Windows CommAudio

**COMP4985 – Data Communications / Internetworking Comm Audio Project**

Albert Liao – [liao.a17@gmail.com](mailto:liao.a17@gmail.com)

Callum Styan – [callumstyan@gmail.com](mailto:callumstyan@gmail.com)

Steve Lo – [kashun\_lo@hotmail.com](mailto:kashun_lo@hotmail.com)

Darry Danzig – [darry.d@gmail.com](mailto:darry.d@gmail.com)

Table of Contents

[Introduction 3](#_Toc351364612)

[Purpose 3](#_Toc351364613)

[Minimum Requirements 3](#_Toc351364614)

[Optional Features 3](#_Toc351364615)

[Application Design 3](#_Toc351364616)

[Networking Overview 3](#_Toc351364617)

[Control Channel (TCP) 5](#_Toc351364618)

[Overview 5](#_Toc351364619)

[Details 5](#_Toc351364620)

[Flow Diagram 7](#_Toc351364621)

[Streaming Connection (UDP) 8](#_Toc351364622)

[Overview 8](#_Toc351364623)

[Details 8](#_Toc351364624)

[Flow Diagram 8](#_Toc351364625)

[File Transfer (TCP) 9](#_Toc351364626)

[Overview 9](#_Toc351364627)

[Purpose 9](#_Toc351364628)

[Flow Diagram 9](#_Toc351364629)

[Program Details 10](#_Toc351364630)

[File Structure 10](#_Toc351364631)

[Windows API 10](#_Toc351364632)

[Sound Library 10](#_Toc351364633)

[Appendix A 10](#_Toc351364634)

[File Transfer Psuedocode 10](#_Toc351364635)

# Introduction

## Purpose

The goal of the comm audio project is to create a set of Windows applications that will support streaming music from one server out to multiple clients over the network. The clients should be able to connect to a server of their choosing and be able to play the current music being sent out. Clients should also be able to download and upload music from and to the server as well as communicate through microphone.

## Minimum Requirements

The applications should have the following capabilities:

* The ability to select a server to connect to
* Ability for the server to stream to multiple clients
* Have a graphical user interface element for all actions
* Support two way microphone
* Send files across computers from client to the server and vice versa
* Stream audio from the server to play on the client in real-time
* User manual
* Test and verification document

## Optional Features

Here are some extra features we plan to add to our application:

* Ability to have a playlist
* Ability to show the current playing song
* Ability for client to control current song playing
* Ability to seek through a song forwards/backwards

# Application Design

## Networking Overview

For this application, we plan to use three channels for communication between the server and client. We went with this approach because we realize that there will be lots of data transfer going on between the applications that isn’t just streaming music; we will need to keep track of the song the server is currently playing, the actions that the client wants the server to perform, such as play / pause, the microphone interaction and also the file transfer.

The three types of channels that we will have are the following:

1. TCP Control Channel
   * This is the first connection that will be established between the server and the client program. It will be used for sending control information back and forth between the client and server such as the song list on the server (including any updates the server makes), a client request for a new song to be played and also seeking within the current song. We are using a TCP connection as it is beneficial to keep this line open while the applications are interacting, and we also don’t want to lose commands being sent from the client to the server.
2. UDP Streaming Channel
   * This connection will be used purely for streaming music. It will be established over a different port than what the control and file transfer channels will use. We chose UDP for this portion as even though it is undesirable to lose sounds packets over the network, the application should have a “real-time” feel to it; that is, if we are dropping packets, the song shouldn’t have to pause and wait while it retries sending the packet over and over again. It’s also beneficial to use UDP when sending to multiple clients, because the server application wouldn’t need to keep track of the multiple song states each client would be in if the connections were out of synch. We will be using multicasting to stream the same music to all of the connected clients at the same time.
3. TCP File Transfer Channel
   * This last connection is only opened when an upload/download request is made. The file transfer could technically be done over the TCP control channel, but it will be much cleaner if we have the applications use a separate channel. We will also be able to support multiple uploads / downloads at a time if we open a new channel for each new file upload/download request. The channels will be closed after each download/upload is finished. We are using TCP for this because we want to maintain the integrity of the files when we send them (we don’t want to lose ANY data).

## Control Channel (TCP)

### Overview

As stated previously, this channel will be used to send control information between the server and its clients such as the currently playing song and a seek request from a client. Commands that we intend to support are the following:

1. The ability for the server to send a client a song list of all the current songs in its playlist.
2. The ability for the server to send the client the name / song details of the currently playing song for display purposes.
3. The ability for the client to request a new song to be played.
4. The ability for the clients to send requests for the server to seek forwards or backwards in the song and also pause and play the song.
5. The ability for the clients to send requests for the server to jump to the beginning of the song, or a specific point in a song.

### Details

For each control signal being sent out, the packet will have a control id to identify what type of command it is, and the a payload right after the control id to provide details of the control signal.

**The ability for the server to send a client a song list of all the current songs in its playlist**

Sending the song list will be achieved by the server sending out a list of songs in the form of newline delimited strings containing the name of each song in its playlist. The client can then interpret this message and display the songs as a list in its user interface.

Packet:

Control ID – 1

Payload – Newline delimited strings of the song names that are currently on the server’s playlist.

**The ability for the server to send the client the name / song details of the currently playing song for display purposes.**

Whenever a new song is being played, or a new client connection is made, the server will send out new song details about the currently playing song so that the client can display things like song title and artist on its GUI.

Packet:

Control ID – 2

Payload – Newline delimited strings containing the following in this order: song name, song artist, album, song length.

**The ability for the client to request a new song to be played.**

When the client wants a new song to be played on the server, it will send the file name over the control channel and the server will play it from the beginning

Packet:

Control ID – 3

Payload – String containing the filename.

**Pausing a song.**

To pause a song, the client will simply send out a control packet with control id 4.

Packet:

Control ID – 4

Payload – None.

**Resuming a song.**

To resume a song, the client will simply send out a control packet with control id 5.

Packet:

Control ID – 5

Payload – None.

**The ability for the clients to send requests for the server to jump to the beginning of the song, or a specific point in a song.**

When the client wants a song to seek to a certain part of the song, it will send the server an integer from 1 to 100 which will tell the server what area of the file to start reading and streaming from. 1 being the beginning of the file and 100 being the end of the file.

Packet:

Control ID – 6

Payload – Integer containing portion of the file to seek to.

**The ability to seek forwards in the song.**

When the client wants the server to skip forwards in a song by a factor of 5, it will send a control packet with control id 7

Packet:

Control ID – 7

Payload – None.

**The ability to seek backwards in the song.**

When the client wants the server to skip backwards in a song by a factor of 5, it will send a control packet with control id 8

Packet:

Control ID – 8.

Payload – None.

### Flow Diagram

Server:



## Streaming Connection (UDP)

### Overview

Streaming will be performed through UDP by the server using multicasting to constantly send out song file data in a linear fashion. The SFML library supports being able to play these songs even without the complete file there and a file will be buffered and then played while this buffer is being filled.

### Details

The packet will contain the following information:

Packet:

SongPortion – integer from 1-100 containing area of song being played

Payload – serialized data containing portion of song file being played.

### Flow Diagram



## File Transfer (TCP)

### Overview

File transfer will be done through a simple TCP connection. When a file is to be uploaded by the client, the client will open a TCP connection with the server and transfer the file over. When the client wants to download a file, it will send a control line request containing the filename that it wants from the server and the server will perform an “upload” to the client.

### Details

The file transfer will be sent using packets of size 1024. The files will be sent from start to end until the file transfer is complete. Using TCP will ensure that the file will maintain its integrity (100% of the file will be there).

Packet:

MorePackets – Either true or false. When it is false, the packet does not contain any song data.

Payload – serialized data containing portions of the song to save in the application folder.

### Flow Diagram



# Program Details

## File Structure

The application will be split up into two program, server / client across two projects.

**Server:**

The server will have its GUI and network portion separated. The following file will be present:

* + resource.h (The menu and dialog box resources for the GUI).
  + CommGui.cpp (Contains the handlers for the GUI portion).
  + Network.cpp (The network portion).
  + Main.cpp (The driver for the whole program.)

**Client:**

The server will have its GUI and network portion separated. The following file will be present:

* + resource.h (The menu and dialog box resources for the GUI).
  + Client.cpp (Contains the handlers for the GUI portion).
  + Network.cpp (The network portion).
  + Client\_main.cpp (The driver for the whole program.)

## Windows API

We will be using the WinSock2 API from Windows to perform the networking aspect of this project. All sockets will be using completion routines as suggested to perform their actions.

## Sound Library

We will be using SFML to play streaming sound on the clients. This library is very lightweight and supports a variety of file types to be played such as MP3 and wmv.

# Appendix A

## File Transfer Psuedocode

**Frames:**  
  
**Request Header:**  
[5 bytes: *header*]  
 *header* = [4 bytes: size of *request\_data*][1 byte: *command*]

*command: 1 =* Get*, 2 =* FileListing, 3 Put

**Requests:**  
 Get:

- request to receive a file from the server

FileListing:

- request for file names available to be received. These are files in the default directory

Put:

- request to send a file to the server

**General Request frames:**   
General form:

[*header*][*request\_data*]

= [4 bytes: sizeof (*request\_data)*][1 byte: *command*][*request\_data*]

Get:   
 [*header*][*filename*]

=[string\_length\_of (*filename)* + 1][1][*filename*]

*filename* = null terminated string.

FileListing:  
 [0][2]  
  
Put:  
 [*header*][*file\_meta\_data*][*file\_contents*]  
 =[size of *filedata*][3][*file\_meta\_data*][*file\_contents*]

*file\_meta\_data*=[string\_length\_of (*filename*) + 1][*filename*][4 bytes: *file\_size*]

*filename* = null terminated string  
**Response Frames**:  
  
Get:   
   
 [*header*][*file\_contents*]  
 header = [1 byte *success*][4 bytes: *error\_code*][4 bytes: *filesize*]:

*success* = true or false

*error\_code* = win32 error codes

( <http://msdn.microsoft.com/en-us/library/windows/desktop/ms681381(v=vs.85).aspx>)

*filesize* = size of the file

FileListing:

[a ‘\n’ separated list of filenames terminated by ‘\0’]

Put:

[1 byte: *success*]

*success* = true or false;

**Client:**  
Main function:  
  
download completion routine{ //gets the header  
   
}  
  
helper download completion routine {  //gets the actual data being sent  
 make sure we’ve received data  
 read data from buffer  
   
 check the amount of data we’ve received against the  
 size of the file we’re downloading (from the header)  
  
 if we’ve received it all  
 clean up (close socket, free data, etc.)  
 save file  
 return  
 else  
 receive more data, use the same completion routine  
}  
  
connect to server{  
 set up socket for TCP  
 set up sockaddrin for server (host/ip, port, etc.)  
  
 connect using the server sockaddrin  
 error check  
}  
  
  
send request{  
 set up socket info struct  
 copy request into buffer  
  
 send request (using send request completion routine)  
}  
  
send request routine{  
 update amount of data sent based on last send call  
  
 check if we have sent all the data for the request  
 if we have  
 clean up  
 return  
 else  
 send again using the same completion routine  
}  
  
download file{ //could be used on server as well  
 set up socket info struct  
 connect to server function call

//we create a new tcp socket connection for each file download

error check

set up request (name of file, etc.)

send request (uses completion routines outline above)

call sleepex to put this thread in an alertable state

}  
  
  
**Server:**  
start server {  
 listen for connections  
}  
  
for each connection in new thread:

receive data from client until request frame has been received.

determine the number of bytes that the request  frame

}  
  
send file listing{

set up socket info struct

put filenames, size, etc. into buffer

send using file listing routine

}  
  
file listing routine{  
 update variable keeping track of bytes sent  
  
 check if we have sent all the data for the file listings  
 if we have  
 close socket, free data  
 return  
 else  
 send again using same completion routine  
 error check  
}  
  
file transfer routine{ //for transfering a file requested by the client  
 set up socket info struct  
  
 update variable keeping track of bytes sent  
 check if we have sent all the data for the requested file  
 if we have  
 close socket  
 indicate that the file was transferred to the client  
 return  
   
 if there is not data in the buffer  
 fill the buffer with data from the file  
  
 send the data using this completion routine  
}  
  
file transfer { //called when we receive a request for a file  
 set up socket info struct  
  
 make sure that the file being requested is in the directory  
 we’re using for the server files (current directory or specified)  
  
 call file transfer routine  
}