# Peer-to-Peer Music Sharing System

**CSCI3280 Group Project Phase 2: 12.03.2023 - 02.05.2023 (deadline at 23:59)**

Late submission penalty: 10% daily deduction (maximum 30%).

DO NOT directly copy code from ***the internet / other groups*** if you want to pass the course.

## Overview

In this project, you are required to achieve **two** main goals.

- ***(Phase 1)*** Firstly, you are required to implement a graphical music player to play WAV audio files, control the playback, display music lyrics, and manage a music library.
- ***(Phase 2)*** Secondly, you are required to build a **Peer-to-Peer (P2P) system** for playing music (in the form of streaming) from remote computers. Each computer works as both client and server, which means you may get the audio data from other computers and share audio data for others to be downloaded.

**Language & Environment:** As the same as in Phase 1.

**Evaluation:** Your system should fulfill the basic requirements and have enough enhanced features. In the final version of your project, you need to submit your code and a project report (max 6 pages). Besides, on *April 25th, May 2nd, and May 3rd, you are required to give a 12-minute demonstration* of your project to show your implemented features, especially peer-to-peer data transfer.

Points of the basic and enhanced features are graded together based on the submitted version of Phase 2.

| ***Evaluation Metrics*** | ***Score (100% in total)*** |
|---|---|
| Basic Requirements: Phase I | 30% |
| Basic Requirements: Phase II | 30% |
| Enhanced Features (Phase I & II) | 25% |
| Demonstration & Report | 15% |

## Basic Requirements

### Network connection

Your program should be able to connect to other PCs using TCP/IP network stack. You can use any method to get the IP address of the connectable PCs except hard coding. (For example, manually inputting IP addresses, using a tracker server, or broadcasting are appropriate). ***The network should support at least three terminals.***

### Music Searching (Online)

Besides local music searching, network searching should be supported in this phase. The search interface should be the same one as those in phase 1.
What's more, your program should search the audio files not only in the local database but also in the database of the other PC connected to your program. ***All the results from the local and network databases should be displayed in the same list control of the UI***. Identical results (e.g., different computers have the same audio file) must be displayed only once.

### Availability Check

As the user may not know where the audio is, the program should check whether the audio file exists locally when the user selects audio from the search results. If the audio file exists, your program shall playback the audio directly. Otherwise, your program will stream the audio file from other computers.

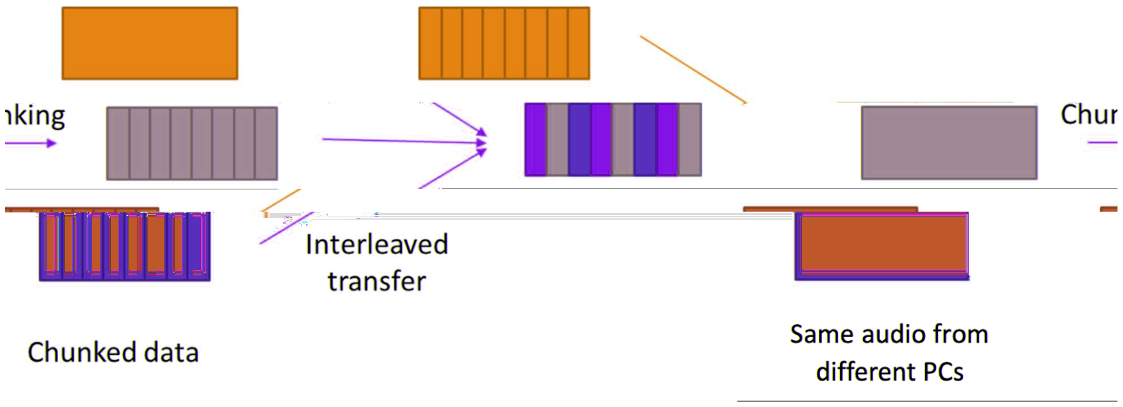
### Real-Time Audio Streaming

When the program is streaming from other computers, your program should automatically play the audio as soon as possible when it receives a piece of audio data (after a certain amount of buffering). ***You are only allowed to buffer no more than 50% of a music file before you play the music***.

## Peer-to-Peer Playing

Your program should be able to receive one audio file from at least two other computers simultaneously. ***The audio data from different computers should be played in an interleaving way.*** For example, suppose PC1 wants to play a file, and it cannot be found locally. It should get audio data from PC2 and PC3, the file is divided into (at least) 4 parts, and PC1 may get the first part from PC2, the second part from PC3, the third part from PC2, and the fourth part from PC3.

To verify the interleaving feature, you should also implement a function in your project to show your interleaving feature using images. We will give you three images in different colors but have the same file name in bitmap format with the same resolution. Let's say PC1 wants to download this image from other endpoints in an interleaving way; you must achieve that your data are completed but collected from different endpoints. The bottom figure shows an example of an interleaving feature.



## Sample Enhanced Features

1. Support streaming other audio formats
2. Support more than three clients
3. Support more than two sources
4. Do any other your own ideas to enhance the system

## Submission Guideline (Phase II)

Each group is required to submit one .ZIP file (CSCI3280_Group_X_Project.zip), including your source code (with an executable file if available), and your final report via blackboard **before May. 2nd 23:59**.

## Demonstration and Report

For this project, you need to do a demonstration of your program and submit a hard copy of the project report in the demonstration.

The report is a brief description, up to 6 pages, to describe your program. You must write down your team number, team member's name and student ID, workload division, and program's operation manual (README). You must also state which third-party libraries have been used in your program and what enhanced features you have implemented.

You need to do a demonstration in front of a tutor. In the demonstration, you should introduce every basic requirement you have fulfilled and every enhanced feature you have implemented clearly and efficiently. Please tell tutors if you have any special requirements on the library, tools, or resources. What's more, you will only have 12-min to demonstrate your program, and a mark will be deducted if you do your demonstration for more than 12 mins.

**The following requirements must also be followed:**

***Before the demonstration:***

- You can use any machines, including your PC, in the demonstration venue. Please be well prepared before you do the demonstration (e.g., setting up your environment, downloading the necessary resources, and preparing your own audio files other than .wav format).
- The demonstration starts when you run your program.

***During the demonstration:***

- You need to demonstrate all your program features.
- Tutors may ask you questions about your program, and your answers will affect your grades.
- You are not allowed to close/restart your program without permission.
- Unstable performance (e.g., No responses or unexpected results) may lead to mark deduction.

## Technical Handout - Will be explained in tutorials in detail

## Network Connections with Socket + TCP/UDP

***Protocol  (TCP/UDP)***

- TCP/IP (Transmission Control Protocol/Internet Protocol) is the set of protocols that governs communication over the internet. It is a standardized communication protocol used for transferring data between different computers and networks.
- An IP address is a unique identifier assigned to each device on a network that uses the Internet Protocol for communication. It consists of four numbers separated by dots, such as 192.168.0.1. IP addresses can be either static (permanently assigned to a device) or dynamic (assigned by a DHCP server).

***What is a port?***

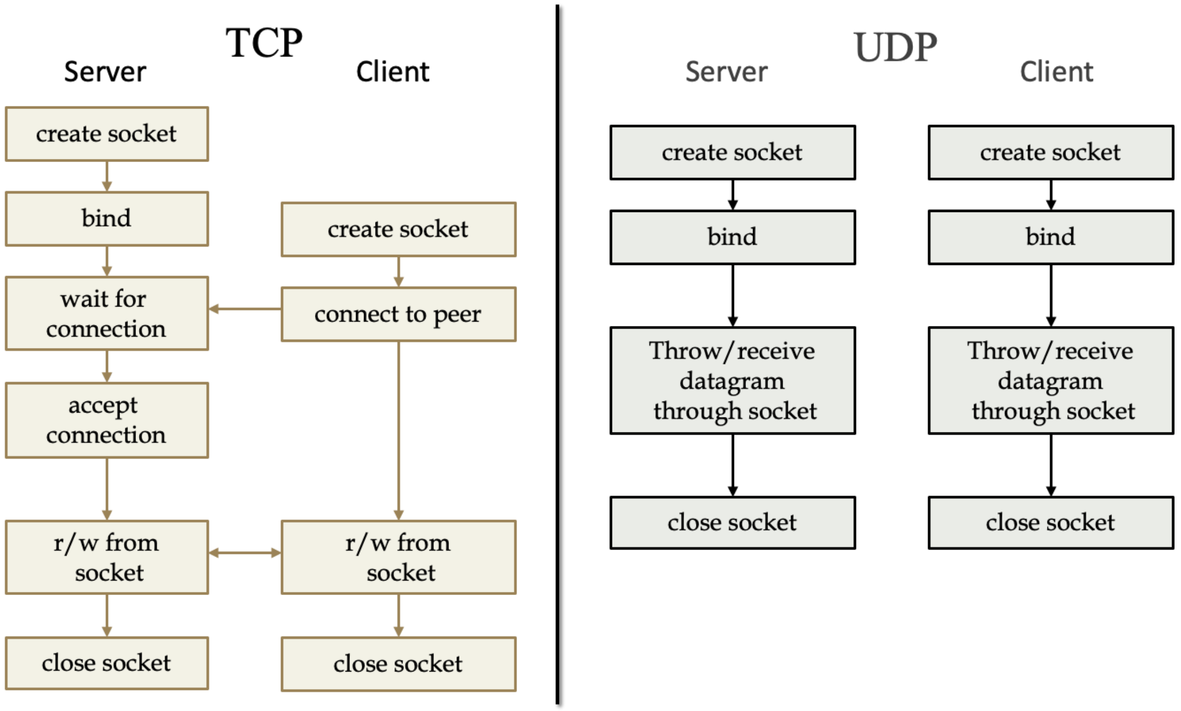
- A port is a communication endpoint used in computer networking. Ports are identified by a number and are used to differentiate between different network services running on the same device.

***Socket:***

- Socket is the endpoint of a communication channel, A network programming interface, abstracting away underlying mechanism
- On the Windows platform, we use WinSock (https://learn.microsoft.com/en-us/windows/win32/winsock/using-winsock)
- Two types of sockets (for two different transport layer protocols): SOCK_STREAM (TCP) / SOCK_DGRAM (UDP)

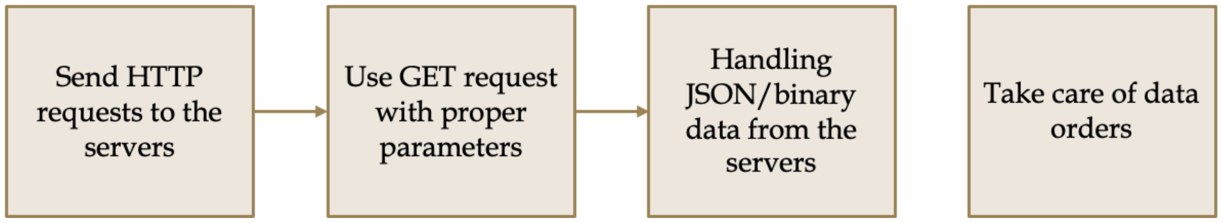
### Some useful functions for Sockets

```
   S                                                 S        DLL
     SAS         ORD         R          LP  SADA  A  SAD

   S
SOCKE

   S                       SOCKE                                            IP
         SOCKE
   L
           SOCKE


            SOCKE
   A

SOCKE          SOCKE


   R

         SOCKE
     SAAPI       SOCKE

   C
                SOCKE

   C                                           S       DLL
     SAC

   E

     SAG  L   E
```



*The pipeline of TCP/UDP data transfer.*

## Network Connections with Socket + TCP/UDP


Send HTTP requests to the servers
Use GET request with proper parameters
Handling JSON/binary data from the servers
Take care of data orders

**HTTP** is a protocol that stands for Hypertext Transfer Protocol. It is widely used to retrieve data from servers using Uniform Resource Identifiers (URI) or URLs. HTTP requests are initiated by clients and responded to by servers. HTTP is a *symmetry network*, a type of network architecture that operates in a way that every node in the network is both a client and server. This allows for more efficient and dynamic communication between nodes.

In HTTP, we can retrieve data using URIs. Here's an example:

http://pc1/get_data?filename=somnus.wav&start=4096&length=4096.

In this example, we request data from the "pc1" node. We are asking for a specific file named "somnus.wav". We also specify the start position and length of the data we want. Similarly, we can request data from other nodes in the network:

http://pc2/get_data?filename=somnus.wav&start=0&length=4096.

This request asks the "pc2" node to send data from the beginning of the file.
We can also check if a file exists in the network using the URL:

http://pc1/check_existence?filename=somnus.wav.

Depending on the request, the server will respond with the appropriate data fragments or status.

- To handle these requests, we will need to implement routing on our server. This means that we will have to define functions to handle different types of requests, such as reading and transferring file fragments or checking the existence of a file.
- When processing these requests, we will also need to extract HTTP GET parameters. In our examples, these are the parameters passed through the URI, such as the filename, start position, and length.

- Once we have processed the request, we will need to respond to the client. For text data, we can use plain text or JSON formats. For binary data, such as audio files, we will need to use the application/octet-stream content type in our header.
- To implement our HTTP server, we can use libraries such as libcurl or Pistache in C++, or Flask in Python.

## P2P Client and Multi-Threads.

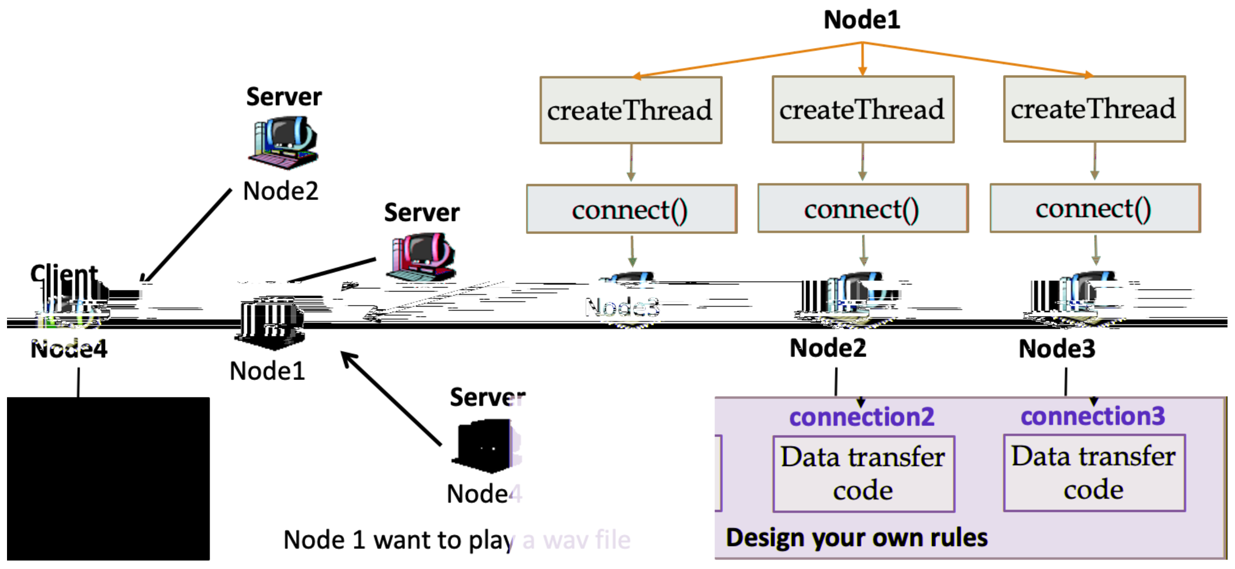
**Why do we need Multi-threads?** In audio playback, waveOutWrite() blocks everything. We need to do other stuff during the audio playback for real-time response.

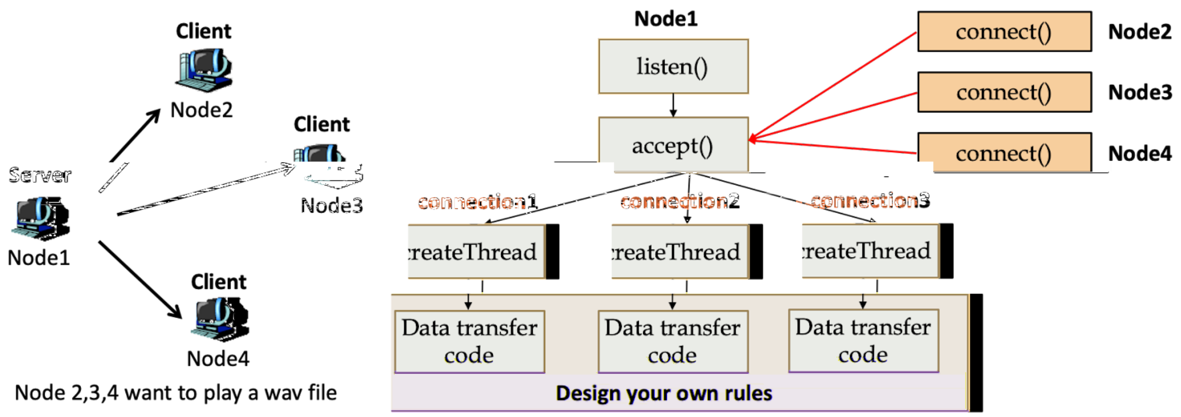
```
while (notEOF){
  loadDataIntoBuffer();       // fread/memcpy or others
  waveOutPrepareHeader(...);
  waveOutWrite(...);
  waitForSingleObject();    // block until playback finished
}
```

Also, in the socket programming, a single thread cannot accept the connection and receive data simultaneously.

```
while (1){
    client_sd = accept(...);  // blocked
    ...
}
while (1) {
    len = recv(...);  // blocked
    ...
}
```

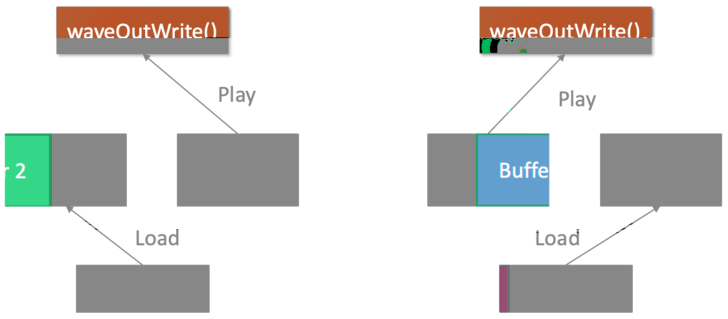
A **thread** is a sequence of such instructions within a program that can be executed independently of other code. The multi-threading design allows an application to do parallel tasks simultaneously.





The P2P process can be represented in the above two figures. When one node is requested for an audio file (***client***), it can create multiple threads to make connections for servers. Then multiple servers will send data chunks to the client for audio playback. When the single node is served as a server, it will listen to multiple connections. Each connection will create a thread and transfer data to each requesting client.

Multiple threads are also helpful for real-time streaming: we can simultaneously read and play with two buffers:



***STD::Thread ([https://cplusplus.com/reference/thread/thread](https://cplusplus.com/reference/thread/thread)):***

- Add #include <thread> to your source file
- Creation:

```
Std::thread t(void *(*start_routine)(void *), void *arg);
```

*void *(*start_routine)(void *):* the function this thread executes
*void *arg:* arguments to pass to thread function above

- Thread type: **std::thread**
- Join threads: **join()** - Suspends the calling thread to wait for successful termination of the thread specified as the first argument pthread_t thread with an optional *value_ptr data passed from the terminating thread's call to pthread_exit().

**Code Example:**

```
#include <iostream>
#include <thread>

using namespace std;

void hello(const char* input) {
    cout << input << endl;
}

int main() {
    thread t(hello, "hello world");
    t.join();
    return 0;
}
```

***Thread Communications:***

Thread Communications are essential in any concurrent program, where multiple threads execute simultaneously.

## 1. Global variables

Firstly, Global variables can be used to share data between threads. Global variables are variables that are declared outside any function and can be accessed by any function in the program. Here's an example of using a global variable for thread communication:

```
#include <iostream>
#include <thread>
using namespace std;

int globalVar = 0;

void threadFunction() {
    globalVar = 10;
}

int main() {
    thread t(threadFunction);
    t.join();
    cout << "Global variable value: " << globalVar << endl;
    return 0;
}
```

In this example, the `threadFunction()` updates the value of the `globalVar` variable, and the main thread prints its value after the thread has completed execution.

## 2. Pointers as thread arguments

Secondly, Pointers can be used as thread arguments to share data between threads. Pointers are variables that store the memory addresses of other variables. Here's an example of using pointers for thread communication:

```
void threadFunction(int* ptr) {
    *ptr = 10;
}

int main() {
    int var = 0;
    thread t(threadFunction, &var);
    t.join();
    cout << "Pointer value: " << var << endl;
    return 0;
}
```

In this example, the `threadFunction()` updates the value of the `var` variable indirectly, by using a pointer to its memory location.

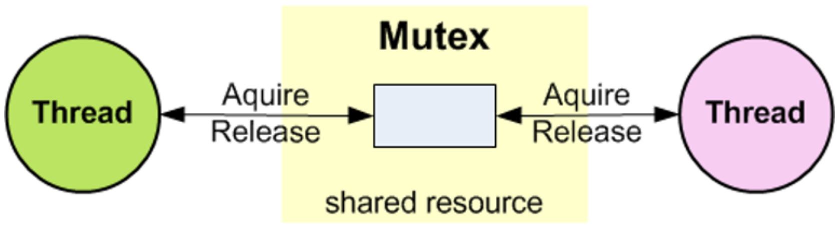
### 3. Take care of every WRITE operation

Thirdly, it's important to take care of every WRITE operation, since multiple threads may attempt to write to the same variable simultaneously, leading to data inconsistencies. Here's an example of a program that doesn't take care of WRITE operations:

```
int var = 0;

void threadFunction() {
    var++;
}

int main() {
    thread t1(threadFunction);
    thread t2(threadFunction);
    t1.join();
    t2.join();
    cout << "Variable value: " << var << endl;
    return 0;
}
```

In this example, the `threadFunction()` increments the value of the `var` variable. Since two threads are executing simultaneously and both are incrementing the same variable, the final value of the variable can be unpredictable.

4. Mutex: https://en.cppreference.com/w/cpp/thread/mutex



Lastly, a Mutex (short for Mutual Exclusion) can be used to protect shared resources from simultaneous access by multiple threads. A Mutex is a lock that only one thread can hold at a time, preventing other threads from accessing the resource until the lock is released. Here's an example of using a Mutex for thread communication:

```
int var = 0;
mutex m;

void threadFunction() {
    m.lock();
    var++;
    m.unlock();
}

int main() {
    thread t1(threadFunction);
    thread t2(threadFunction);
    t1.join();
    t2.join();
    cout << "Variable value: " << var << endl;
    return 0;
}
```

In this example, the `threadFunction()` uses a Mutex to protect the var variable from simultaneous access by multiple threads. The `m.lock()` statement acquires the Mutex lock, allowing only one thread to access the shared variable at a time. Once the update is complete, the `m.unlock()` statement releases the lock, allowing other threads to access the variable.