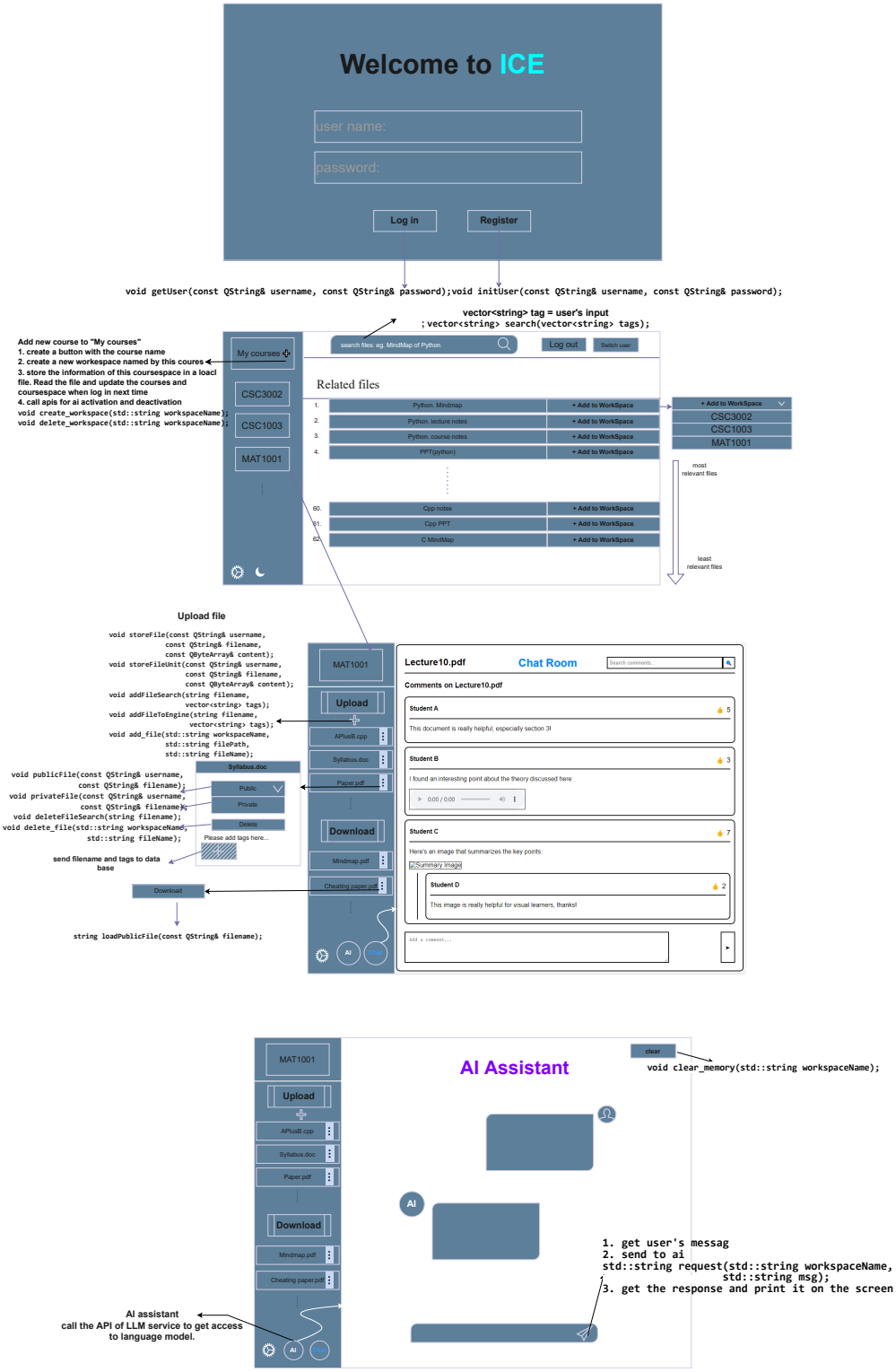
System Architecture Diagram



1. UI (User Interface)

The UI includes three parts.

1. log in & register interface. (Class MainWindow)
2. Users' courses. (Class Courses)
3. Workspace for a specific course (Class CourseSpace)
   (a) upload and download files
   (b) workspace's Ai assistant (Class AIAssistant)
   (c) workspace's chat room (Class ChatRoom)



The first part collects basic information from the user, and call our database to get or stored the user's information.

The second part has (a) a list of courses the user created and (b) a search space for the user. The course's name is set by the user. Files can be searched and added to specific courses.

The third part is a workspace for a specific course, including various files. Users can use AI assistant to solve questions or discuss particular files in the chat room.

2. **Client Backend**

File Sharing System

To build a whole process of sharing files, we consider three parts: 1) Users and User Authentication, 2) File Operations, and 3) Sharing and Revocation. To achieve the three parts, we construct 8 main functions:

1) Users and User Authentication

- `InitUser(string username, string password)`: for user registration. Users input their usernames and passwords, and their personal information will be uploaded to the Database through structs.
- `GetUser(string username, string password)`: for user log in. Users input their usernames and passwords, and the system identifies them and retrieves their personal information from the Database.

2) File Operations

- `StoreFile(string filename, vector<uint8_t> content)`: for storing files into Database.
- `LoadFile(string filename)`: for loading (retrieving) files from Database.
- `AppendToFile(string filename, vector<uint8_t> content)`: for adding contents after the original files.

3) Public and Private

- `PublicFile(string filename)`: for user sharing files. If the user set a file into public, the file will appear in the search page, and other users can retrieve and download the shared file.
- `PrivateFile(string filename)`: for user stopping sharing files. If the user set a file back to private (the default stage when a user upload a file is private), other users cannot see and retrieve the file through search page, only the owner can retrieve and operate the file.
- `LoadPublicFile(string filename)`: for loading the public file shared by users.

4) Sharing and Revocation (Not implement in project yet but in code, can be future expansion functions.)

- `CreateInvitation(string filename, string recipientUsername)`: for senders to create invitations to let the recipients be able to read and load the files.
- `AcceptInvitation(string senderUsername, UUID invitationPtr, string filename)`: for recipients to accept the invitation to view the file the senders sent. The recipients can create their own name for the file in the individual's workspace.
- `RevokeAccess(string filename, string recipientUsername)`: for the original senders (the original file owner) to revoke some recipients' permissions (if the recipients/senders want to delete the file or the senders want to set the file to be private).

To deal with the problem that the socket protocol can transfer up to 65278B of data, we use the "linked-list" structure to store the file content. First, we create a `File` struct to store the meta info of the whole file containing the first block of content (the uuid of the first `FileContent` struct). Then, we create several `FileContent` struct and linked them through a `Next` uuid (act as pointer), each `FileContent` can store 10000B of data. This strategy solve the transfer limitation problem.

The Database for this section will be divided into two parts: Keystore and Datastore. Keystore is a remote database used to store public keys for users to do encrypt-decrypt operations and verify the integrity and authenticity. The content we stored in the Keystore is a name-value pair, mapping the string to the key value. Datastore is also a database that stores data (user information, file content, invitation information). The content we stored in the Datastore is a name-value pair. Name in the pair is a UUID, which maps the content (value) we want to store in the Datastore.

Also, we want to build a secure file-sharing system, so we will use some cryptography methods like hash, symmetric-key encryption, hmac, public-key encryption, digital signatures, password-based key derivation function (PBKDF), hash-based key derivation function (HBKDF) to construct a protection web, protecting our system from predictable attacks.

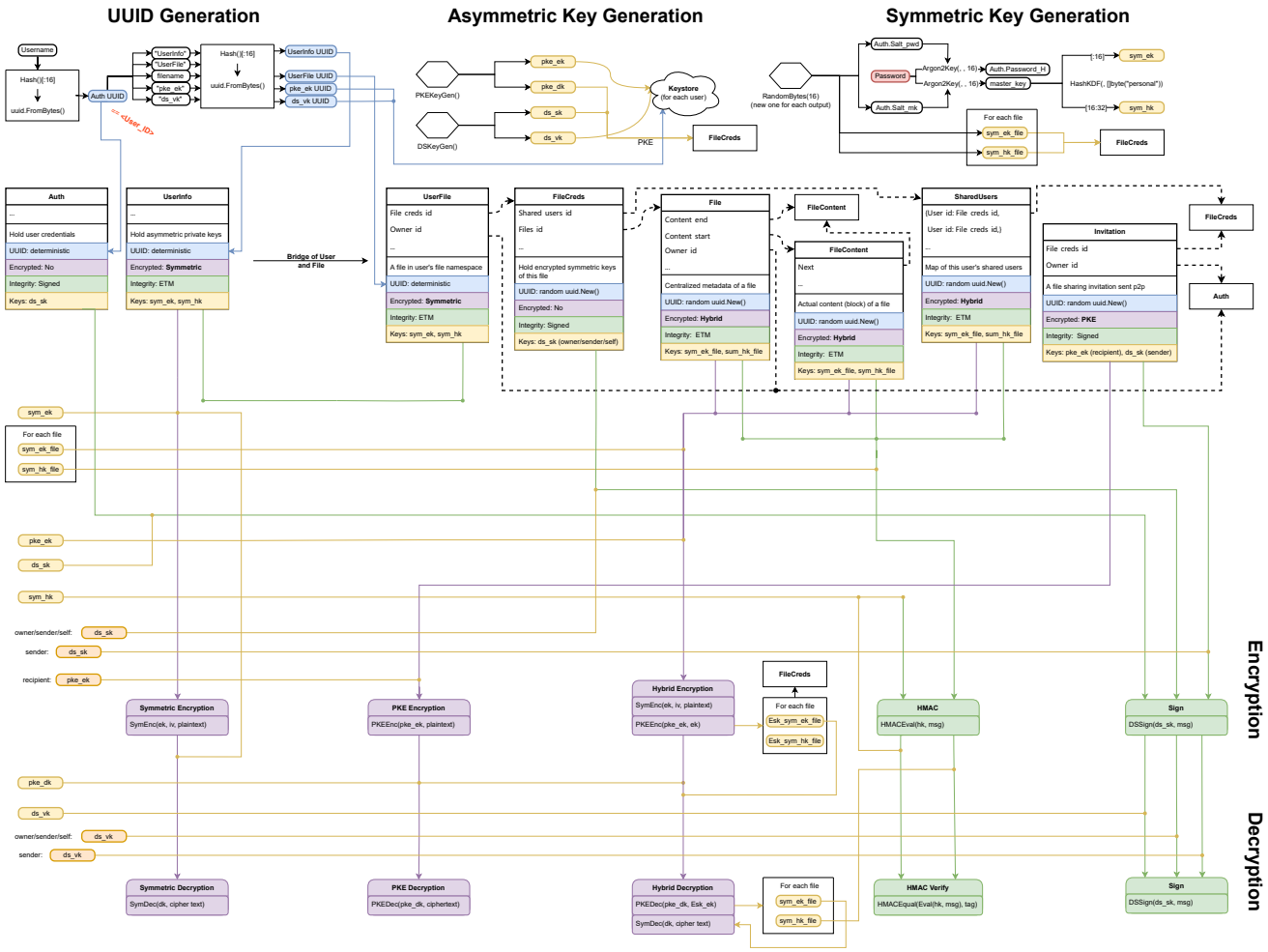
Security Abstraction

To ensure the security of user and file data, CryptoPP open source crypto library is used for crypto algorithms. Symmetric encryption and decryption are implemented with AES-CBC mode. Symmetric authentification is implemented with HMAC. Asymmetric encryption and decryption are implemented with RSA. Asymmetric authentification is implemented with RSA. Hashing is implemented with SHA3_256. Key derivation is implemented with argon2 in Argon2niche open source library with AVX SIMD CPU acceleration. UUID is used directly from uuid open source library.

The crypto algorithms are encapsulated into these functions to simplify the logic in client backend:

```
byteArray randomBytes(size_t len);
byteArray argon2Key(byteArray pwd, byteArray salt, size_t keyLen);
byteArray vectorHash(byteArray key);
void PKEKeyGen(PKEEncKey& publicKey, PKEDecKey& privateKey);
byteArray PKEEnc(PKEEncKey& publicKey, const byteArray& data);
byteArray PKEDec(PKEDecKey& privateKey, const byteArray& data);
void DSKeyGen(DSVerifyKey& publicKey, DSSignKey& privateKey);
byteArray DSSign(DSSignKey& privateKey, const byteArray& data);
bool DSVerify(DSVerifyKey& publicKey, const byteArray& data, const byteArray& signature);
byteArray HMACEval(const byteArray& key, const byteArray& data);
bool HMACEqual(const byteArray& key, const byteArray& data, const byteArray& mac);
byteArray HashKDF(const byteArray& key, const byteArray& data);
byteArray SymEnc(const byteArray& key, const byteArray& iv, const byteArray& plaintext);
byteArray SymDec(const byteArray& key, const byteArray& ciphertext_comb);
```

Here is the overview picture of our security web:



Database Server Communication

The client backend communicate with database server through TCP socket. Since both platforms are Windows, both programs call WINSOCK socket API to achieve socket interaction. The communication protocol frame is as follow:

```
#pragma pack(push, 1)   // packed attribute identifier for MSVC compiler
typedef struct {
    uint8_t table_name; // 0: Keystore, 1: Datastore; any for non-database operation
    uint8_t op;         // 0: Set, 1: Get, 2: Delete, 3: Quit, 4: AddFileToEngine, 5:
Search, 6: DeleteFileFromEngine
    uint8_t key[256];
    uint8_t value[65278];
}request_t;
#pragma pack(pop)
```

Multithreading is used to handle simultaneous client connections.

MySQL Database Server

To relieve the demand on memory, we select MySQL database system to store user and file data.

The database service runs on Windows 11 platform for its compatibility with MySQL 9.1.0. Xdevapi of MySQL connector C++ is used for calling database operations from C++ program. 2 tables (datastore and keystore) are used to store data and public keys, respectively. The data are stored as key-value pairs using "document", which is a new feature in MySQL 9.1.0, for automatic element size management.

Search Server Integration

The search engine is deployed at the database server side for better visibility and security to all public files. The server maintains the SearchEngine object. When a client posts certain requests, the search engine will be invoked and do corresponding operation.

The searching functionalities are encapsulated into 3 functions to simplify search engine interaction logic:

```
void addFileToEngine(string filename, vector<string> tags);
void deleteFileFromEngine(string filename);
vector<string> search(vector<string> tags);
```

Database Operation Abstraction

The purpose of database operation abstraction is to simplify the logic of interacting with database on client backend.

The database interact socket sendings are encapsulated into 6 functions:

```
void DatastoreSet(const uuid& id, byteArray data);
void KeystoreSet(const string& name, byteArray data);
byteArray DatastoreGet(const uuid& id);
byteArray KeystoreGet(const string& name);
void DatastoreDelete(const uuid& id);
void KeystoreDelete(const string& name);
```

This part of code is implemented as a library for client backend in userlib.cpp.

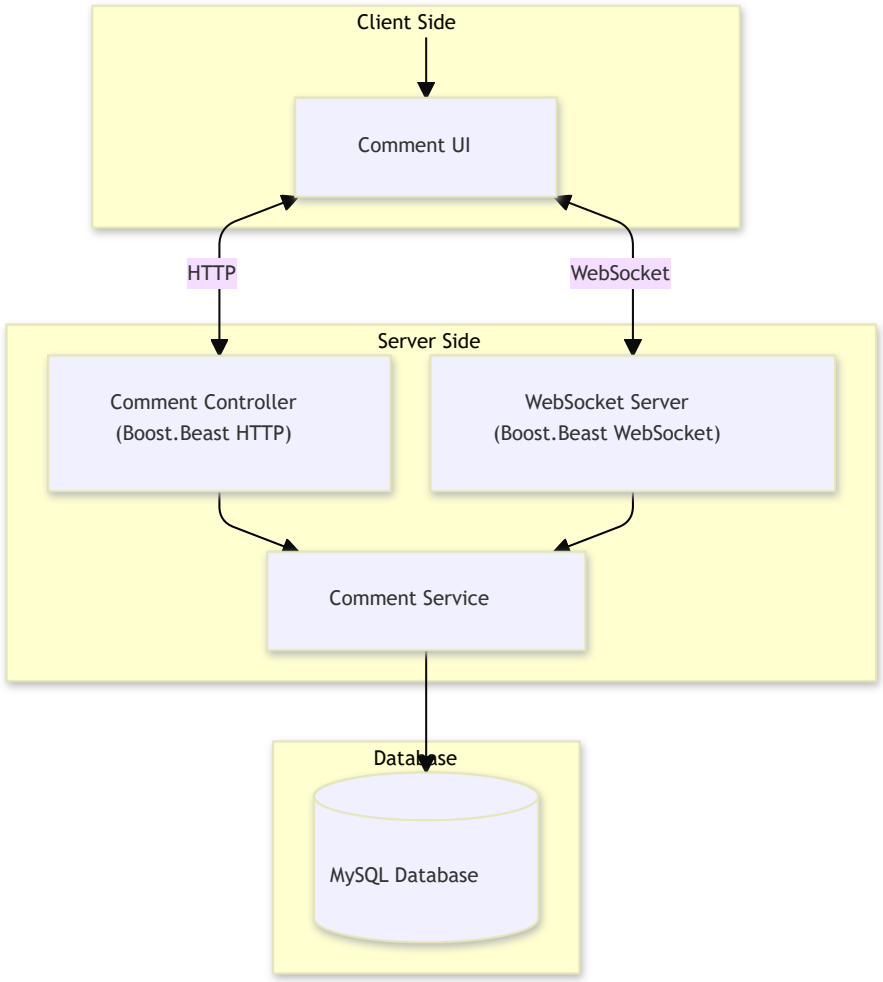
Client Frontend Communication

The client frontend communicate with client backend through TCP socket. Since both platforms are Windows, both programs call WINSOCK socket API to achieve socket interaction. The communication frame is of fix length 65536 Bytes, but the detailed slicing are different for each operation. In this communication, backend serves as server and frontend serves as client. The operations passing through this communication includes:

```
void initUser(const QString& username, const QString& password);
void getUser(const QString& username, const QString& password);
void storeFile(const QString& username, const QString& filename, const QByteArray&
content);
void storeFileUnit(const QString& username, const QString& filename, const QByteArray&
content);
string loadFile(const QString& username, const QString& filename);
void publicFile(const QString& username, const QString& filename);
void privateFile(const QString& username, const QString& filename);
void addFileSearch(string filename, vector<string> tags);
void deleteFileSearch(string filename);
vector<string> Search(vector<string> tags);
void appendToFile(const QString& username, const QString& filename, const QByteArray&
content);
void appendToFileUnit(const QString& username, const QString& filename, const QByteArray&
content);
string loadPublicFile(const QString& filename);
```

3. Comment Section (Chatting Room)

In this project, I successfully implemented the **chatroom functionality**, which allows users to interact through comments by posting, upvoting, downvoting, deleting, and retrieving them for specific courses. My contributions focused on the following key areas:



1. Comment Management:

- Designed and implemented the `CommentController` class to handle core comment functionalities, including posting, upvoting, canceling upvotes, deleting, and retrieving comments.
- Integrated the `CommentController` with the database and WebSocket server to ensure persistent storage and real-time broadcasting of updates.
- Utilized the **Crow framework** for defining RESTful APIs and routing HTTP requests.

2. Database Integration:

- Established a custom database communication protocol via raw sockets. The `KeystoreSet` and `KeystoreGet` methods were used for storing and retrieving comments as JSON arrays, leveraging **nlohmann::json** for efficient serialization and deserialization.
- Ensured that all comment data was persistently stored under a unique key for each course, simplifying retrieval and update processes.

3. Performance Optimization with Caching:

- Implemented a local in-memory cache (`comments` data structure) to temporarily store comment data during the session. This reduced frequent database interactions, significantly improving response time for common operations.

4. Real-Time Updates:

- Integrated WebSocket functionality using **Boost.Beast**, enabling real-time broadcasting of updates (e.g., new comments or upvotes) to all connected clients.
- Developed the `WebSocketSession::broadcast` method to deliver consistent, real-time updates across multiple user connections.

5. Concurrency and Thread Safety:

- Addressed potential race conditions by using `std::mutex` to lock access to shared resources, ensuring thread-safe operations and data integrity under high user concurrency.

Why My Work Stands Out:

I believe my work stands out because it delivers a well-rounded, user-friendly, and highly responsive chatroom system by combining multiple technical approaches:

- **Performance and Responsiveness**: The introduction of a local cache to reduce database access delays demonstrates a thoughtful approach to system optimization.
- **Real-Time Interactivity**: The seamless integration of WebSocket broadcasting enhances user engagement by providing immediate feedback for all actions in the chatroom.
- **System Reliability**: I ensured robust thread safety mechanisms, error handling, and synchronization between in-memory data and the database, creating a dependable system.
- **Scalability**: The modular design of the `CommentController` and its integration with the Crow framework ensures the chatroom functionality can be easily extended or scaled as needed.

4. **Search Ranking Algorithm**

Abstraction

This part mainly realizes the function of finding related files through the input information. This algorithm can sort the files in the database according to their similarity with the query file, and then output the sorted sequence.

Description

Document Information Processing and Tag Extraction

`string preprocessText(const string& text)`:
separate the document words according to the delimiter. Unify the word case, word segmentation form, and plural form.

`vector<string> extractTags(const string& text)`:
extract and store the document tags.

TF-IDF Calculation

`void addDocument(const string& filename, const vector<string>& terms)`:
For each document, the method takes its filename and a list of terms, calculates the term frequency, and stores it in an unordered_map associated with the document.

`void calculateIDF()`:
This method calculates the inverse document frequency for each term across all documents. IDF reflects how common or rare a term is in the entire corpus.

`unordered_map<string, double> calculateTFIDF(const string& filename)`:
Determining how often each tag appears relative to the total tag count in that document and how common or rare a tag is across the entire document collection. To calculate the contribution weight of a tag to the overall document similarity.

Cosine Similarity Calculation

`cosineSimilarity(vec1, vec2)`:
Calculates the cosine similarity between two TF-IDF vectors, representing the similarity between two documents.

Clustering

`vector<int> clusterDocuments(const vector<unordered_map<string, double>>& vectors)`:
Perform cluster analysis on the documents, classify the documents with high tag similarity into one cluster.

Search Engine

`void addFile(const string& filename, const vector<string>& tags)`:
Add files and their tags to the search engine, then calculate the TFIDF of each tags of files.

`void deleteFile(const string& filename):`
Delete files and their tags in the search engine.

Add_relationship_with_clusters

`void add_relationship_with_clusters()`:
Calculate the total similarity between each cluster and other clusters, regard each cluster as a point, the relationship between clusters as an edge, and the similarity as the edge weight, and establish an undirected graph to add relationships between clusters.

Build

`void build()`:
Calculate TFIDF values of different clusters and perform clustering operations.

SearchFiles

`vector<pair<string, double>> searchFiles(const vector<string>& queryTags)`:

This function searches and sorts the results related to the file based on the user's query tag. First, it accepts a query tag vector queryTags, converts these tags to lowercase and records their frequency of occurrence, and saves them in queryVector.

Then, the function traverses the file list fileTags and calculates the similarity between each file and the query tag. Specifically, it calculates the TF-IDF value of the file tag and compares it with the query tag vector through the cosine similarity algorithm to get a similarity value. All files and their similarity results are stored in results and sorted by similarity, with the most relevant files at the front.

Next, the function further processes the clustering information of the files and sorts them with the help of cluster ID. It uses the maximum spanning tree (MST) algorithm to rearrange the order of different clusters and ensure that the files are sorted by cluster and similarity. The function ensures that each file is added only once, and finally returns the files and their similarities sorted by relevance and cluster order.

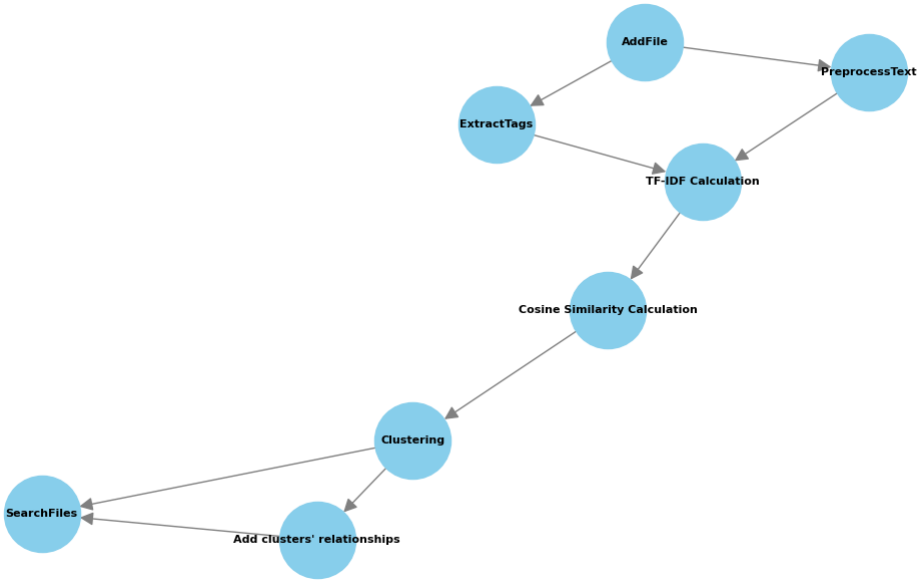
Maximum Spanning Tree Algorithm

First, an empty priority queue maxHeap is used to store pairs of edge weights and neighboring node indices, which helps identify the strongest connections between clusters.

Then, For each cluster that hasn't been visited yet, it is marked as visited, and its neighbors (clusters that are connected via an edge with a weight greater than 0) are added to the max heap.After that, continue to expand the tree by popping the top item from the max heap (which contains the available edge with the largest weights), visiting the corresponding cluster, and adding its unvisited neighbors to the heap.

By repeatedly processing the most connected nodes (clusters) , this algorithm gradually builds the tree by including the most strongly connected clusters , ensuring that the edges of largest edge weights are selected, similar to the approach used in a maximum spanning tree algorithm.

Process Diagram



Conclusion

This Algorithm first preprocesses and stores the **tags** of the documents, then uses the **TF-IDF algorithm** to extract and count the features of the tags of these files, and then cluster the files by their tags. After that, it uses search function to find the most related files and clusters, then uses **maximum spanning tree** algorithm to find the other related clusters. Finally it can output the sorted sequence according to their similarity with the query file.

4. Contributions

TEAM MEMBER	RESPONSIBILITIES	CONTRIBUTION
毛韦淇 Weiqi Mao (123030053)	Search Ranking Algorithm	20%
陈柏成 Baicheng Chen (123090015)	File Sharing Section	20%
陈丹阳 Danyang Chen (123090018)	Comment Section (Chatting Room)	20%
陈玥彤 Yuetong Chen (123090050)	UI (User Interface)	20%
张树涵 Shuhan Zhang (123090823)	Database System, Security abstraction and AI Assistant	20%
Note: There is overlap across different components, and the above responsibilities reflect the primary areas each member is focusing on.		Every member has an equal contribution.

5. Reflections

Team

The main issue we encountered was inconsistent environments. Since the programs in this project were distributed, everyone was using different setups, which made integrating the code challenging. Our solution was to use socket communication. Designing protocols and serialize data structures were difficult. But after the protocols were well defined, the communications can be well encapsulated. Through this process, we learned how to merge all the code in a multi-person project, a skill that was not taught in class.

Teamworking is necessary for this project. To acknowledge the demands of other team members and the project, we often held group meetings to communicate on progress, demands, and plans. It is convenient to use Github for version control, but not every team member are familiar with it. Many unnecessary works were done because of poor version control. This can be improved by integrating Git into the project tighter.

Unit testing are necessary in distributed works. Because many programs may have complicated dependencies, not every member can run the entire program on their machine. So it is necessary to design a unit testing mechanism to test the functionality of one's program before integrating with others' work. This is also done poorly in this project and caused unnecessary works.

Team member

Yuetong Chen

The difficulty I encountered is navigation between different pages. Every page is an object of a certain class, deleted when this page is closed. To insure information consistency when naviagting between different windows, I pass "username" and "coursesname" as arguments to each class, and create corresponding files to store information. In this way, every time a page object is created, it reads the files and initialize the setting according to the files. In this project, I wrote five classes and implemented interactions between them, gaining a deeper understanding of the concepts of classes and pointers taught in class. Additionally, I self-learned how to create a UI interface, implement API calls, and design file storage to maintain information consistency.

Baicheng Chen

In this project, the biggest issue I encountered was the conflict between different external libraries, such as the inability to use JSON for conversion and storage of RSA::PublicKey and RSA::PrivateKey from the cryptography library. My solution was to first convert the key type to a byteArray before storing it, and convert it back after retrieving it. Additionally, due to time constraints, I did not implement multi-file management for this project. Instead, I placed all the related logic into a single cpp file, which is not ideal for future maintenance. This issue will need to be given more attention in future coding projects. In my opinion, classroom learning provides the basic grammar and foundational knowledge, but for a computer science major, self-learning and programming are also essential. They are the best ways to apply the knowledge.

Shuhan Zhang

Working on programs unlike writing Leetcode questions or solving problems in exams, there were many issues to be noticed apart from programming knowledge. The global view of project and the managing codes are also vital. The obstacles I encountered were as follows.

This project relies on MySQL database system. To interact with the database program with C++, a official tool called "MySQL Connector C++" is necessary. However this connector library is maintained poorly. There were 2 completely different sets of exclusive APIs, and the documentation was very complicated. Fortunately, there were some tutorials on stackoverflow explaining the environment setup and API usage. By surfing through those tutorials, I managed to finish the program.

Some part of this program relies on complicated dependencies. My parts stick to Visual Studio IDE and vcpkg package manager for integrity, and smaller libraries are copied directly into the source code. This simplifies the process of project integration.

Normally, C++ offers multiple strategies using different features in solving similar tasks. But for 3rd party libraries, there might be strange requirement in calling methods that does not align with common habits. For example some encryption and decryption algorithms requires special BER encoders and decoders. This can be solved by actively reading documentations.

Weiqi Mao

In this project, The bigest difficulty I faced was how to handle the relationships between different clusters. To solve this, I treated each cluster as a node in a graph and used the similarity between clusters as the edges with weights. Essentially, I represented clusters as points, and the stronger their similarity, the stronger the connection (or edge weight) between them. I first calculated pairwise similarities between clusters by comparing documents within them. These similarity scores were stored and used to form the graph.

Then I applied the Maximum Spanning Tree (MST) algorithm to connect the clusters. Starting with any cluster, I used a max-heap (priority queue) to select the highest similarity edge. After visiting a cluster, I added edges to its unvisited neighbors with positive similarity scores. This allowed me to build a tree connecting the clusters in such a way that the strongest similarities were represented by the most important connections. The MST helped avoid redundant connections while ensuring the most meaningful relationships between clusters were highlighted. This approach provided an efficient way to represent and analyze the similarity between clusters in the dataset.

Through this project, I learned how to apply theoretical concepts such as graph algorithms and similarity metrics to practical problems, bridging the gap between abstract theoretical knowledge learned in class and efficient algorithm design.

Danyang Chen

During the project, I encountered several critical challenges and implemented effective solutions to overcome them. For example, direct interaction with the database caused significant delays during high user activity. To address this, I implemented a local cache to temporarily store comments during sessions, followed by asynchronous synchronization with the database, greatly improving system responsiveness. Additionally, concurrency issues arose when multiple users accessed shared resources simultaneously, which I resolved by using `std::mutex` to ensure thread safety and prevent data corruption. For real-time broadcasting, ensuring consistent delivery of WebSocket messages while addressing message loss was a challenge. I tackled this by delving into the asynchronous architecture of **Boost.Beast** and optimizing the broadcasting logic, which enhanced the system's stability and performance.

This project bridged the gap between theoretical knowledge and practical application, significantly enhancing my technical and problem-solving skills. Classroom learning provided a strong foundation in **object-oriented programming** and **data structures**, while self-directed exploration enabled me to master **Boost.Beast** for WebSocket communication, utilize **nlohmann::json** for efficient nested JSON handling, and experiment with caching strategies to balance performance and database synchronization. Integrating Crow, Boost.Beast, and the custom database module taught me how to debug and design workflows for distributed systems, underscoring the importance of combining structured classroom knowledge with self-driven learning to solve real-world challenges effectively.