SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

Project Report: Build Your Own Spotify

COMPUTER SYSTEMS AND PROGRAMMING

Professor:
Giorgio Richelli

Student:
Elizaveta Lapiga
2107930

Academic Year 2024/2025

1 Software Architecture

The BYOS (Build Your Own Spotify) project follows a client-server architecture, allowing multiple users to interact with a centralized music server over a network.

High-Level Design

- **Server:** Manages user authentication, song storage, metadata indexing, streaming, and ratings.
- **Client:** Interfaces for users and administrators to communicate with the server, perform playback, and manage content.
- **Communication:** Clients connect to the server over TCP using a known port defined in a configuration file.

Core Components

- **Authentication Module:** Verifies user credentials and role (user/admin) using a fixed-size buffer (64 bits).
- **Song Manager:** Stores songs locally on the server and handles addition, deletion, renaming, and metadata updates.
- **Metadata Index:** Maintains an in-memory index of song tags (title, artist, album, year, genre) for fast lookup.
- **Streaming Engine:** Sends audio files to clients upon request.
- **Rating System:** Allows users to rate songs and computes average ratings and download counts.
- **Concurrency Handling:** Uses multiple concurrent processes to support simultaneous clients.

Recovery and Fault Tolerance

- On startup, the server scans the music directory and rebuilds its in-memory metadata structures.
- Ensures IPC resources and temporary files are cleaned up on shutdown.
- Clients handle unexpected disconnections and clean up temporary data before exiting.

Data Structures

The server uses the following C data structures to represent song metadata and manage the music library in memory:

```
struct ID3v1Tag {
    char tag[3];
    char title[31];
    char artist[31];
    char album[31];
    char year[5];
    unsigned char genre;
};

struct SongMetadata {
    char filename[256];
    struct ID3v1Tag tag;
};

extern struct SongMetadata *song_index;
extern int song_count;
extern int song_capacity;

extern int semid;
```

Explanation:

- `ID3v1Tag` holds basic metadata for each song, following the ID3v1 format.
- `SongMetadata` stores the filename and its corresponding tag.
- `song_index` is a pointer to an array of songs currently stored in memory.
- `song_count` tracks how many songs are currently loaded.
- `song_capacity` indicates the maximum capacity of the song index array.
- `semid` indicates the semaphore ID to control the access to the dynamically allocated array of songs indexes.

2 Functional Design Choices

Commands for All Users and Admins

- `list` – List all songs available on the server

- `play <song_name>` – Download and play a specific song
- `logout` – Log out from the current user session
- `login <user> <pass>` – Log into the account
- `info <song_name>` – Display metadata of the song (tags)
- `search <album/artist/year/genre> <value>` – Search for songs using metadata
- `rate <song_name> <1-5>` – Rate the song (e.g., 1 to 5)
- `avg <song_name>` – View average rating of a song
- `dlcount <song_name>` – View how many times the song has been downloaded
- `exit` - exit program

Additional Commands for Admins

- `add <song_name>` – Upload a new song to the server
- `delete <song_name>` – Delete a song from the server
- `rename <song_name> <new_name>` – Rename an existing song
- `createuser <name> <password> <role>` – Create a new user account (with role)
- `changetag <song_name> <album/artist/year/genre> <value>` – Modify a song's metadata

3 Non-Functional Design Choices

- Concurrent processing using `fork()` for each client
- Efficient memory via dynamic allocation for song indexing
- Every restart of the server songs directory rescan and rebuilding indexes array
- The index array of songs is updated during rename, change tag, or delete operations performed by the admin. A semaphore ensures exclusive access to the array during these operations
- Text-based protocol for commands
- Role-based access restriction for admin-only commands

- Graceful shutdown and resource cleanup (exit, logout)
- Configurable server port via config file
- Structured error handling with numeric codes
- Use of ID3v1 format with fixed-size tag fields
- Client has isolated DB session: Each child process independently calls `init_database("music.db")` after fork().
- SQLite access protected with `sqlite3_busy_timeout()`, it helps avoid immediate failure when a DB is locked by another process, giving SQLite time to retry.
- Proper memory and file cleanup
- Validation of file transfer using size

4 Main Functions and Their Parameters

Network Setup (`network_utils.c`)

```
int create_socket()
-- Creates a socket and returns its file descriptor.

void handle_error(const char *message)
-- Handles errors by printing a message and exiting the program.

void configure_server(
    struct sockaddr_in *server_addr,
    int port,
    const char *ip_address
)
-- Sets Up Server Address.

void configure_client(
    struct sockaddr_in *server_addr,
    int port,
    const char *server_ip
)
-- Sets Up client Address.

int load_config(
    char *ip_buffer,
```

```
    size_t ip_buf_size,
    int *port_out
)
-- Loads the IP and port for configuration.
```

Tag and Song Management (tag_handler.h)

```
void init_song_index()
-- Initializes the dynamic song index.

void free_song_index()
-- Frees the memory allocated for song index.

void add_song_to_index(struct SongMetadata *new_song)
-- Adds a song entry to the in-memory index.

int read_id3v1_tag(const char *filepath, struct ID3v1Tag *tag)
-- Reads tag info from an MP3 file.

void search_tag(int client_fd, const char *command)
-- Searches tag fields based on client command.

const char* get_genre_name(unsigned char genre)
-- Converts genre byte to a human-readable name.

void handle_changetag(int client_fd, const char *command, const char *role)
-- Updates tag field for a song.

void index_songs(const char *music_dir)
-- Rebuilds song index from the music directory.
```

Database Operations (db_handler.h)

```
int init_database(const char *filename)
-- Opens SQLite DB and initializes tables.

int delete_song_db_entries(const char *song)
-- Deletes all rating/download data for a song.

void close_database()
-- Closes the SQLite DB connection.
```

```
int rate_song(const char *song, const char *user, int rating)
-- Inserts/updates user rating.

float get_average_rating(const char *song)
-- Computes average rating for a song.

int increment_download(const char *song)
-- Increments download counter.

int get_download_count(const char *song)
-- Returns total download count.
```

Disk Space Checking (disk_space.h)

```
int check_disk_space(const char *path, long required_bytes, long *disk_space)
-- Verifies available disk space.
```

Login/Auth (login.h and login_client.h)

```
int check_credentials(const char *username, const char *password, char *role_out)
-- Validates login info.

int client_login(int sock_fd)
-- Handles login prompt and logic on client side.
```

Request Handling – Server (request_handler.h)

```
void handle_cmd(
    int client_fd,
    const char *command,
    int *logged_in,
    char *role,
    char *username
)
-- Parses and routes client commands.

void handle_list(int client_fd)
-- Sends song list to client.

void handle_get(int client_fd, const char *filename)
-- Sends requested song file.
```

```
void handle_add(int client_fd, const char *command, const char *role)
-- Receives and stores a new song.

void handle_delete(int client_fd, const char *command, const char *role)
-- Deletes a song from disk, DB, and index.

void handle_rename(int client_fd, const char *command, const char *role)
-- Renames an existing song.

void handle_newuser(int client_fd, const char *command, const char *role)
-- Adds a new user to credentials file.

int handle_login(
    int client_fd,
    const char *command,
    char *role_out,
    char *username_out
)
-- Server-side login logic.

void handle_info(int client_fd, const char *filename)
-- Sends song metadata.

void handle_rate(int client_fd, const char *args, const char *user)
-- Records user rating.

void handle_avg(int client_fd, const char *args)
-- Sends average rating.

void handle_dlcount(int client_fd, const char *args)
-- Sends download count.

int song_exists(const char *songname)
-- Checks if a song is in the index and not deleted.

int remove_song_from_index(const char *filename)
-- Removes a song from the in-memory index.
```

Request Handling – Client (`recieve_handler.h`)

```
void handle_rcv(int sock_fd, const char *command)
```

```
-- Processes a generic response.

void handle_rcv_list(int sock_fd)
-- Receives and displays song list.

void handle_rcv_get(int sock_fd, const char *filename)
-- Receives and saves a song file.

void handle_snd_add(int sock_fd, const char *filename)
-- Sends a new song file to server.

void handle_rcv_delete(int sock_fd, const char *filename)
-- Handles server-side deletion response.

void handle_rcv_newuser(int sock_fd)
-- Handles account creation.

void handle_rcv_rename(int sock_fd)
-- Handles rename result.

void handle_rcv_tag(int sock_fd)
-- Processes search tag result.

void handle_search_response(int sock_fd)
-- Receives and prints search result.

void handle_response(int response)
-- Parses numeric response codes.

void handle_rcv_changetag(int sock_fd)
-- Handles tag update.

void recv_and_print(int sock_fd)
-- Reads and prints a single-line message.

void handle_rcv_rate(int sock_fd)
-- Handles rating request.

void handle_rcv_avg(int sock_fd)
-- Receives and displays song rating.

void handle_rcv_dlcount(int sock_fd)
-- Receives and displays download count.
```

```
void handle_play(const char *filename)
-- Plays a song using system player.

void cleanup_cache()
-- Removes temporary downloaded files.
```

Semaphore (semaphore.h)

```
void init_semaphore()
-- Initiate semaphore.
```

Cache Handler (cache_handler.h)

```
int check_cache(const char *filename)
-- Checks if a file exists in local cache.

void cleanup_cache()
-- Removes cached files from temp dir.
```

5 Response Codes Header File: response_codes.h

```
OK 0
ERR_GENERIC 1
ERR_PERMISSION 2
ERR_FILE_NOT_FOUND 3
ERR_INVALID_ROLE 4
ERR_PARSE 5
ERR_USER_EXSISTS 6
ERR_FILE_OPEN_FAIL 7
ERR_INCOMPLETE_TRANSFER 8
ERR_TAG_NOT_FOUND 9
ERR_TAG_PARSE_FAIL 10
ERR_RESPONSE_RECV_FAIL 11
ERR_UNKNOWN_COMMAND 12
ERR_FILE_EXISTS 13
ERR_DISK_IS_FULL 14
ERR_LOCK_FAILED 15
ERR_INVALID_TAG_TYPE 16
```

Notes

- The constants define error codes used for client-server communication.
- `OK` (0) indicates success.
- Other codes represent specific failure conditions (e.g., `ERR_FILE_NOT_FOUND`, `ERR_PERMISSION`).

6 Known Problems

- No password encryption; plain text credentials
- Metadata field validation not enforced
- Commands and data sent unencrypted over TCP
- SQLite used without mutex; single-process safety only
- No full retry mechanism for failed transfers; However, incomplete uploads are retried once.
- Manual deletion of files can cause inconsistency if not re-indexed
- Username and password are stored in fixed-size buffers rather than dynamically allocated arrays. However, the input is explicitly truncated to 63 characters.

7 Test Cases

Login

Test Case	**Description**	**Expected Result**
Valid login	Correct username/password	Login successful
Invalid login	Wrong password	Login failed
Nonexistent user	Username not in file	Login failed
Admin login	Admin credentials	Login as admin

Song Listing

Test Case	Description	Expected Result
List songs	Songs in directory	Full list shown
No songs	Empty music directory	No songs shown

Upload / Add Song

Test Case	Description	Expected Result
Upload as admin	Admin uploads song	File added, index updated
Upload as user	Normal user attempts upload	Permission denied
Duplicate upload	File already exists	ERR_FILE_EXISTS

Download / Play

Test Case	Description	Expected Result
Play song	Use system audio player	Download and plays song
Play song again	Use system audio player	Song plays without downloading
Play unknown song	File not in index	ERR_FILE_NOT_FOUND

Metadata

Test Case	Description	Expected Result
Info command	View song metadata	Title/artist/year shown
Search by tag	Search by genre/artist	Matching songs returned
Invalid field	Search by wrong tag type	ERR_INVALID_TAG_TYPE
No match song	Search by no existent tag value	No songs found

Ratings and Stats

Test Case	Description	Expected Result
Rate song	User gives rating (1–5)	Rating saved in DB
Re-rate song	User changes rating	Previous rating updated
Invalid rating	Out of bounds rating	ERR Invalid usage
Get average	Fetch average rating	Float value shown
No ratings yet	No ratings recorded	0.00 value is shown
Download count	Song downloaded many times	Count matches total

Admin Functions

Test Case	Description	Expected Result
Create user	Admin creates new user	User added to credentials.txt
Rename song	Change song filename	File and index updated
Delete song	Remove song from server	File + DB + index cleaned
Delete missing song	Try to delete nonexistent song	ERR_FILE_NOT_FOUND
Change metadata	Modify album/genre/etc.	Metadata updated and index updated