CS 3251 Project 4
Architecture Overview
Jerry Yee, Ryan

μ -zic: A convenient music App

Folders

The server stores files in **./repo** and the client stores its files in the music directory within its external storage public directory. **These folders must exist before running the program**. The client sets the IP to connect to server at **130.207.114.23 (GT Shuttle Server)**.

Server

The bulk of the work on the server side is done between the files server.c and fileUtil.c. The server is assumed to be always on and listening for clients to connect to it and make requests. When they want to start a session, the client will send a connect() request to the server. When it receives the connection request, the server will accept() and begin a session with the client on a new socket. On the newly-implemented *cap* command, the server calls the doCapServer() function in fileUtil.c in order to parse through the iTunes Music Library.xml file which is in the server's local directory. Parsing is done by seeking to each tag using simple string operations using three predefined strings, TAG_NAME, TAG_PLAYCOUNT, and TAG_SIZE. A data structure is populated called CapInfo with these three members and a next pointer to the next element.

```
typedef struct capInfo
{
   char name[FILENAME_MAX];
   int size;
   int playCount;
   struct capInfo *next;
} capInfo;
```

After getting a sorted linked list from the iTunes XML file, the server traverses the linked list, checks the size of each file, and adds the filename to the output buffer until the cap size is reached. It then sends this list to the client, which responds with a pull request containing the files to be sent and the server pushes these files to the client.

The server is multithreaded to handle multiple clients, so each new connection with a client creates a new **pthread** on a different socket. The decision to use pthread was based on the workflow process (and carried over from Project 2). The workflow was to first get single user

functionality working and then integrate multi-user functionality. When we integrated multiple users, all we had to do was move the user specified code into its own function and create a thread with that function for each user. If we were to use select(), it would require more code rewrites. Within the main function for threads on the server is a call to the function HandleClientRequest, which deals with handling all requests that a client may send.

Client

The client begins its main activity by setting up the UI and displaying a text field and a spinner containing options for the *cap* command, as well as a button for sending the request to the server. The user enters a command (*list*, *diff*, *pull*, *cap*, or *leave*) and presses the button to send a request to the server. If the user wants to implement a *cap* command, they must select a cap size (in MB) along with the command. When the send request button is clicked, the client goes into the sendMessage() method within the Main Activity. In this method, the client creates an intent to send to the DisplayMessage Activity, and sends a string along with the intent. This string is the value returned from executing TCPSendMessage(), a child of asynchronous task. In this private class, the client first checks to see if it is connected. If not, it creates a new socket. It then writes a request message to the server and receives a response, which it then decodes and finds the request type before handling it.

For a *diff* request, the client gets all files from its local directory and computes an MD5 checksum for each, which it compares with the checksum values that the server sent.

For a *pull* request, the user sends another message to the server with the names of files it needs, and the server responds to start transferring the files. Once they are done transferring, the client returns the number of songs pulled and their names as a string to be displayed to the user.

For a *cap* request, the user gets a message with song names and responds to the server with a *pull* request that contains these song names. The client and server then begin file transfer. If a song name from the iTunes XML file is not in the server library, it skips over it and only sends files that are existing in its directory.

For a *leave* request, the client closes the open socket, calls finish() to end the main activity, and returns a message to notify the client that the connection has been closed. The app terminates after the client reads this message.

Input Handling and Message Framing

The basic format for sending a request to the server will begin with the client entering a request in the Android UI. This request will then be put into a Message object, which is then encoded as a tab-delimited string to send to the server. When the server receives the message, it tokenizes the string to get the information from it and stores this information in the respective fields of a *msg_t* struct. When the server responds, it encodes its sndInfo struct (*msg_t* type) to create a tab-delimited string that the client will receive and decode into a Message object.

Commands

list: Upon receiving a list request, the server will open its music directory and iterate through it, placing each song name into the filenames field of sndInfo (an instance of msg_t to be sent back to the client). After each song name, the server will add a delimiter ("|") so that the client can distinguish song names from each other. The server will then send the encoded struct back to the client, which will receive and decode it into a Message object. The decoding function in Message will store the file names in a string array, which will be iterated through and placed into a string to get a return value for sendMessage().

diff: Upon receiving a diff request, the server will iterate through its music directory in the same way it does for list. However, it will also compute a checksum for each file and place them in the cksums() field so that the client can compare its files effectively. The server will then encode and send the sndInfo struct, and the client will receive and decode it into a Message object. Using this, the client will call a fileCompare() function. This function will first crawl through the client's music directory and get a list of song names and compute checksums, then parse through the two lists of songs/checksums. Using nested for loops, the function will compare the server and client side checksums, copying the song names of any differences it finds to an ArrayList. This ArrayList will then be converted into a String array for the client, which will parse through it and return song names in the DisplayMessage Activity.

pull: The client will first send a diff request and get a response from the server. Using the songs found from diff (stored in rcvMessage), the client will send another message with a requestType of "pull", with the song names that need to be pulled from the server in the filenames field. The server goes through each file in this string and gets the file names and sizes. It then responds with all of the file sizes so the client knows when to close each file. For each file, the server sends raw data until it reaches the fileSize, then waits for a response from the client that it is done processing that file. It loops until there are no more file names left in the delimited string.

leave: The client sends a message with a request type of *leave*. The server sends back the message and closes the socket and the client does the same and finishes the activity.

cap: The client sends a cap request along with a cap value to the server, which then decodes the message and calls doCapServer() in order to parse the XML file and return a list of the most popular song names within the bandwidth limit. The client and server then begin file transfer in the same way that they did for *pull*.

Note: All files are expected to be mp3 format.

Structs - Server

```
struct msg_t {
        char request[12];
        char filenames[32][128];
```

int cksums[32];
int len;

};

Contains 4 fields:
request, a string containing the type of request to send the server (**list, diff, pull, cap leave**)
filenames, an array of strings containing song names delimited by "|"; used for **list**, **diff** and **pull**
cksums, an array of ints containing checksums for each song, used in **diff** and **pull**
len, an integer containing the length of the data being sent. In **list** and **diff**, *len* is equal to the number of songs in the message. In **pull** and **cap**, *len* is used to describe the length of file data being sent.

```
struct file_info_t {
        char filename[FILENAME_MAX];
        size_t filesize;
};
```
Used in **pull** and **cap**.

Contains 2 fields:
filename, the name of the file being sent
filesize, the size of the file being sent.

Objects - Client
Message: class containing all of the fields described in the server's *msg_t* struct, used client-side representation of message. Contains getters for all fields, as well as an encode() and decode() function to convert to and from the tab-delimited strings used for message transfer.