February 16, 2015

Rhea Lauzon // Jeff Bayntun // Michael Chimick // Julian Brandrick

4O

Data Communications (Comp 4985)

Comm Audio

Contents

[Requirements 2](#_Toc412569645)

[Server: 2](#_Toc412569646)

[Client: 2](#_Toc412569647)

[Specifications: 3](#_Toc412569648)

[Multicasting 3](#_Toc412569649)

[Control Channel 3](#_Toc412569650)

[Data Channel (Peer-To-Peer) 3](#_Toc412569651)

[Voice Chat (Peer-To-Peer) 3](#_Toc412569652)

[Sending File 3](#_Toc412569653)

[State Flow Diagrams 4](#_Toc412569654)

[Server Side (High Level) 4](#_Toc412569655)

[Client Side (High Level) 5](#_Toc412569656)

[Client (Peer-To-Peer) 6](#_Toc412569657)

[Message Protocol 6](#_Toc412569658)

[Messages: 6](#_Toc412569659)

[Pseudocode 8](#_Toc412569660)

# Requirements

* Create an audio streaming program
* Able to send sound data using UDP
* Must transfer data between two windows workstations and play the music
* Must have a Windows Interface
* Must be able to save and retrieve sound files provided
* The default sound file shall be in the .wav format
* Two way microphone support must work
  + Can between any two machines
* Multicasting capability
* Completion routines
* Server and client can be written as different programs

### Server:

* Able to transfer and play sound
* Sends the same data to every client at the same time (streaming) like radio
* Sends peer to peer music (for download)
* Can specify subnet address and port

### Client:

* Must be able to connect to a known remote server/workstation
* Able to transfer and play sound
* Client slightly behind server for buffering
* Able to download songs
* Can specify an IP and port

## Specifications:

### Multicasting

* Specified subnet address on both ends

## Control Channel

* Hard coded control channel port num
* Get the IP from fetching it from the multicast

### Data Channel (Peer-To-Peer)

* UDP data channel both ways
* Port via control channel + 1
* IP from multicast

### Voice Chat (Peer-To-Peer)

* Separate data channel (UDP) from data-channel

### Sending File

* Via secondary TCP channel

# State Flow Diagrams

## Server Side (High Level)



## Client Side

### High Level



### Client (Peer-To-Peer)



# Message Protocol

There are several messages that are sent between the client and server via the control channel once it is established. The control channel is established after the multicast is established. The following is a description of all messages that will be sent between the two and their description.

Message Format: MESSAGE\_TYPE~DATA`

The message type and data are delimited by a ‘~’ for separation and is not part of either. The end of the message is delaminated by a ‘`’ which is also not part of the data.

## Messages:

Color Key:

|  |  |
| --- | --- |
|  | Server->client |
|  | Client->Server |
|  | Either side |

|  |  |  |
| --- | --- | --- |
| MESSAGE\_TYPE | DATA | Description |
| END\_CONNECTION | No data | Message from client to server or server to client specifying they will be disconnecting and the control channel will be removed |
| MIC\_CONNECTION | No data | Message from client to server or another client for mic connection |
| SONG\_REQUEST | Name of song to be played | Message from client specifying the peer-to-peer song that they would like to listen. This should trigger a UDP connection between the client and server for the transfer of the requested song. |
| SAVE\_SONG | Name of song to be saved | Sent by client to the server to request a song to be saved. |
| LIBRARY\_INFO | Name of each song that is available from the server. Each song should be separated by a ‘|’, and the artist and song name are separated by a ‘^’  EX: Imagine^John Lennon|Let It Be^The Beatles | Message from server to client upon multicast connection. The server sends the name and artist of all songs available to the client. This data is then used to display a choice of songs for peer-to-peer listening. |
| NOW\_PLAYING | Name of the song playing current with the corresponding information separated by a ‘^’  Format: name^artist^album^length  EX: Imagine^John Lennon^Imagine^3:01 | Message sent from server to client for the song current being played. |
| END\_SONG | Empty | Server to clients indicating the current song is done |
| CURRENT\_LISTENERS | List of IPs of currently listening devices  Format: IP|IP|IP | Server to client to specify all currently listening clients and its own IP for microphone capabilities. |

# Pseudocode

## Server

Initialize Server function

{

Set up GUI

Initialize Multicast // details to come later

Load Songs

create Accept thread

Start multicasting on random until server shuts down

}

Accept Thread function

{

while true

Create a socket and Listen on SERVER TCP LISTEN PORT

when a new connection arrives, validate

if valid, create new Session to handle connection

}

Create Session function

{

allocate new session structure

initialize control socket

create control threads: control, mic send, mic rcv, send file

create semaphore for sending file // and any other session sems if needed.. update

store client network info in session// sockaddr\_in

add session to map of sessions // may need sem for this, used by multiple threads

}

Control Thread function

{

establish session from void parameter

set up callback for tcp received to control routine//maybe do a send instead, with songlist

enter alertable state and wait for control socket to receive

}

ControlRoutine

**{**

establish session from socket used for transfer

if received data

if file transfer requested, start send with protocol requested

if close start cleanup

if improper message send error message

if sending

if still data to send, send again

else prepare for receive // do wsa receive with callback to this function

}

Send Thread

{

get session structure from void parameter

enter forever loop

wait for send semaphore

create tcpsocket to send on

open file, get contents into char\* // need semaphore for filename access with control?

connect socket

post first send with callback to sendRoutine

enter forever loop // might be able to just work this into the original wait

enter alertable state

if file finished sending

close socket

signal send semaphore

//adjust session data??

Break

}

Send Routine

{

get session based on socket

adjust pointers so next chunk of file is sent

make send call with this function as callback

// need to coordinate end of send somehow

}

## Client

### Client Multicast

### Client Peer-To-Peer

display songlist function

{

for each song in the list of songs available from the server

{

add the song and artist name to a clickable list item on GUI

update GUI

}

}

request song stream function

{

get the song name from the GUI item clicked

generate a control message for song request with the message data set to the song name

open the UDP channel for receiving data

send the control message on the TCP control channel to the server

call the unregister from multicast function

call the receive song information function

}

receive song information function

{

get the song name, artist, and album

set the GUI track player with the data

call the play single song function

}

play single song function

{

while the song has not finished

receive song bytes from the UDP channel and store them in the buffer

play the buffer

close the UDP channel with the server

register for multicast

}

### Client File Transfer

request song download function

{

get the song name from the GUI item selected

generate a control message for SAVE\_SONG with the message data set to the song name

create file transfer thread

send the control message on the TCP control channel to the server

}

File transfer thread

{

while true

Create a socket and listen for TCP connections

when a new connection arrives, validate

if valid, create new connection

begin receiving file data until all data is received

}

### Client Microphone

## Helpers

## Unregister Multicast

unregister from multicast function

{

set the socket to drop membership

}