FEBRUARY 16, 2015

DATA COMMUNICATIONS (COMP 4985)

COMM AUDIO

RHEA LAUZON // JEFF BAYNTUN // MICHAEL CHIMICK // JULIAN BRANDRICK
4O

Contents

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)

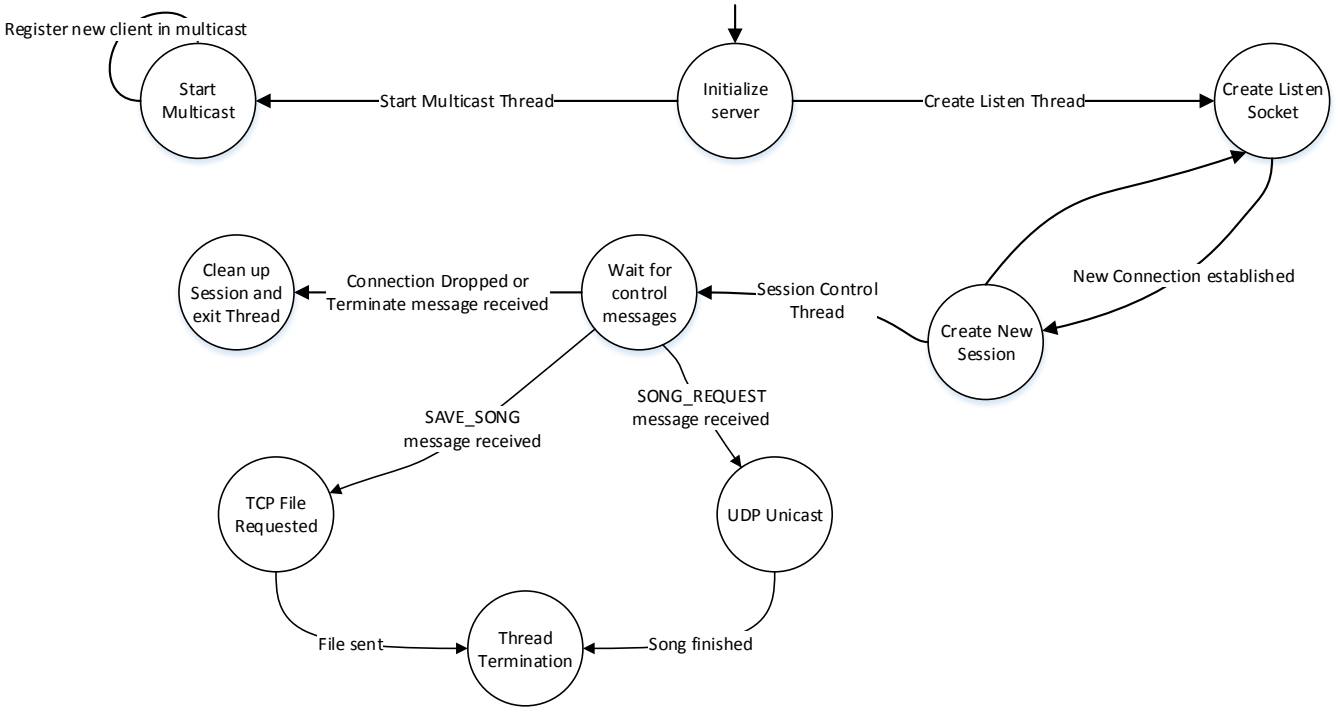
- Separate data channel (UDP) from data-channel

Sending File

- Via secondary TCP channel

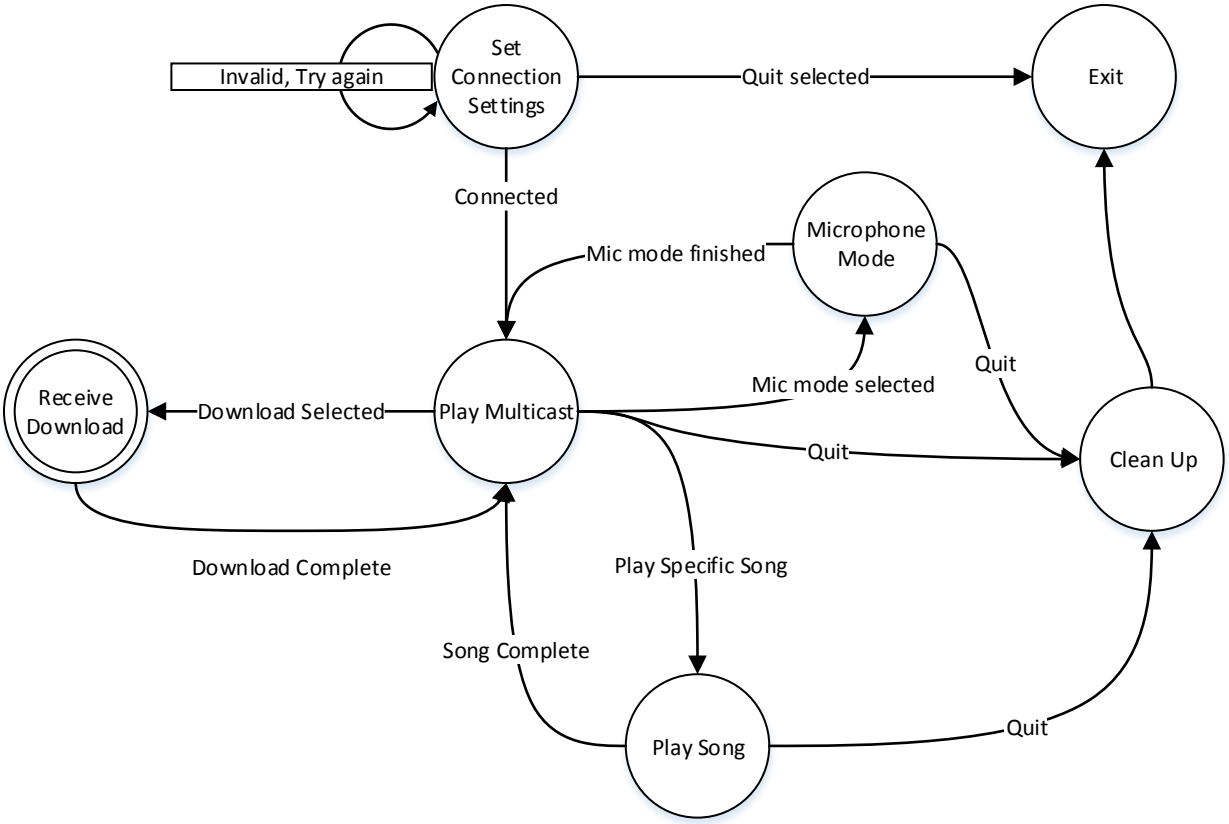
State Flow Diagrams

Server Side (High Level)

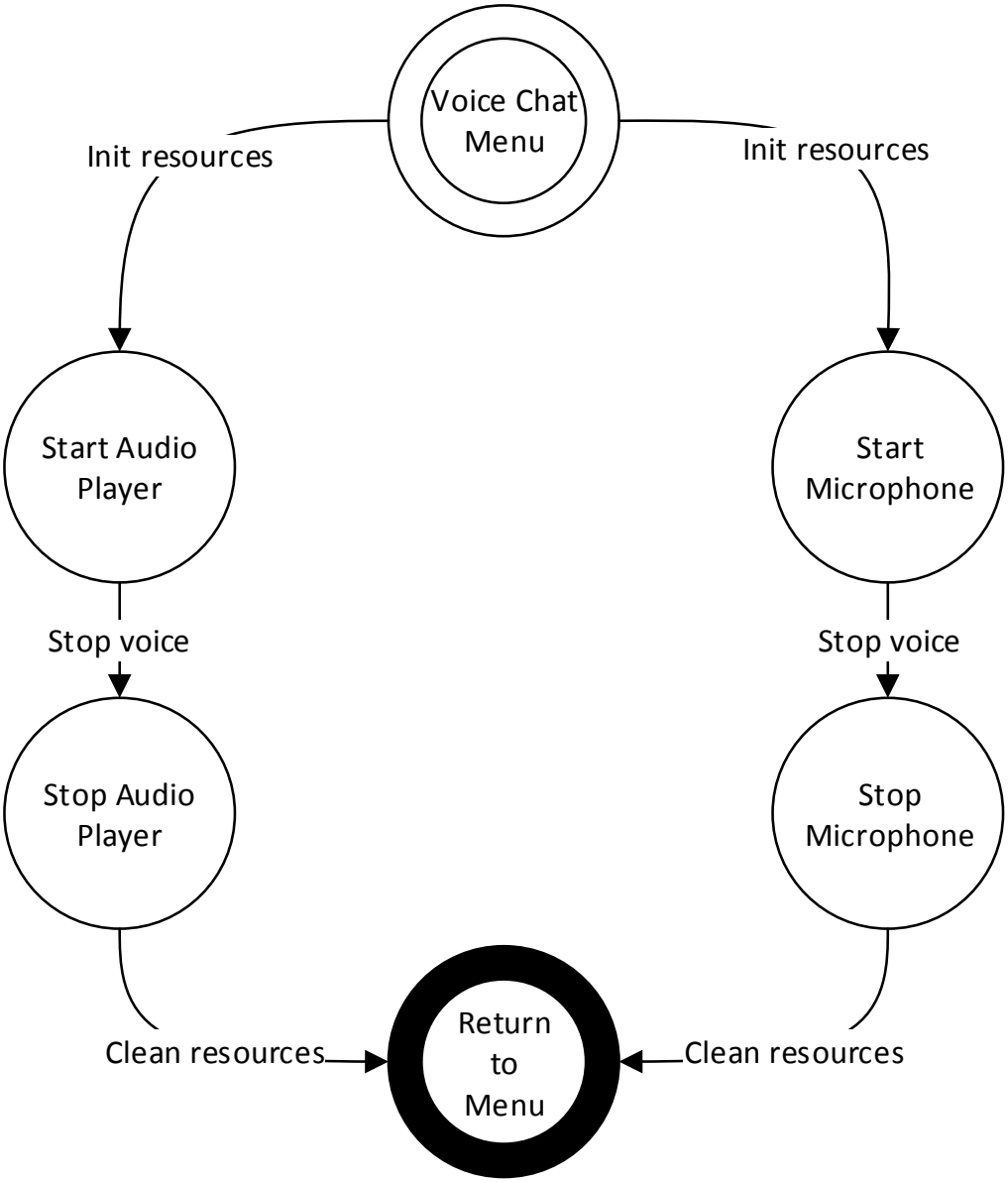


Client Side

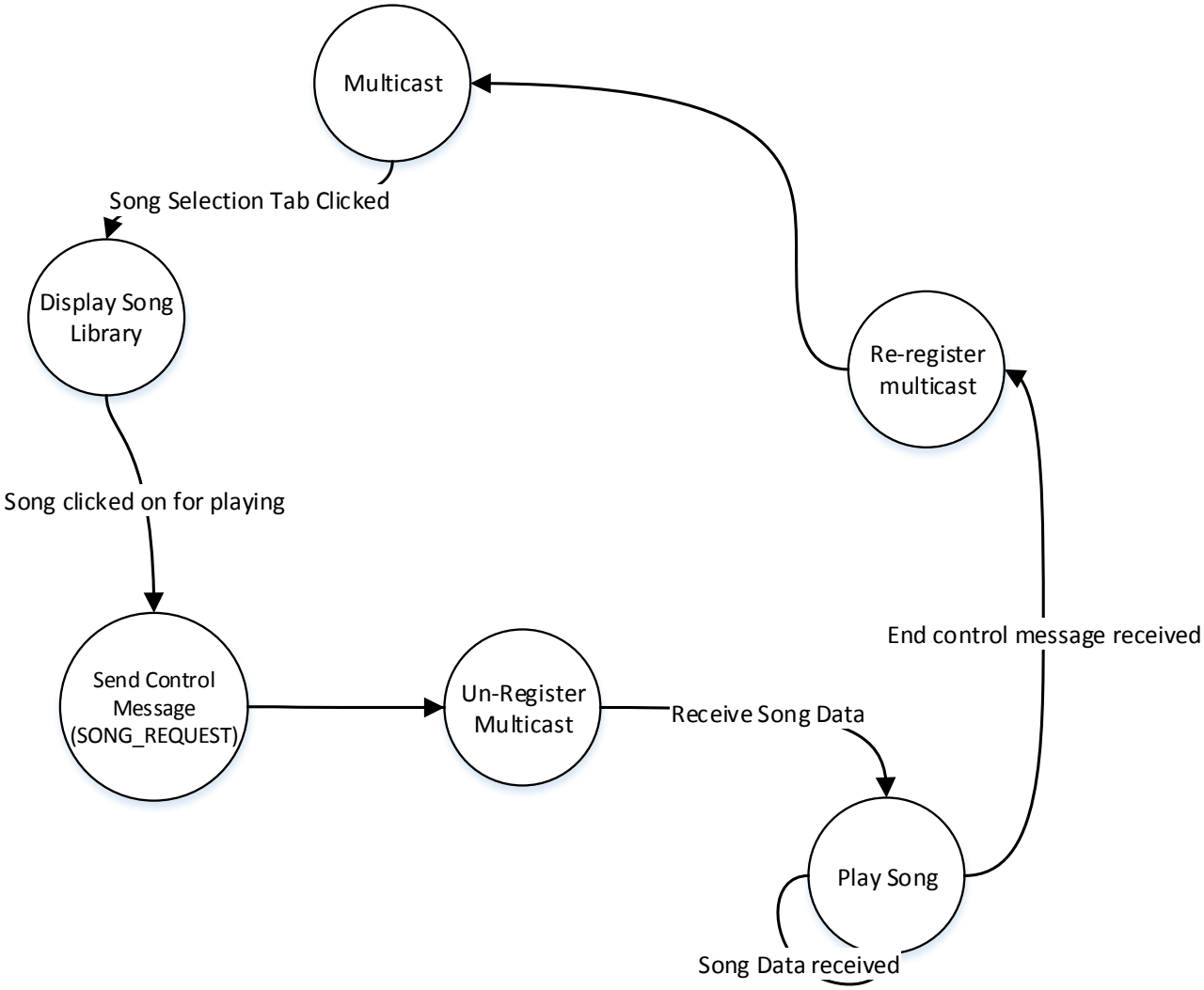
High Level



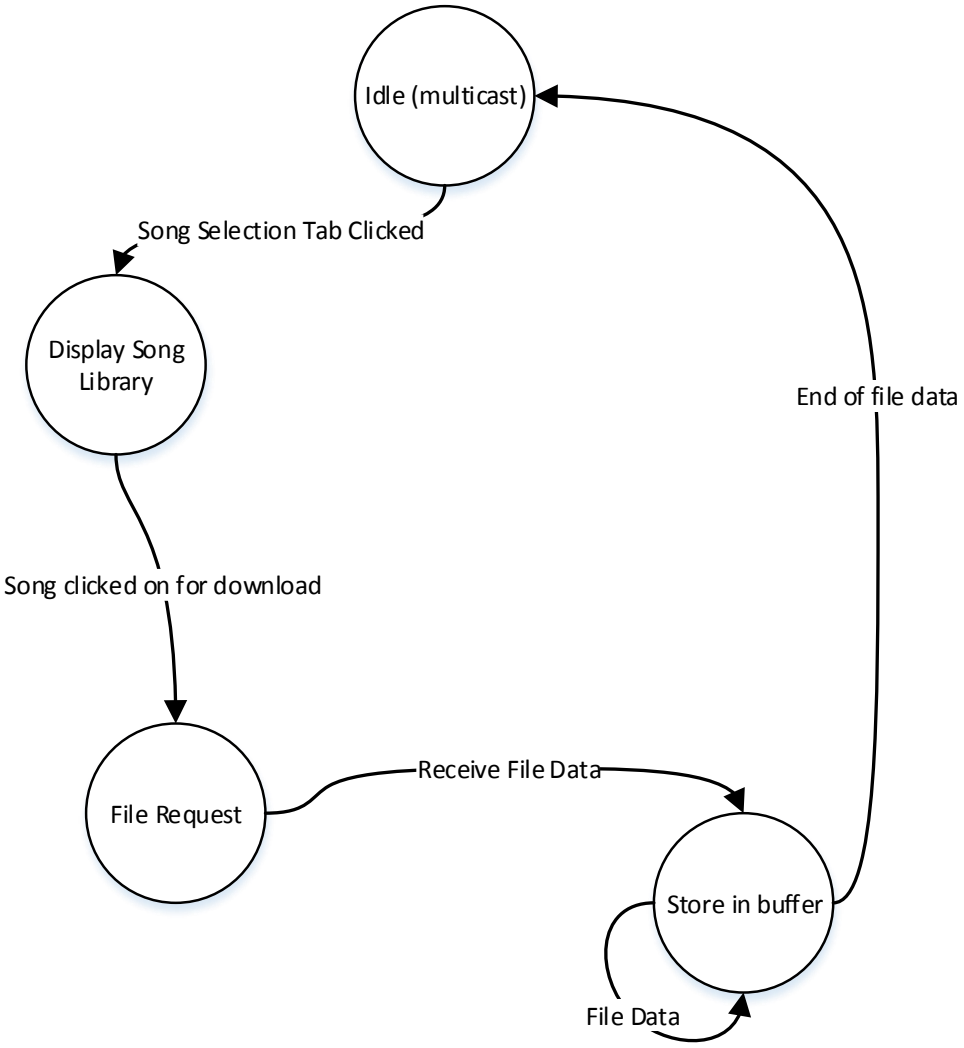
Client (Voice Chat)



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

Server

```
1. Initialize Server function
2. {
3.      Load list of song titles and locations
4.      Initialize Multicast thread // details below
5.
6.     create Accept thread
7. }

1. CreateListen function
2. {
3.      create SESSIONS semaphore, initialize to 1
4.      create a socket and Listen on SERVER TCP LISTEN PORT
5.
6.      while true
7.          when a new connection arrives, create new Session to handle connection
8.     allocate new session structure
9. }

1. CreateSession function
2. {
3.      wait on SESSIONS semaphore
4.      add allocated session structure to sessionMap
5.      signal on SESSIONS semaphore
6.
7.      create control thread
8. }

1. ControlThread function
2. {
3.      establish session from input parameter
4.      create finishedTransfer semaphore, initialize to 0
5.      create userChanged semaphore, initialize to 0
6.
7.      send song list
8.      send name/artist/album of currently playing song
9.      signal clients on their userChanged semaphore
10.
11.     make TCP Receive call with controlRoutine as call-back
12.     while true
13.         enter alertable state and wait for the finishedTransfer and userChanged
              semaphores as well as WSA Callbacks
14.         if userChanged signalled
15.             send new user list
16.         if finishedTransfer signaled //unicast/TCP song finished
17.             send END_SONG message
18.         if error occurs
19.             clean up session
20. }
```

```
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. }
```

```
1. Unicast Thread
2. {
3. create UDP socket
4. pass socket to waitingClients queue
5. start sending the song through to the client
6. when sending is finished, signal on finishedTransfer semaphore
7.
8. terminate Unicast thread
9. }
```

```
1. TCPFilefunction
2. {
3.     get session based on socket
4. initialize TCP socket
5. send file stream
6. when sending is finished, signal on finishedTransfer semaphore
7.
8. terminate TCPFile thread
9. }
```

```
1. CleanupSession function
2. {
3.     wait on SESSIONS semaphore
4.     close finishedTransfer semaphore
5.     close userChanged semaphore
6.     signal on SESSIONS semaphore
7.
8.     send new user list
9.     exit session control thread
10. }
```

```
1. Multicast Thread function
2. {
3. initialize multicast socket
4.      create multicast session
5.      set time to live
6.      disable loopback
7.
8.      initialize multicast song list
9.      start playing song over multicast socket
10.}
```

Client

```
1. Idle function
2. {
3.     Initialize GUI
4.     Call join multicast function
5. }

1. StartMulticast
2. {
3.     Start the multicast thread
4. }

1. EndMulticast
2. {
3.     Drop the multicast membership
4.
5.     End the multicast thread
6. }

1. JoinMulticast
2. {
3.     Open the socket
4.
5.     Set any socket options
6.
7.     Bind the socket
8.
9.     Add the socket to the multicast session
10.
11.    Start the sound output thread - playMulticastSong
12.
13.    Call the recieve multicast function
14.}

1. receiveMulticastData
2. {
3.     Forever loop
4.
5.         Receive data from the server
6.
7.         Place the data into the circular buffer
8. }

1. MultiWaveCallback // called when a buffer has finished playing
2. {
3.     If the buffer is done playing
4.         add the buffer to the end of the output queue
5. }

1. updateVolume // takes a number parameter
2. {
3.     Set the volume to the new number
4. }
```

```
1. playMulticastSong
2. {
3.     Initialize the waveformat settings
4.
5.     Open the local output device
6.
7.     Add the buffers to the output queue
8.
9.     Wait for some of the buffer to fill
10.
11.    Start the sound output
12.}

1. getCircularBuffer
2. {
3.     return the multicast buffer
4. }

1. closeAudio
2. {
3.     Close the output device
4.
5.     Cleanup the buffers
6. }

1. display songlist function
2. {
3.     for each song in the list of songs available from the server
4.     {
5.         add the song and artist name to a clickable list item on GUI
6.         update GUI
7.     }
8. }

1. request song stream function
2. {
3.     get the song name from the GUI item clicked
4.     generate a control message for song request with the message data set to
   4.1.      the song name
5.
6.     open the UDP channel for receiving data
7.     send the control message on the TCP control channel to the server
8.     call the unregister from multicast function
9.     call the receive song information function
10.}

1. receive song information function
2. {
3.     get the song name, artist, and album
4.     set the GUI track player with the data
5.     call the play single song function
6. }
```

```
1. play single song function
2. {
3.     while the song has not finished
4.         receive song bytes from the UDP channel and store them in the buffer
5.         play the buffer
6.
7.     close the UDP channel with the server
8.     register for multicast
9. }
```

```
1. request song download function
2. {
3.     get the song name from the GUI item selected
4.     generate a control message for SAVE_SONG with the message data set to
   4.1.      the song name
5.     create file transfer thread
6.     send the control message on the TCP control channel to the server
7. }
```

```
1. File transfer thread
2. {
3.     while true
4.     Create a socket and listen for TCP connections
5.     when a new connection arrives, validate
6.     if valid, create new connection
7.
8.     begin receiving file data until all data is received
9. }
```

```
1. VoiceChat function
2. {
3.     start device VoicePlay
4.     create dialog box for voice chat
5. }
```

```
1. class Player
2. {
3.     InitPlayer function
4.     {
5.         setup audio player device
6.         create UDP socket
7.         setup callback for PlayData
8.     }
9.
10.    StartVoicePlay function
11.    {
12.        start device playback
13.        bind UDP socket to VOICE_CHAT port
14.    }
15.
16.    StopVoicePlay function
17.    {
18.        stop audio player playback
19.    }
20.
21.    PlayData function
22.    {
23.        while socket has receivable data
24.            receive data from socket
25.            write data to audio device
26.    }
27.}
```

```
1. class Microphone
2. {
3.     InitMicrophone function
4.     {
5.         setup microphone device
6.         create UDP socket
7.     }
8.
9.     StartVoice function
10.    {
11.        setup socket for receiving from client
12.        write received data from the socket to the microphone
13.    }
14.
15.    StopVoice function
16.    {
17.        stop the microphone
18.    }
19.}
```

```
1. Voice Recv function
2. {
3.      Open udp listening socket
4.
5.      while die signal not given
6.     {
7.          wait for data
8.
9.          wait semaphore InBuffer
10.         wait semaphore InRecv
11.
12.         place data into voiceInBuffer
13.
14.         signal semaphore InRecv
15.         signal semaphore InPlay
16.     }
17.
18.     cleanup
19.     kill thread
20.}
```

```
1. Voice Play function
2. {
3.      open local speakers
4.
5.      while die signal not given
6.      {
7.          wait semaphore InPlay
8.          wait semaphore InRecv
9.
10.         get data from voiceInBuffer
11.
12.         signal semaphore InRecv
13.         signal semaphore InBuffer
14.
15.         play data on speakers
16.     }
17.
18.     cleanup
19.     // don't kill, this is the parent thread
20.}
```