Project 2: UFmyMusic

CNT4007

Ana Garcia

Lysandra Belnavis-Walters

Table of Contents

Overview

UFmyMusic combines music across machines by syncing files from the server to the client with the use of sockets. The server acts as a catch-all storage unit, holding all songs saved across multiple machines for a given user. These songs may not be accessible from your local machine but are quickly downloadable with the PULL function. Furthermore, UFmyMusic supports concurrent client requests due to its multithreaded server.

Key Features

1. **LIST Command**
   The LIST command lists all the songs that a server currently has for the client.
2. **DIFF Command**
   The DIFF command differentiates songs that are exclusive to the server from those that are on both the client and server.
3. **PULL Command**
   The PULL command downloads songs from the server to the client's machines. To maintain the organization of the files, these songs are prefixed with "Client" instead of "Server".
4. **LEAVE Command**
   The LEAVE command allows (1) the client to leave; and (2) the server to free any memory it was using for the client.
5. **Handling Multiple Clients Concurrently**
   The server can handle multiple clients concurrently with the help of pthreads. We chose to use pthreads as they were easy to implement and easy to integrate with our existing code base. To handle multiple clients, we must also be able to differentiate those clients which is why we prompt the client for a username (something that they can identify themselves and their files with).
6. **Using Hashing to Differentiate Songs**
   Hashing is used to differentiate songs on the basis of an ISWC code. An ISWC code is "a unique code assigned to a specific musical work or composition written by the songwriter(s)" [1]. Thus, we'll use this to uniquely identify the songs with the acute understanding that two files with the same ISWC code are the same song. Alas, we also use hashing to merge songs with different (possibly faulty) filenames but identical file contents.
7. **Requires a Single Directory**
   Instead of creating directories for each and every client, the filename is used to indicate ownership of a song file. For example, files prefixed with "Server" are server files, and files prefixed with "Client" are client files. The username in the filename indicates which particular entity the file belongs to. Thus, a server file with a username of "A" (i.e. "Server_A_T-XXX.XXX.XXX.txt") indicates that the server is holding that file for client A. Likewise, a client file with a username of "A" (i.e. "Client _A_T-XXX.XXX.XXX.txt") indicates that the song file belongs to client A. Finally, a filename will also include the ISWC code of the song it represents so that we don't have the issue of duplicate filenames (as each client can only have at most 1 instance of each unique song).

Key Data Structures

To flesh out this project, we required the help of encapsulation, all of which we describe below.

1. **Song:** A struct that represents a song (fundamentally a text file), in which each holds its own unique ISWC code.
    a. Song files are titled in the format "Server_[Username]_T-XXX.XXX.XXX.txt" or "Client_[Username]_T-XXX.XXX.XXX.txt".
    b. The content inside a song file must have the form "T-XXX.XXX.XXX\n[Song Title] [Song Artist]", where the first line ideally matches the ISWC used in the filename. There is no strict rule on the format of the proceeding lines after the first two lines.
    c. Although our project is built off the understanding that the ISWC code in the filename will match the ISWC code in the file contents, incorrect filenames (a typo in the ISWC code) are still accounted for as long as the ISWC code in the file is accurate. To be clear, faulty filenames would have to be manually created; we do not create faulty filenames as their mere existence would compromise our organization system. In short, due to the fact that we name our files very specifically, the duplicate-content-different-filename situation could not actually happen. However, if it were to happen for whatever reason, our program would still be able to function due to the hashing technique it employs.
    d. When creating a text file for a song, the song is always born on the server first—that is, we only (manually) create the text files starting with the "Server_" prefix. Song files with a "Client_" prefix are created in the PULL operation.
    e. Songs are hashable objects, allowing for the identification and management of duplicates (see **Libraries** below).
    f. The rest of the song file (after the first two lines for the ISWC code and the song title, respectively) can be whatever you'd like. These extra lines are not read or stored in a Song struct. These extra lines will still be pulled to the client. For the purposes and simplicity of this project, we only fill out the first two lines (ISWC code and song title).
    g. As a further side note, make sure to end the lines of a song file with a newline. Therefore, if there are n lines in the song file, make sure that each of those n lines are terminated with a newline.
2. **Libraries:** Array of Song objects.
    a. This struct provides a neat way to encapsulate an array of Song structs.
    b. A hashtable is used to check when two songs contain the same content but have different file names (i.e. The ISWCs in the file name are different). In this case, one of the duplicate files will be stored. We don't concern ourselves with the particular

file we should push to the client because of our core understanding that files with the same ISWC code will have the same content.

c. If our current song is in the hashtable, we know it is a duplicate. Else, we add it to the library as a unique song. This mechanism prevents any confusion that could be caused with duplicate song files.

3. **Headers**: A struct used to prepare the receiving entity for a variable length of bytes.
    a. As a client or server may need to send a variable length of bytes, the receiving entity must be aware of the number of bytes that they're planning to send.
    b. Header structs are used to notify the receiving entity of (1) the action it's requesting (`action`); and (2) the number of bytes it's going to send next (`totalBytesToRecv`).

Structs

To more easily handle song files in the program, we use structs to represent file(s). A song file is represented with a **Song** struct, and a list of song files is represented with a **Library** struct. We also have other structs such as the Array and Map structs to help us store valuable information.

1. **Song**: This struct contains three fields: a string (char*) for the song's ISWC code, a string (char*) for the song's title, and a UT_hash_handle hh.
2. **Library**: This contains an Array struct – an abstraction that imitates some functionality of C++'s vector – which holds Song structs.
    - Library structs are used to contain the songs that a particular entity (client or server) owns. When a library is used for a server, we're loading the songs a server owns for a particular client.
3. **Array**: Imitates some functionality of the std::vector in C++. Used to easily store related pieces of information.
4. **Map**: Imitates some functionality of the std::map in C++. Used to help with comparing libraries.
    - We use a Map struct to store the results of the DIFF command – where the key is the client's identifier (i.e. their username) and the value is an Array of ISWC codes that the client did not have – so that we can reference them in the PULL command.
5. **Header**: Used to initialize and prepare for further communication between the client and server. Contains two fields: a 32-bit number to indicate an action and a 32-bit number to indicate the total of bytes for the receiving host to receive (totalBytesToRecv).
    - Numbers of 1, 2, 3, and 4 indicate the LIST, PULL, DIFF, and LEAVE actions, respectively.
    - If the receiving entity sees that the total of bytes to receive is not 0, it knows that the sender is sending more information of a length totalBytesToRecv.
6. **ActionData:** This struct contains several fields: (1) a pointer-to-Map struct which stores the differences (particularly the ISWC codes) between a server and a particular client; (2) a pointer-to-Header struct which just stores the Header struct that a client will send; (3) a string (char*) for the client's ID (the username); (4) an integer representing the client's socket; and (5) an integer representing the server's socket.
    - As shown in the book, this struct is used so that we can pass information to threads.

Main Files

1. **Client.c and Client.h**
    a. Handles user interactions.
    b. Connects to the server via client socket.
    c. Performs actions such as LIST, DIFF, PULL, and LEAVE (refer to Main Operations).
2. **Server.c and Server.h**
    a. Listens for client connections.
    b. Handles multiple clients using pthreads.
    c. Processes client requests.
3. **ClientHandler.c and ClientHandler.h**
    a. Manages client-side operations (implements client's LIST, DIFF, PULL, and LEAVE functionalities).
4. **ServerHandler.c and ServerHandler.h**
    a. Manages server-side operations (implements server's LIST, DIFF, PULL, and LEAVE functionalities).
    b. Contains a function that returns the server's address – used so that the client can connect to the server (which will be on the returned address).
5. **Song.c and Song.h**
    a. Defines structures for songs and libraries.
    b. Handles loading, saving, and comparing libraries.
    c. Encodes and decodes Song structs and Library structs.
    d. Manages duplicates via hashtables.
6. **File.c and File.h**
    a. Handles I/O operations.
    b. Encodes and decodes files.
    c. Contains helper functions such as clientFileList which returns an Array of a server's client's filenames.
7. **Host.c and Host.h**
    a. Handles sending and receiving bytes.
    b. Handles byte transmissions (sending and receiving) and conversions.
    c. Encodes and decodes Header structs.
8. **Array.c and Array.h**
    a. Holds functions to create and delete an Array struct, and to add or get elements from an Array struct.
9. **Map.c and Map.h**
    a. Holds functions to create or delete a Map struct, retrieve its entries by key, and put (which could be adding or updating) entries.

10. create_songs.c

a. Program used to aid user in creating server song files. Prompts user to enter a username, ISWC, song title, and artist.
b. If used, make sure that the song title you enter is less than 255 characters to prevent any memory errors!

Encoding and Decoding

As the encoding and decoding of objects is an important part of this project, it's probably important to discuss.

1. **Song:** We only encode the ISWC code and title of a song. As these fields are of variable length, we need to write the length of those fields to the byte stream before writing the actual characters. Thus, an encoded Song struct follows this format: [Length of ISWC Code], [Length of Song Title], [Song's ISWC Code], [Song's Title Bytes]. When decoding a Song struct, we read the first two 4-byte integers from the stream (the length of the ISWC code and the length of the song title) and use those numbers to extract the ISWC code and the song title from the stream.
2. **Library:** As a Library is a list of Song objects, all we need to do to encode a Library struct is to (1) write the number of encoded Song structs in the byte stream; and (2) write all the Song structs to that byte stream – and that's what we do. To decode a Library struct we work backwards: we read the first 4-byte integer and use that to extract all the Song objects via the bytesToSong function in Song.c.
3. **File:** Encoding and decoding a file follows the same format as a Song struct. We write the number of bytes in the file (4-byte integer), and then we write the actual bytes of that file (the content) to the byte stream. This functionality is stored in the fileToBytes function. On the flip side, when decoding a file, we read the number of bytes in that file (it's the first four bytes), and then we extract the bytes of said length into a string. This functionality is used in the downloadFile function where we download the file that a server sends.
4. **Files:** The process of encoding and decoding multiple files is similar to encoding and decoding multiple Song structs. We write the number of files that are going to be encoded in the byte stream (again with a 4-byte integer), and then we write the representation of each of those files to that byte stream by repeatedly calling fileToBytes. Decoding a byte stream representation of a multiple files (referred to as a file list in the program) is done in the downloadFileList function. Similar to the decoding of a Library struct, we first read in the number of files (first four bytes), and use that to repeatedly find (1) the number of bytes in a file; and (2) the bytes for that file.

Determining if Two Files are Equivalent

With the help of hashing, we can determine whether two files contain the same content or not by determining if they have the same ISWC code. This only requires that the first line of a file be read – not the entire file. This is because we built our project with the understanding that files with the same ISWC code will ultimately be the same (or at least, treated the same). Thus, we only need to inspect the ISWC code to determine whether two files are equivalent or not.

Now, let's say that a horrible error takes place and files are incorrectly created. As a result, the server has two files with the same file content (at minimum, the same ISWC code), but different filenames (only differing in the ISWC portion of the filename). These two files are obviously the same as they have the same contents, but would our program treat them differently because of the differing filenames? No! This is because our project (1) inspects the ISWC code of the file itself; and (2) uses hashing so that duplicate files with the same ISWC code are treated the same. Also, only one of these files will be sent, and that will be the file that has the correct ISWC code. However, any one of these duplicate files could be used for outputting information in the LIST and DIFF commands. We provide an example of this in our shell script.

However, to be absolutely clear, the instances of duplicate-content-different-filenames do not happen during the execution of the program as we've structured our project so that a host will only have one file for each unique song due to the file naming scheme. Therefore, the ISWC code of the filename should always match the ISWC code in the file content. This means that if you see the files "Server_A_T-111.111.111.txt" and "Sever_A_T-111.111.112.txt", you can safely infer from the different ISWC codes in the filename that these files would be storing information about two different songs. If these song files are somehow storing information about the same song (i.e. the duplicate-content-different-filename situation we discussed in the prior paragraph) this is only because someone went in and deliberately messed with the filenames. We are just using duplicate-content-different-filename situation to (1) better explain that our project determines whether two files are equivalent or not by using the ISWC code (i.e. if the song files have the same ISWC code, they are treated the same); and (2) show you that our project can handle this situation, if needed. To reiterate once more, files that have the same ISWC code in their contents and different ISWC codes in their filenames (i.e. duplicate-content-different-filename) **are not a standard part of our architecture and will not naturally exist or be created within the constraints of our project unless you've intentionally created those files to test the duplicate-content-different-filename scenario**.

Client-Server Communication

Upon execution, the server program will initialize its server socket and wait to receive a username. The server will be listening on port 9000. When the client program has started, the client socket will initialize and connect to the server socket. The user will then be prompted to enter in a username. Once a valid username has been inputted (must be less than 200 characters and only consist of alphanumeric characters), it will be sent to the server. When the server receives this username, it will initialize a packet of data that will be used in a newly created thread; this thread will handle all the actions that a client could request. As we've used a thread, the server can continue to accept connections from other clients.

Similar to a UDP connection, we use 8-byte (4 bytes for the action and 4 bytes for totalBytesToRecv) headers to initialize and prepare for communication. Data sent over the connection (after the header has been sent) will contain encoded data (i.e. songs, libraries, files, or strings) in byte streams. When the data is received at the other end, it is decoded back into the correct object.

After a thread has been created for a client, the server-thread will wait for the client to send the header which will store the action the client requested (and the number of bytes it's going to send in the case of a DIFF command). Upon receiving the header, the server-thread will carry out whatever functions it may need and send back information. Before sending this information, however, the server-thread will first send a header that indicates the number of bytes it plans to send. This back-and-forth behavior will continue between the two parties until the client leaves. The server-thread will also terminate itself once its designated client leaves and release all the resources it had held for that client.

Main Operations

UFmyMusic provides four different actions for users: (1) LIST, (2) DIFF, (3) PULL, and (4) LEAVE. The user will be prompted to choose from any of the four options at the start of the program and after actions (1)-(3).

1. **LIST** lists all files present in the client and/or the server. All files will be present in the server and may also exist in the client, but cannot exist solely in the client.
    a. The LIST action starts with the client (who is identified by Bob) sending the server a header with an action of 1 and a totalBytesToRecv of 0. Upon receiving this header, the server will load all the files it has for Bob by crawling the directory and looking for files that start with "Server_Bob" (i.e. loading Bob's library). After it has gathered all of Bob's files, the server will load them all into Song structs so that it can more easily create the output. After creating the output, the server sends back a header with an action of 1 and a totalBytesToRecv set to the length of that string output. It will also send the bytes of the string output after sending the header. The client will thus receive the header and then receive and display the string output.
2. **DIFF** displays which files are present in the client (represented by a plus symbol (+)) and which are only present in the server (represented by a minus symbol (-)). Useful to refer back to after (3) PULL.
    a. The DIFF action starts with the client (Bob) loading its library by crawling the directory for files that are prefixed with "Client_Bob". After finding these files and storing their information in Song structs (and ultimately a Library struct), the client will encode its Library in a byte stream via the libraryToBytes function. Soon after, the client sends the server a header with an action of 2 and a totalBytesToRecv set to the length of said byte stream. The server will receive all this data and load the client's library on its end via the bytesToLibrary function. Then, the server will load the library it has stored for the client. With the server and client library, the server compares the two using ISWC codes. During its comparison, (1) a string output will be created and maintained for the client; and (2) the ISWC codes of the songs the client does not have will be stored in an Array struct and attached to a client with a Map struct. When this is all said and done, the server will send a header wherein the totalBytesToRecv is the length of the output it's going to send. The client which will ultimately receive this header and the output will finally display the output to the screen.

3. **PULL** sends all files missing from the client to the client. This function duplicates and downloads files, changing their name prefix from "Server" to "Client". If there are no files to pull, nothing happens.
    a. The PULL action starts with the client (Bob) sending a header of action 3. After being notified of this request, the server will load all the song files that the client doesn't have using the stored ISWC codes from the DIFF command. As you may remember, a song's filename has three parts: the "Server" or "Client" prefix, the client's username, and the ISWC code. Since the server has the client's username (stored in the ActionData struct) and the ISWC codes of the songs the client doesn't have, it can effectively create a list of the filenames of the songs that it must send to the client. Alas, after encoding these files into a byte stream (via fileListToBytes), the server will send the client a header with an action of 3 and a totalBytesToRecv set to the length of that byte stream. The client will ultimately receive the bytes of all the files and download them to their local machine using the downloadFileList function.
    b. You must call DIFF before calling PULL to see updated actions. For example, if you pull files using PULL, call DIFF to update the server with the knowledge of the new files you have, else you'll keep downloading the same file(s) repeatedly.
4. **LEAVE** closes the client connection.
    a. The client will send the server a header of action 4 and terminate its process. The server, upon receiving a header of 4, will terminate itself (that is, the thread), not before releasing all the memory it had allocated for the client (i.e. the ActionData).

What to Expect

Before trying out the program yourself, it might be helpful to see what output you should do and expect.

1. Start the server, leaving it running while we initiate the client connection(s). The following steps may be duplicated in the case of multiple clients.

```
rain:~/CNT4007/cnt4007_p2> ./server
```

2. Once the client connection has been initiated, the user will be prompted for their username which corresponds to the files created, if any.

```
rain:~/cnt4007_p2/cnt4007_p2> ./client
Welcome (Back) to UFMyMusic!
Enter Username: Alice
```

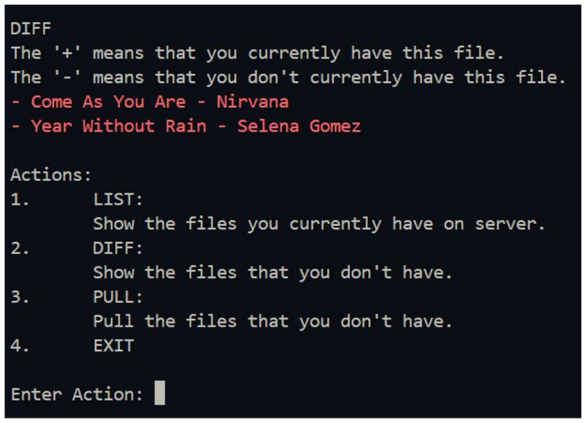
3. Next, the user will be asked to select from a list of actions: (1) LIST, (2) DIFF, (3) PULL, and (4) EXIT. Enter any digit 1-4 to run the respective command. This action list will be output after each command, except for (4) EXIT.

```
Actions:
1.      LIST:
        Show the files you currently have on server.
2.      DIFF:
        Show the files that you don't have.
3.      PULL:
        Pull the files that you don't have.
4.      EXIT

Enter Action: 1
```

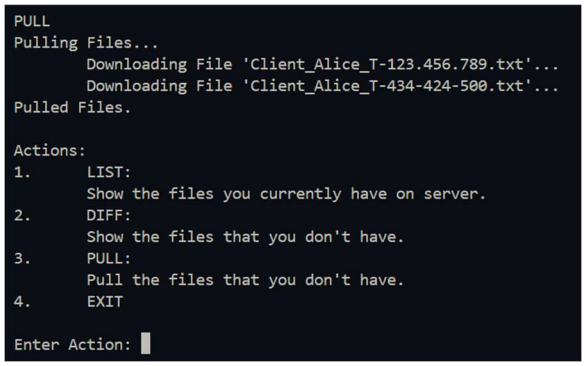
4. LIST brings up all files belonging to the server, which may or may not be in the client. The song's title and ISWC code (a.k.a. the first two lines of the song file) will be listed.

```
LIST
Your Files:
Come As You Are - Nirvana [T-123.456.789]
Year Without Rain - Selena Gomez [T-434-424-500]

Actions:
1.      LIST:
        Show the files you currently have on server.
2.      DIFF:
        Show the files that you don't have.
3.      PULL:
        Pull the files that you don't have.
4.      EXIT

Enter Action:
```

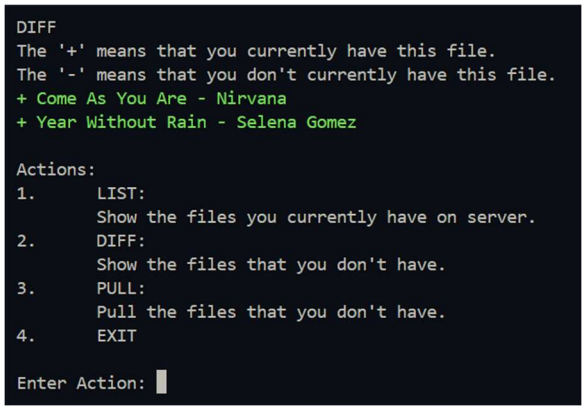
5. DIFF displays the files currently on the client (or, in other words, the songs belonging to both the current user and server) with a plus symbol (+) and the files that are not present in the client but exist on the server with a minus symbol (-).



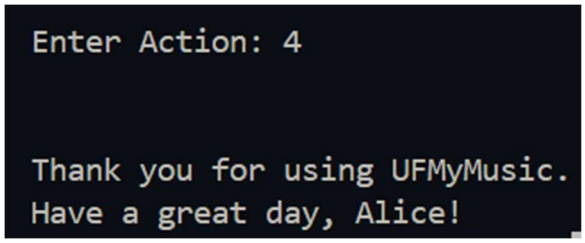
6. PULL downloads all server-exclusive files to the client by essentially duplicating the file and replacing the prefix with “Client_”.



7. Running DIFF again will show that the files have been added to the client with a plus symbol (+) and green text.



8. EXIT closes the client connection.



If you "ls" into the current directory, you'll be able to see that two new files were indeed created when the PULL command was executed. These client files will have the same exact content that the corresponding server file had.

Sources

1. ISWC Codes
2. Hash Tables for C
3. Documentation for C Hash Tables