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. |
| 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

1. Initialize Server function
2. {
3. Set up GUI
4. Initialize Multicast // details to come later
5. Load list of song titles and locations
6. create Accept thread
7. Start multicasting on random until server shuts down
8. }
9. Accept Thread function
10. {
11. create sessionMap semaphore, initialize to 1
12. while true
13. Create a socket and Listen on SERVER TCP LISTEN PORT
14. when a new connection arrives, validate
15. if valid, create new Session to handle connection
16. }
17. CreateSession function
18. {
19. allocate new session structure
20. initialize control socket
21. create threads in suspend state: control, mic send, mic rcv, send file
22. create sendFile semaphore, initialize to 0
23. store client network info in session
24. access sessionMap Semaphore
25. add session to map of sessions
26. release sessionMap semaphore
27. start threads
28. }
29. ControlThread function
30. {
31. establish session from input parameter
32. make TCP Receive call with controlRoutine as call-back
33. while true
34. enter alertable state and wait for control socket to receive
35. }
36. Control Routine
37. {
38. establish session from socket used for transfer
39. if received data
40. if file transfer requested,
41. set filename in session object
42. signal sendFile semaphore
43. if close start cleanup
44. if improper message send error message
45. if sending
46. if still data to send, send again
47. else do TCP Receive call with callback to ControlRoutine
48. }
49. Control Routine
50. {
51. establish session from socket used for transfer
52. if received data
53. if file transfer requested,
54. set filename in session object
55. signal sendFile semaphore
56. if close start cleanup
57. if improper message send error message
58. if sending
59. if still data to send, send again
60. else do TCP Receive call with callback to ControlRoutine
61. }
62. Send Thread
63. {
64. get session structure from parameter
65. enter forever loop
66. wait for sendFile semaphore
67. set session sending flag to true
68. create socket to send on
69. open file, load contents into memory
70. connect socket
71. post first send with callback to sendRoutine
72. enter forever loop
73. enter alertable state
74. if session sending flag is false
75. close socket
76. signal sendFile semaphore
77. Break
78. }
79. Send Routine
80. {
81. get session based on socket
82. adjust pointers so next chunk of file is sent
83. if whole file is sent
84. set session sending flag to false
85. return
86. make send call with this function as callback
87. }

## 

1. Multicast Thread function
2. {
3. Start Multicast
5. while server not in shut down state
6. run Music Multicast
8. Stop Multicast
9. }
10. Start Multicast function
11. {
12. initialize resources
13. start multicast session
14. }
15. Music Multicast function
16. {
17. choose random song from song folder
19. set currently playing song to picked song
21. load song
23. set up callback for UDP music sending to Multicast Routine
24. }
25. Multicast Routine function
26. {
27. send audio file until it has all been sent
28. }
29. Stop Multicast function
30. {
31. communicate end of session to all clients
33. end multicast session
35. clean up used resources
36. }
37. Multicast Thread function
38. {
39. Start Multicast
41. while server not in shut down state
42. run Music Multicast
44. Stop Multicast
45. }
46. Start Multicast function
47. {
48. initialize resources
49. start multicast session
50. }
51. Music Multicast function
52. {
53. choose random song from song folder
55. set currently playing song to picked song
57. load song
59. set up callback for UDP music sending to Multicast Routine
60. }
61. Multicast Routine function
62. {
63. send audio file until it has all been sent
64. }
65. Stop Multicast function
66. {
67. communicate end of session to all clients
69. end multicast session
71. clean up used resources
72. }

## Client

1. Idle function
2. {
3. Initialize GUI
4. Call join multicast function
5. }
6. Join Multicast function
7. {
8. Add membership to multicast session
9. Start recieving thread
10. Call playing function
11. Make sure other thread is dead
12. Kill thread
13. }
14. /\*
15. The following functions require these initial conditions:
16. binary semaphore mRecv = 1
17. binary semaphore mPlay = 0
18. semaphore mBuffer = size of multicastBuffer
19. \*/
20. Multicast Recv function
21. {
22. Open udp listening socket
23. while die signal not given
24. {
25. wait for data
26. if the data is song data
27. wait semaphore mBuffer
28. wait semaphore mRecv
29. place data into multicastBuffer
30. signal semaphore mRecv
31. signal semaphore mPlay
32. if the data is library information
33. update songs list and GUI elements
34. if the data is client information
35. update list of clients and update GUI
36. if the data is now playing
37. update now playing GUI
38. }
39. cleanup
40. kill thread
41. }
42. Multicast Play function
43. {
44. open local speakers
45. while die signal not given
46. {
47. wait semaphore mPlay
48. wait semaphore mRecv
49. get data from multicastBuffer
50. signal semaphore mRecv
51. signal semaphore mBuffer
52. play data on speakers
53. }
54. cleanup
55. // don't kill, this is the parent thread
56. }
57. display songlist function
58. {
59. for each song in the list of songs available from the server
60. {
61. add the song and artist name to a clickable list item on GUI
62. update GUI
63. }
64. }
65. request song stream function
66. {
67. get the song name from the GUI item clicked
68. generate a control message for song request with the message data set to
    1. the song name
69. open the UDP channel for receiving data
70. send the control message on the TCP control channel to the server
71. call the unregister from multicast function
72. call the receive song information function
73. }
74. receive song information function
75. {
76. get the song name, artist, and album
77. set the GUI track player with the data
78. call the play single song function
79. }
80. play single song function
81. {
82. while the song has not finished
83. receive song bytes from the UDP channel and store them in the buffer
84. play the buffer
85. close the UDP channel with the server
86. register for multicast
87. }
88. request song download function
89. {
90. get the song name from the GUI item selected
91. generate a control message for SAVE\_SONG with the message data set to
    1. the song name
92. create file transfer thread
93. send the control message on the TCP control channel to the server
94. }
95. File transfer thread
96. {
97. while true
98. Create a socket and listen for TCP connections
99. when a new connection arrives, validate
100. if valid, create new connection
101. begin receiving file data until all data is received
102. }
103. // Should be started as a thread
104. Start Voice Chat function
105. {
106. Start sending thread
107. Start recieving thread
108. Start recording thread
109. Call sound playing function
110. Make sure other threads are dead
111. Kill thread
112. }
113. /\*
114. The following functions require these initial conditions:
115. binary semaphore OutRecord = 1
116. binary semaphore OutSend = 0
117. semaphore OutBuffer = size of voiceOutBuffer
118. binary semaphore InRecv = 1
119. binary semaphore InPlay = 0
120. semaphore InBuffer = size of voiceInBuffer
121. \*/
122. // All the following functions would be in a separate thread
123. Voice Record function
124. {
125. open local microphone
126. while die signal not given
127. {
128. record data
129. wait semaphore OutBuffer
130. wait semaphore OutRecord
131. place data into voiceOutBuffer
132. signal semaphore OutRecord
133. signal semaphore OutSend
134. }
135. cleanup
136. kill thread
137. }
138. Voice Send function
139. {
140. open udp sending socket
141. while die signal not given
142. {
143. wait semaphore OutSend
144. wait semaphore OutRecord
145. get data from voiceOutBuffer
146. signal semaphore OutRecord
147. signal semaphore OutBuffer
148. send data on socket
149. }
150. cleanup
151. kill thread
152. }
153. Voice Recv function
154. {
155. Open udp listening socket
156. while die signal not given
157. {
158. wait for data
159. wait semaphore InBuffer
160. wait semaphore InRecv
161. place data into voiceInBuffer
162. signal semaphore InRecv
163. signal semaphore InPlay
164. }
165. cleanup
166. kill thread
167. }
168. Voice Play function
169. {
170. open local speakers
171. while die signal not given
172. {
173. wait semaphore InPlay
174. wait semaphore InRecv
175. get data from voiceInBuffer
176. signal semaphore InRecv
177. signal semaphore InBuffer
178. play data on speakers
179. }
180. cleanup
181. // don't kill, this is the parent thread
182. }