February 16, 2015

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

4O

Data Communications (Comp 4985)

Comm Audio

Contents

[Requirements 2](#_Toc413174756)

[Server: 2](#_Toc413174757)

[Client: 2](#_Toc413174758)

[Specifications: 3](#_Toc413174759)

[Multicasting 3](#_Toc413174760)

[Control Channel 3](#_Toc413174761)

[Data Channel (Peer-To-Peer) 3](#_Toc413174762)

[Voice Chat (Peer-To-Peer) 3](#_Toc413174763)

[Sending File 3](#_Toc413174764)

[State Flow Diagrams 4](#_Toc413174765)

[Server Side (High Level) 4](#_Toc413174766)

[Client Side 5](#_Toc413174767)

[High Level 5](#_Toc413174768)

[Client (Peer-To-Peer) 6](#_Toc413174769)

[Client (File Download) 7](#_Toc413174770)

[Message Protocol 7](#_Toc413174771)

[Messages: 7](#_Toc413174772)

[Pseudocode 9](#_Toc413174773)

[Server 9](#_Toc413174774)

[Client 11](#_Toc413174775)

[Client Multicast 11](#_Toc413174776)

[Client Peer-To-Peer 11](#_Toc413174777)

[Client File Transfer 12](#_Toc413174778)

[Client Microphone 12](#_Toc413174779)

[Helpers 12](#_Toc413174780)

[Unregister Multicast 12](#_Toc413174781)

# 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 (Voice Chat)



### Client (Peer-To-Peer)



### Client (File Download)



# 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. |
| PLAY\_SONG | Name of the song to be placed via unicast  EX: | Message from server to client of a song requested |
| 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 list of song titles and locations

create Accept thread

Start multicasting on random until server shuts down

}

Accept Thread function

{

create sessionMap semaphore, initialize to 1

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

}

CreateSession function

{

allocate new session structure

initialize control socket

create threads in suspend state: control, mic send, mic rcv, send file

create sendFile semaphore, initialize to 0

store client network info in session

access sessionMap Semaphore

add session to map of sessions

release sessionMap semaphore

start threads

}

ControlThread function

{

establish session from input parameter

make TCP Receive call with controlRoutine as call-back

while true

enter alertable state and wait for control socket to receive

}

Control Routine

**{**

establish session from socket used for transfer

if received data

if file transfer requested,

set filename in session object

signal sendFile semaphore

if close start cleanup

if improper message send error message

if sending

if still data to send, send again

else do TCP Receive call with callback to ControlRoutine

}

Send Thread

{

get session structure from parameter

enter forever loop

wait for sendFile semaphore

set session sending flag to true

create socket to send on

open file, load contents into memory

connect socket

post first send with callback to sendRoutine

enter forever loop

enter alertable state

if session sending flag is false

close socket

signal sendFile semaphore

Break

}

Send Routine

{

get session based on socket

adjust pointers so next chunk of file is sent

if whole file is sent

set session sending flag to false

return

make send call with this function as callback

}

Multicast Thread function

{

Start Multicast

while server not in shut down state

run Music Multicast

Stop Multicast

}

Start Multicast function

{

initialize resources

start multicast session

}

Music Multicast function

{

choose random song from song folder

set currently playing song to picked song

load song

set up callback for UDP music sending to Multicast Routine

}

Multicast Routine function

{

send audio file until it has all been sent

}

Stop Multicast function

{

communicate end of session to all clients

end multicast session

clean up used resources

}

## Client

Idle function

{

Initialize GUI

Call join multicast function

}

### Client Multicast

Join Multicast function

{

Add membership to multicast session

Start recieving thread

Call playing function

Make sure other thread is dead

Kill thread

}

/\*

The following functions require these initial conditions:

binary semaphore mRecv = 1

binary semaphore mPlay = 0

semaphore mBuffer = size of multicastBuffer

\*/

Multicast Recv function

{

Open udp listening socket

while die signal not given

{

wait for data

if the data is song data

wait semaphore mBuffer

wait semaphore mRecv

place data into multicastBuffer

signal semaphore mRecv

signal semaphore mPlay

if the data is library information

update songs list and GUI elements

if the data is client information

update list of clients and update GUI

if the data is now playing

update now playing GUI

}

cleanup

kill thread

}

Multicast Play function

{

open local speakers

while die signal not given

{

wait semaphore mPlay

wait semaphore mRecv

get data from multicastBuffer

signal semaphore mRecv

signal semaphore mBuffer

play data on speakers

}

cleanup

// don't kill, this is the parent thread

}

### 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 Voice Chat

// Should be started as a thread

Start Voice Chat function

{

Start sending thread

Start recieving thread

Start recording thread

Call sound playing function

Make sure other threads are dead

Kill thread

}

/\*

The following functions require these initial conditions:

binary semaphore OutRecord = 1

binary semaphore OutSend = 0

semaphore OutBuffer = size of voiceOutBuffer

binary semaphore InRecv = 1

binary semaphore InPlay = 0

semaphore InBuffer = size of voiceInBuffer

\*/

// All the following functions would be in a separate thread

Voice Record function

{

open local microphone

while die signal not given

{

record data

wait semaphore OutBuffer

wait semaphore OutRecord

place data into voiceOutBuffer

signal semaphore OutRecord

signal semaphore OutSend

}

cleanup

kill thread

}

Voice Send function

{

open udp sending socket

while die signal not given

{

wait semaphore OutSend

wait semaphore OutRecord

get data from voiceOutBuffer

signal semaphore OutRecord

signal semaphore OutBuffer

send data on socket

}

cleanup

kill thread

}

Voice Recv function

{

Open udp listening socket

while die signal not given

{

wait for data

wait semaphore InBuffer

wait semaphore InRecv

place data into voiceInBuffer

signal semaphore InRecv

signal semaphore InPlay

}

cleanup

kill thread

}

Voice Play function

{

open local speakers

while die signal not given

{

wait semaphore InPlay

wait semaphore InRecv

get data from voiceInBuffer

signal semaphore InRecv

signal semaphore InBuffer

play data on speakers

}

cleanup

// don't kill, this is the parent thread

}