February 16, 2015

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

4O

Data Communications (Comp 4985)

Comm Audio

Contents

[Requirements 2](#_Toc416291099)

[Server: 2](#_Toc416291100)

[Client: 2](#_Toc416291101)

[Specifications: 3](#_Toc416291102)

[Multicasting 3](#_Toc416291103)

[Control Channel 3](#_Toc416291104)

[Data Channel (Peer-To-Peer) 3](#_Toc416291105)

[Voice Chat (Peer-To-Peer) 3](#_Toc416291106)

[Sending File 3](#_Toc416291107)

[State Flow Diagrams 4](#_Toc416291108)

[Server Side (High Level) 4](#_Toc416291109)

[Client Side 5](#_Toc416291110)

[High Level 5](#_Toc416291111)

[Client (Voice Chat) 6](#_Toc416291112)

[Client (Peer-To-Peer) 7](#_Toc416291113)

[Client (File Download) 8](#_Toc416291114)

[Message Protocol 8](#_Toc416291115)

[Messages: 8](#_Toc416291116)

[Pseudocode 10](#_Toc416291117)

[Server 10](#_Toc416291118)

[Client 13](#_Toc416291120)

# 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 |
| 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 for downloading, multicast, or unicast |
| 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

1. Initialize Server function
2. {
3. Load list of song titles and locations
4. Initialize Multicast thread // details below
5. create Accept thread
6. }
7. CreateListen function
8. {
9. create SESSIONS semaphore, initialize to 1
10. create a socket and Listen on SERVER TCP LISTEN PORT
11. while true
12. when a new connection arrives, create new Session to handle connection
13. allocate new session structure
14. }
15. CreateSession function
16. {
17. wait on SESSIONS semaphore
18. add allocated session structure to sessionMap
19. signal on SESSIONS semaphore
20. create control thread
21. }
22. ControlThread function
23. {
24. establish session from input parameter
25. create finishedTransfer semaphore, initialize to 0
26. create userChanged semaphore, initialize to 0
27. send song list
28. send name/artist/album of currently playing song
29. signal clients on their userChanged semaphore
30. make TCP Receive call with controlRoutine as call-back
31. while true
32. enter alertable state and wait for the finishedTransfer and userChanged semaphores as well as WSA Callbacks
33. if userChanged signalled
34. send new user list
35. if finishedTransfer signaled //unicast/TCP song finished
36. send END\_SONG message
37. if error occurs
38. clean up session
39. }

## Server

1. Control Routine
2. {
3. if received data
4. if unicast requested
5. wait on session semaphore
6. establish session from socket used for transfer
7. signal on session semaphore
8. give session structure the requested filename
9. create unicast thread with session
10. if TCP file requested
11. create TCP file thread
12. if close
13. start cleanup
14. if improper message
15. start cleanup
16. if sending
17. if still data to send, send again
18. else do TCP Receive call with callback to ControlRoutine
19. }
20. Unicast Thread
21. {
22. create UDP socket
23. pass socket to waitingClients queue
24. start sending the song through to the client
25. when sending is finished, signal on finishedTransfer semaphore
26. terminate Unicast thread
27. }
28. TCPFilefunction
29. {
30. get session based on socket
31. initialize TCP socket
32. send file stream
33. when sending is finished, signal on finishedTransfer semaphore
34. terminate TCPFile thread
35. }
36. CleanupSession function
37. {
38. wait on SESSIONS semaphore
39. close finishedTransfer semaphore
40. close userChanged semaphore
41. signal on SESSIONS semaphore
42. send new user list
43. exit session control thread
44. }

## 

1. Multicast Thread function
2. {
3. initialize multicast socket
4. create multicast session
5. set time to live
6. disable loopback
7. initialize multicast song list
8. start playing song over multicast socket
9. }
10. Multicast Thread function
11. {
12. Start Multicast
13. while server not in shut down state
14. run Music Multicast
15. Stop Multicast
16. }
17. Start Multicast function
18. {
19. initialize resources
20. start multicast session
21. }
22. Music Multicast function
23. {
24. choose random song from song folder
25. set currently playing song to picked song
26. load song
27. set up callback for UDP music sending to Multicast Routine
28. }
29. Multicast Routine function
30. {
31. send audio file until it has all been sent
32. }
33. Stop Multicast function
34. {
35. communicate end of session to all clients
36. end multicast session
37. clean up used resources
38. }

## Client

1. Idle function
2. {
3. Initialize GUI
4. Call join multicast function
5. }
6. StartMulticast
7. {
8. Start the multicast thread
9. }
10. EndMulticast
11. {
12. Drop the multicast membership
13. End the multicast thread
14. }
15. JoinMulticast
16. {
17. Open the socket
18. Set any socket options
19. Bind the socket
20. Add the socket to the multicast session
21. Start the sound output thread - playMulticastSong
22. Call the recieve multicast function
23. }
24. receiveMulticastData
25. {
26. Forever loop
27. Receive data from the server
28. Place the data into the circular buffer
29. }
30. MultiWaveCallback // called when a buffer has finished playing
31. {
32. If the buffer is done playing
33. add the buffer to the end of the output queue
34. }
35. updateVolume // takes a number parameter
36. {
37. Set the volume to the new number
38. }
39. playMulticastSong
40. {
41. Initialize the waveformat settings
42. Open the local output device
43. Add the buffers to the output queue
44. Wait for some of the buffer to fill
45. Start the sound output
46. }
47. getCircularBuffer
48. {
49. return the multicast buffer
50. }
51. closeAudio
52. {
53. Close the output device
54. Cleanup the buffers
55. }
56. display songlist function
57. {
58. for each song in the list of songs available from the server
59. {
60. add the song and artist name to a clickable list item on GUI
61. update GUI
62. }
63. }
64. request song stream function
65. {
66. get the song name from the GUI item clicked
67. generate a control message for song request with the message data set to
    1. the song name
68. open the UDP channel for receiving data
69. send the control message on the TCP control channel to the server
70. call the unregister from multicast function
71. call the receive song information function
72. }
73. receive song information function
74. {
75. get the song name, artist, and album
76. set the GUI track player with the data
77. call the play single song function
78. }
79. play single song function
80. {
81. while the song has not finished
82. receive song bytes from the UDP channel and store them in the buffer
83. play the buffer
84. close the UDP channel with the server
85. register for multicast
86. }
87. request song download function
88. {
89. get the song name from the GUI item selected
90. generate a control message for SAVE\_SONG with the message data set to
    1. the song name
91. create file transfer thread
92. send the control message on the TCP control channel to the server
93. }
94. File transfer thread
95. {
96. while true
97. Create a socket and listen for TCP connections
98. when a new connection arrives, validate
99. if valid, create new connection
100. begin receiving file data until all data is received
101. }
102. VoiceChat function
103. {
104. start device VoicePlay
105. create dialog box for voice chat
106. }
107. class Player
108. {
109. InitPlayer function
110. {
111. setup audio player device
112. create UDP socket
113. setup callback for PlayData
114. }
116. StartVoicePlay function
117. {
118. start device playback
119. bind UDP socket to VOICE\_CHAT port
120. }
122. StopVoicePlay function
123. {
124. stop audio player playback
125. }
127. PlayData function
128. {
129. while socket has receivable data
130. receive data from socket
131. write data to audio device
132. }
133. }

1. class Microphone
2. {
3. InitMicrophone function
4. {
5. setup microphone device
6. create UDP socket
7. }
9. StartVoice function
10. {
11. setup socket for receiving from client
12. write received data from the socket to the microphone
13. }
15. StopVoice function
16. {
17. stop the microphone
18. }
19. }
20. Voice Recv function
21. {
22. Open udp listening socket
23. while die signal not given
24. {
25. wait for data
26. wait semaphore InBuffer
27. wait semaphore InRecv
28. place data into voiceInBuffer
29. signal semaphore InRecv
30. signal semaphore InPlay
31. }
32. cleanup
33. kill thread
34. }
35. Voice Play function
36. {
37. open local speakers
38. while die signal not given
39. {
40. wait semaphore InPlay
41. wait semaphore InRecv
42. get data from voiceInBuffer
43. signal semaphore InRecv
44. signal semaphore InBuffer
45. play data on speakers
46. }
47. cleanup
48. // don't kill, this is the parent thread
49. }