# Design and Architecture

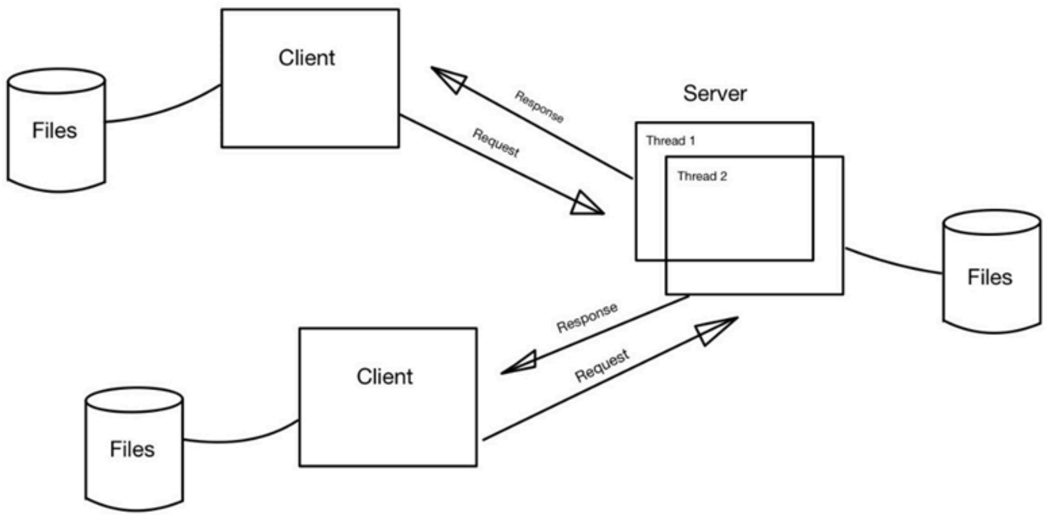
## Project Structure

- server.cpp - Implements the server logic.
- client.cpp - Implements the client interface and commands.
- makefile - Contains build instructions for the entire project.

## Required Directory Structure

- music/: This directory is on the server side and contains all the music files that are available for synchronization with the client.
- local_music/: This directory is on the client side and holds the music files that have been synchronized from the server or are available locally for synchronization.

## System Overview



## Client-Server Model

The application is built under a simple client-server model, where *many* clients communicate with a *single* server. Each client is an independent process and maintains a single, separate file folder to store its files. In order to communicate with multiple clients simultaneously, the server spawns a new thread for every new client connection. This design choice utilizes pthreads to manage concurrent client connections. Each thread is responsible for maintaining communication with its corresponding client and terminates itself when the client closes the connection.

We chose pthreads over select() for several reasons:

- Concurrency Management: pthreads allow each client connection to be handled by a separate thread, which provides a clear and isolated context for each client session. This isolation simplifies the management of each client's state and requests, ensuring that operations for one client do not interfere with another.
- Resource Utilization and Scalability: Although pthreads may consume more system resources compared to select(), they leverage the multi-core processors effectively by distributing the load across multiple cores. This approach enhances the server's ability to scale and handle numerous simultaneous connections without the bottleneck of a single-threaded I/O operation cycle.
- Simplicity in Implementation: Using pthreads simplifies the coding structure for handling multiple connections, as each connection is managed independently. This leads to a design that is easier to implement and debug compared to the event-driven model required when using select().

## Request and Response Model

### 1. LIST

The client sends a Request to the server with “LIST” specified as the command_type. The server then replies with a Response containing an array file_names of all its files file names. The client then prints these file names out.

### 2. DIFF

The client sends a Request to the server with “DIFF” specified as the command_type. The server then replies with a Response containing an array of file_names along with an array of corresponding checksums. The checksum is a compact, numeric representation of the file’s content, ensuring that even if files have the same name, any difference in content will result in a different checksum. The client then compares these checksums to its own checksums and prints out any files with checksums not present in the client folder.

### 3. PULL

The client performs a DIFF request as detailed above and iterates through each ‘missing’ file. For each ‘missing’ file, it sends a Request to the server with “PULL” specified as the command_type along with the file name as the parameter. The server then replies with a Response with the content of the requested file. The client receives the file content and saves it in its local folder, ensuring that its directory is synchronized with the server. The client continues this process for each missing file until all discrepancies are resolved.

### 4. LEAVE

The client sends a Request to the server with "LEAVE" specified as the command_type to indicate that it is disconnecting from the server. The server replies with a Response by closing the connection, and the client performs any necessary cleanup operations before terminating its execution.