Design Document

Data Communications: Assignment 3, Comm Audio

Manuel Gonzales, A00866174, 4O
Georgi Hristov, A00795026, 4O
Calvin Rempel, A00871348, 4O
Eric Tsang, A00841554, 4O

January 10, 2015

Table of Contents

Requirements & Technical Overview 4
Server 4
Client 4
State Transition Diagrams 5
Server 5
Client 6
State Pseudocode 7
Server 7
Initialize 7
Idle 7
Running 7
Stopping 8
Client 8
Configuration 8
Connecting 8
Connected 8
Asynchronous Transmitting 9
Asynchronous Playing Music 9
Asynchronous Plying Voice 9
Data Flow Diagrams 10
Server 10
Client 11
Flow Chart Diagrams 13
Server 13
Session Manager 13
Session 14
Upload 15
Stream 16
Client 17
Control 17
Receive 18
Transmit 19

Music Buffering 19
Voice Buffering 20
Record 21
Music Reader 22
Output 23
Thread Pseudocode 24
Server 24
Session 24
Upload 25
Stream 25
Client 26
Control 26
Transmit 27
Receive 28
Music Buffering 28
Voice Buffering 29
Music Reader 30
Record 30
Output 31
Class Pseudocode 32
Jitter Buffer 32
Insert 32
Remove 32
Circular Buffer 33
Enqueue 33
Dequeue 33
Message Queue 34
Enqueue 34
Dequeue 34

Requirements & Technical Overview

There are two applications involved: the server, and the client.

Server

The server application:

- connects the clients with one another, so they can communicate to each other without involvement from the server
- updates clients about new clients, or leaving clients
- sends song data to clients that requested a download
- streams music data to all the clients using a multicast address

Client

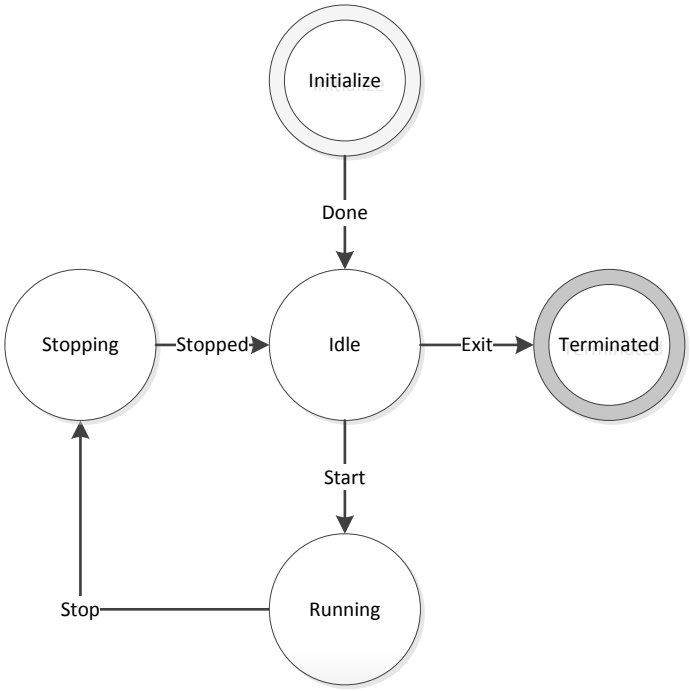
The client application:

- receives the streamed music, and save it to a temporary file, so its user can seek, pause, and play the stream
- transmits voice to all other clients using the same multicast address, or by sending voice data to individual clients
- received voice data from other clients, and play them simultaneously, discerning them from each other by checking the data's source address
- can request to download files from the server

State Transition Diagrams

The state transition diagrams in this section describe the states of the server and client applications as they appear to the user.

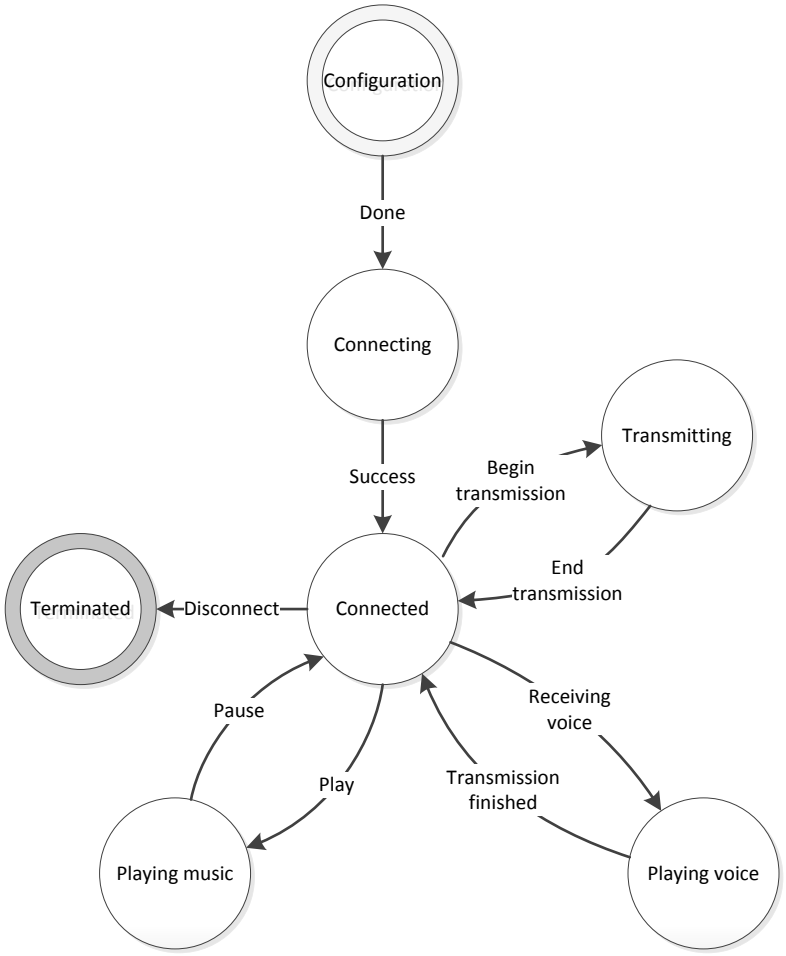
Server



The diagram above illustrates the states of the server application as it appears to the user:

- **Initialize**; the program reads its previous configuration from a file, sets up the GUI, and other one-time tasks needed to start the program.
- **Idle**; the program is waiting for further commands from the user. While in this state, the user can change the port number that will be used for the server's listening port.
- **Running**; the application has open a listening port, and is waiting for, or currently serving clients.
- **Stopping**; a stop command has been issued from the user, so the application is currently terminating all the threads used to server clients, and freeing their resources.
- **Terminated**; the program is no longer running.

Client



The above diagram illustrates the states of the client application as it appears to the user:

- **Configuration**; the client application gathers connectivity information from the user that's needed in order to connect to the server application by prompting the user for the server's address, and port number with dialog boxes.
- **Connecting**; the application is currently connecting with the server, and then exchanging basic information needed by the application to run, like the multicast address for voice transmissions, or music.
- **Connected**; the client has established a connection with the server in this state, and is currently receiving any streamed audio or voice. While the application is in this state, it can transition to any of the Playing Voice, Playing Music, or Transmitting states, simultaneously, for example, the program can be in both the Playing Music and Transmitting states at the same time.
- **Transmitting**; the application is transmitting voice packets on the voice multicast address.
- **Playing voice**; the application is playing voice received packets immediately.
- **Playing music**; the application is playing music from a temporary file created to save the received streamed music.
- **Terminated**; the application has ended.

State Pseudocode

Server

Initialize

```
1  Read last used port from config file
2  Create GUI
3  Go To State: IDLE
```

Idle

```
1  IF user connects:
2      Create a listening socket
3      Attempt to bind socket to stored port
4
5      WHILE the socket is not bound to a port:
6          Prompt user for a different port
7
8      Go To State: RUNNING
9
10 IF user exits:
11     Cleanup GUI
12     Go To State: TERMINATED
```

Running

```
1  Listen for data on all sockets
2
3  IF a new connection is made:
4      Store the new socket
5      Send TCP message to socket with initialization data
6      Send TCP message to all clients informing of new client
7
8  IF a client request a file download:
9      Open requested file
10     Write file to client socket
11
12 IF a client requests song change:
13     Close current song (if exists)
14     Clear buffers
15     Open requested song file
16     Send TCP message to all clients informing of song change
17     Decode song data
18
19     Write song data on multicast UDP socket
20
21 IF a client disconnects:
22     Remove stored socket
23
24 IF user stops server:
```

```
25      Go To State: STOPPING
```

Stopping

```
1  Close all open files
2  Close all sockets
3  Clear Buffers
4  Go To State: IDLE
```

Client

Configuration

```
1  Create/Show "configuration" GUI
2      Prompt for Port and Host
3
4  Go To State: CONNECTING
```

Connecting

```
1  Create TCP socket / Connect To Server
2  Create UDP socket / Bind to Port
3
4  IF success
5      Read initialization data from TCP socket
6      Go To State: CONNECTED
7  IF failure
8      Go To State: CONFIGURATION
```

Connected

```
1  IF receive voice data (on multicast socket)
2      IF client IP in datagram matches current IP
3          Set voice jitter buffer as active buffer
4          Disable microphone input button
5          Put voice data in voice jitter buffer
6          Set 1s timer
7              IF timer elapses before more voice data received
8                  Enable microphone input button
9                  Set music jitter buffer as active buffer
10             ELSE
11                 Reset timer
12
13 ELSE IF user holding microphone input button
14     Disable playback
15
16 IF receive music data (on multicast socket)
17     Write data to music jitter buffer
18     Write music data to temp music file
19
20 IF receive music data (on TCP socket)
```

```
21      Write music data to temp music file
22
23 IF receive music file data (on TCP socket)
24      Write data to open file
25
26 IF user requests song download
27      Create new file with name of song
28      Write request to TCP socket
```

Asynchronous Transmitting

```
1  Read data from microphone
2  Write data to UDP multicast socket including client IP
3
4  Set 1s timer
5           IF timer elapses before more voice data received
6               Set music jitter buffer as active buffer
7           ELSE
8               Reset timer
```

Asynchronous Playing Music

```
1  WHILE connected
2       Read music data from temp file starting from seek point
   into music buffer
3       IF data exists and music buffer is active buffer
4           Playback music buffer
```

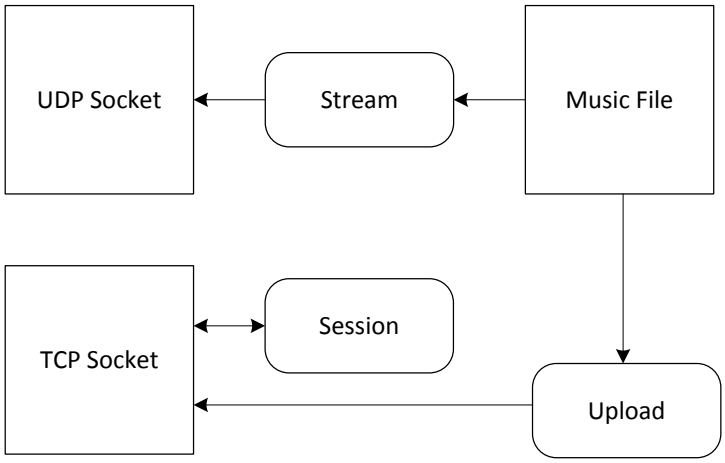
Asynchronous Plying Voice

```
1  WHILE connected
2       Read data from voice buffer
3       IF data exists and voice buffer is active buffer
4           Playback voice buffer
```

Data Flow Diagrams

The data flow diagrams in this section illustrates the how the data flows within the server and client applications between the sockets, files, buffers, and threads.

Server



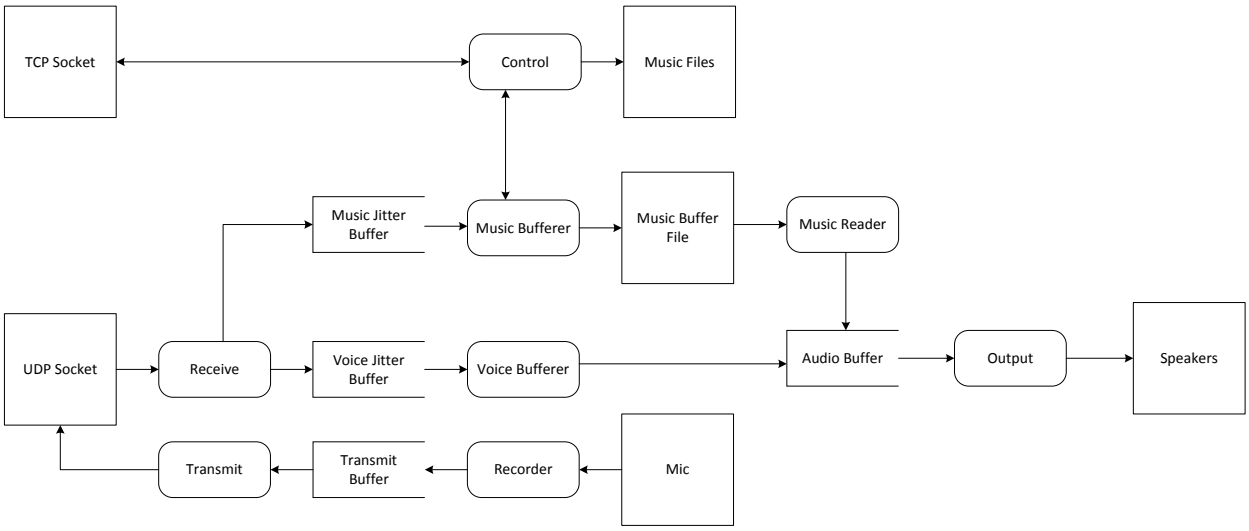
Above is a data flow diagram illustrating how data flows within the server application. The kind of data that circulates in the application includes:

- **Control**; control data includes a few things. some examples are information indicating to clients that a new song will now be streamed, or a new client has connected, or an existing client has disconnected. This data is sent through the TCP socket.
- **Streaming music**; song data that is currently being multicasted across the network to the clients. This data is sent through the UDP socket.
- **Music data**; when clients request to download music, the music data is sent to the requesting client through the TCP socket.

As illustrated, there are three processes in the server application:

- **Stream**; this process reads from the currently selected music file, and multicasts it on the network
- **Session**; this process interfaces directly with the client, one is created for each client. This process is used to exchange control data with the client
- **Upload**; this process is spawned whenever a session's client requests to download a song file. It reads from the requested song file, and sends it to the requesting client

Client



Above is a data flow diagram illustrating how data flows within the client application. As shown, there are various processes in the client application:

- **Control**; this process exchanges data between the TCP socket, music files, and the music buffering process. If the received data is music data, it will be written to a file, or forwarded to the music buffering process. Otherwise, it is control information, like requesting to download a file, or requesting to change streams. This process exists while the program is in the connected state.
- **Receive**; this process reads from the UDP socket, and writes the read data to the corresponding jitter buffer depending on the type of data that is read from the socket. This process exists while the program is in the connected state.
- **Transmit**; this process reads from the transmit buffer, and sends it out the UDP socket to a user specified address. This process exists while the program is in the connected state.
- **Music buffering**; this process reads from the music jitter buffer, and writes it the corresponding music buffer file. This process exists while the program is in the connected state.
- **Voice buffering**; this process reads from the voice jitter buffer, and writes the data to the audio buffer. This process exists when the application is in the connected state.
- **Record**; this process reads from the mic, and writes the read data into the transmit buffer. This process only exists while the application is in the transmitting state.
- **Music Reader**; this process reads from the temp files created to hold the streamed music data, and writes the data to the audio buffers. This process only exists while the application is in the connected state.
- **Output**; this process reads from the audio buffer, and plays it out through the system's speakers. This process is created when the program reaches the connected state, and is terminated when the program leaves the connected state.

The kind of data that circulates in the application includes:

- **Control**; control information is exchanged between the TCP socket, and the control process. Such information includes download requests, stream change requests, and others.
- **Music data**; there are two kinds of music data that can be received:
    - **Retransmission of a dropped streamed music data packet**; the data is received through the TCP socket, read by the control process, passed to music buffering process, and written to the appropriate place in the music buffer file.
    - **Music being downloaded**; the data is received through the TCP socket, read by the control process, and then appended to a song file.
- **Streamed voice data**; this data is sent from another client, and needs to be played immediately. The path it takes in the client application is as follows:
    1. Received through the UDP socket
    2. Read by the receive process
    3. Placed in the voice jitter buffer to account for jitter from the network
    4. Passed to the audio buffer, to be read by the output process
    5. Read by the output process and played as audio out the system's speakers
- **Streamed music data**; this data is sent from the server, and is saved in a temporary file, so the user may pause, seek, and resume the stream whenever they like. The path it takes is as follows:
    1. Received through the UDP socket
    2. Read by the receive process
    3. Placed into the music jitter buffer, to account for any jitter from the network
    4. Read by the music buffering process
    5. Written to the music buffer file where it may be read by the music reader process
    6. Read by the music reader process
    7. Placed into the audio buffer
    8. Read from the audio buffer by the output process, and played on the system's speakers
- **Voice data**; this data is created by the user of this client application. It is created by the mic, read by the recorder process, and placed into the transmit buffer, where it is read by the transmit process, which sends it out the UDP socket to a multicast address, or to another client application.

Flow Chart Diagrams

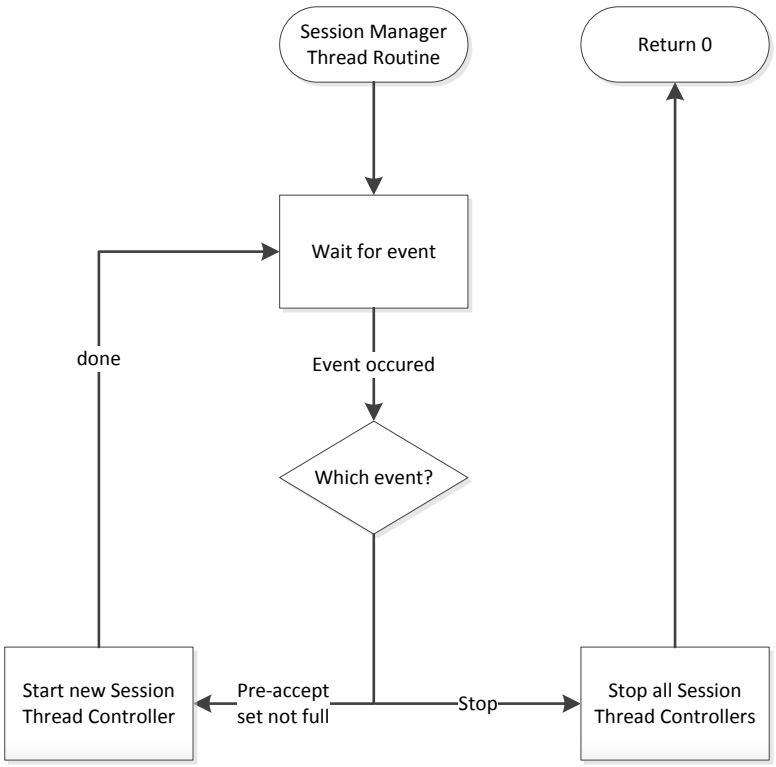
The following flow chart diagrams in this section show how each of the threads identified in the data flow diagrams work from a high level. They show how resources are allocated, used, and deallocated for each thread.

In general, as the flow diagram approaches the bottom of the page, it is allocating more resources, and doing work. As the flow diagram ascends the page, it is releasing resources, and terminating.

Server

This section contains flow diagrams illustrating the flow of execution of the processes in the server application.

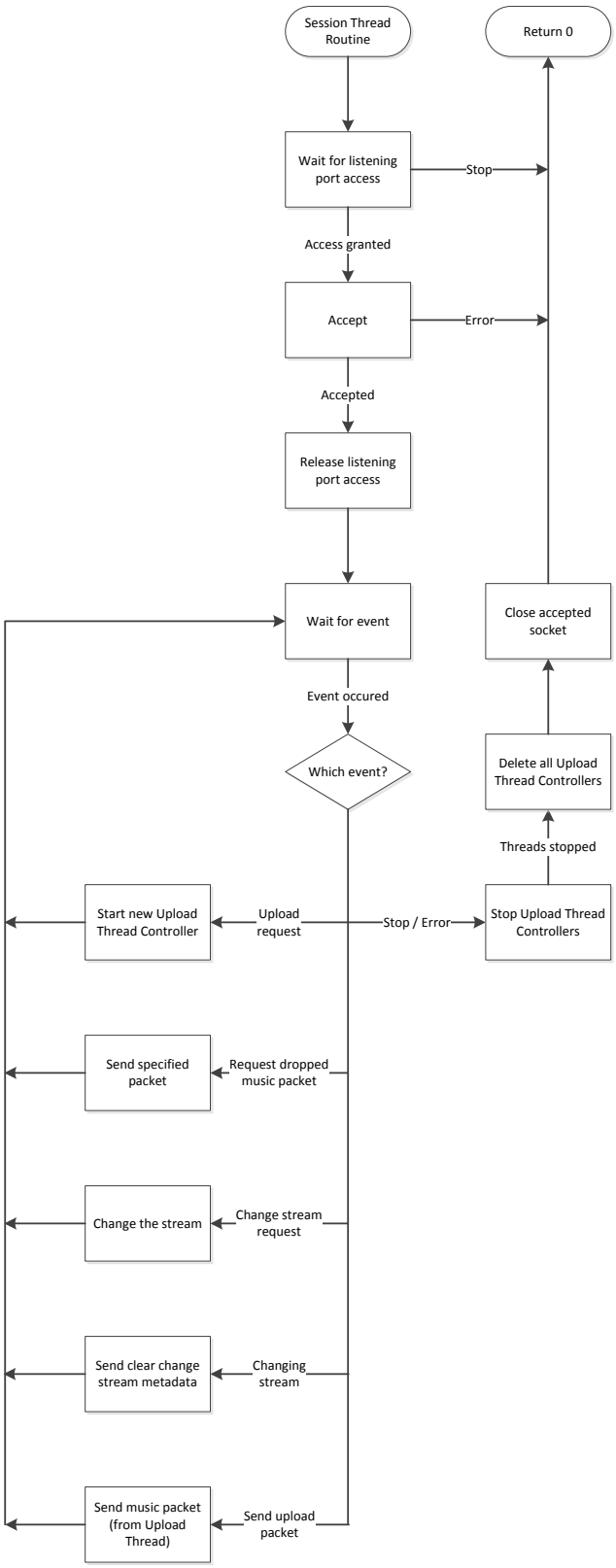
Session Manager



The session manager thread is responsible for making sure that there are enough session threads listening for connections on the server's listening socket. It does this by keeping the pre-accept set as full as possible. The pre-accept set is a set of session threads that are waiting for a connection. This thread is alive when the server is in the Running state.

The session manager thread can be signaled to stop. When this happens, it signals all the session threads that it created to stop as well. Once all session threads have stopped, this thread terminates as well.

Session

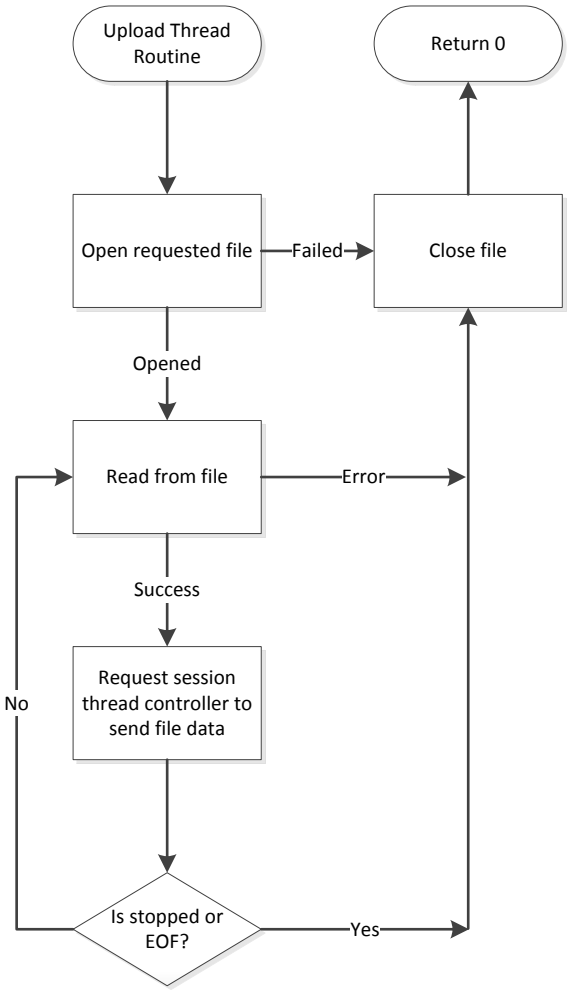


The session thread is used to interact with the client, and listen for its requests, as well as sending the client control information.

The thread starts by accepting a new connection from the server's listening socket, then moves on to servicing the client until the client disconnects, or the thread is commanded to stop.

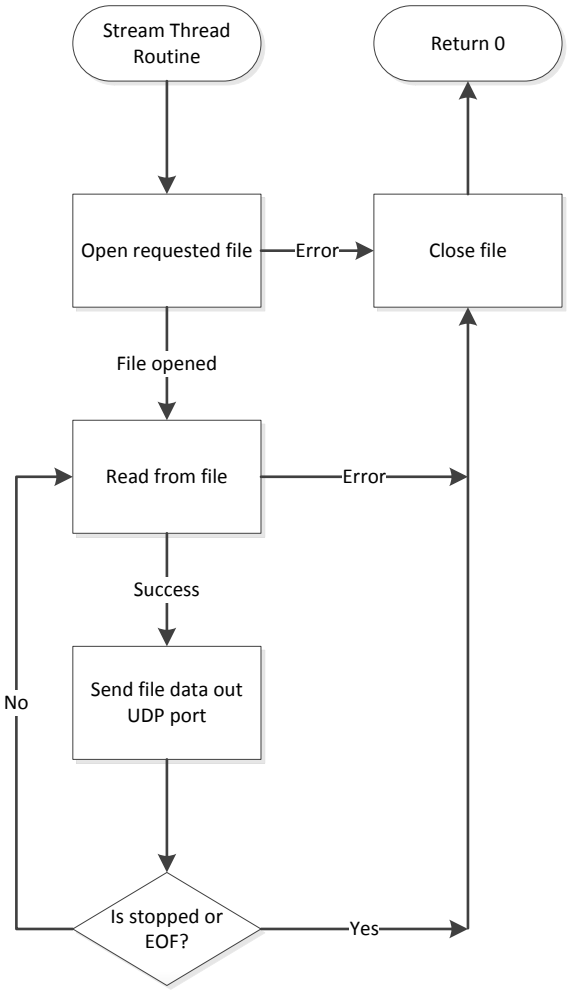
During the thread's operation, it may create upload threads to service the client's download requests. When the session thread is stopped, it stops all the upload threads it created before terminating itself.

Upload



The upload thread is used to service a client's request to download a music file. This thread reads from the specified file, and sends it to the client through the TCP socket until the upload is complete, or a stop command is issued to the thread.

Stream

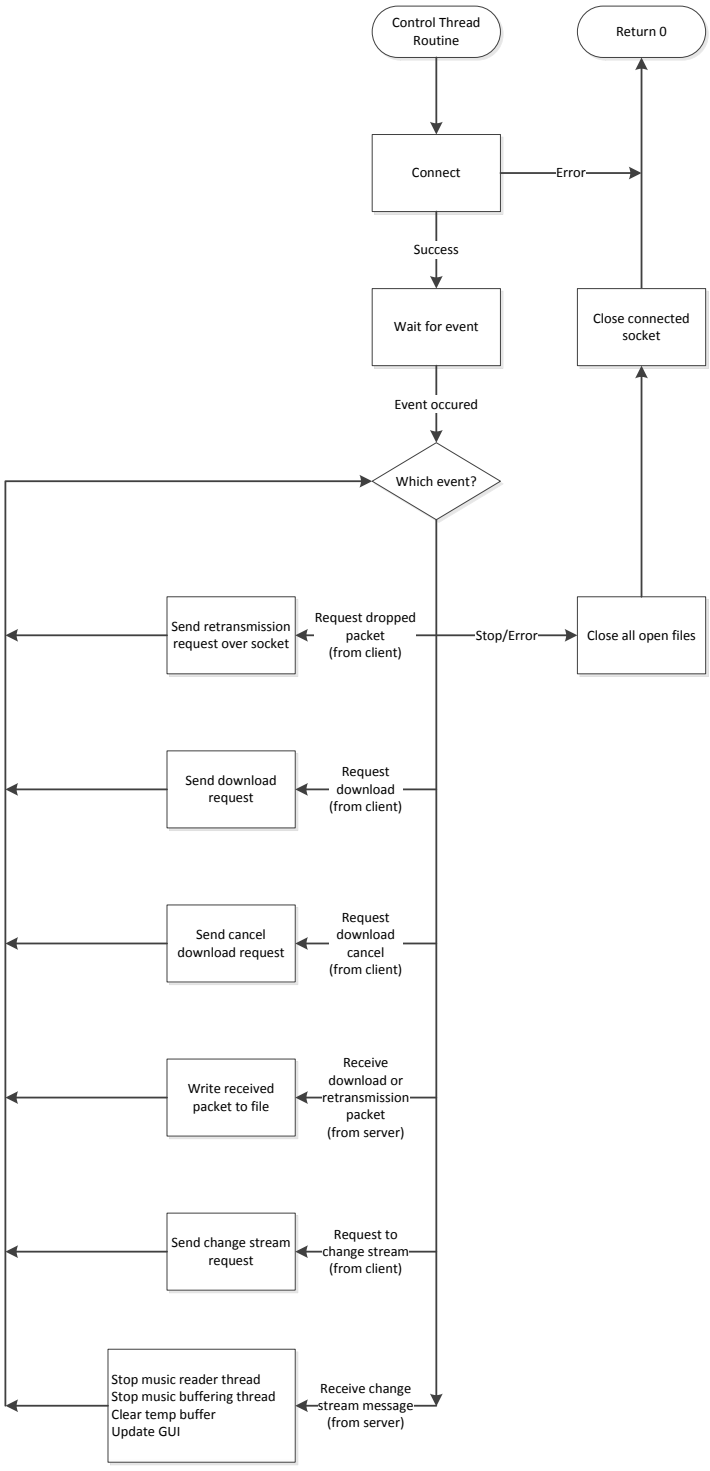


The stream thread is used to multicast a file on the network. There should only be one of these threads running in each server application at a time. This thread is alive when the server is in the Running state. It reads from a file, and multicasts it on the UDP socket until it finishes transmitting the whole file, or it is issued a stop command.

Client

This section contains flow diagrams illustrating the flow of execution of the processes in the client application.

Control



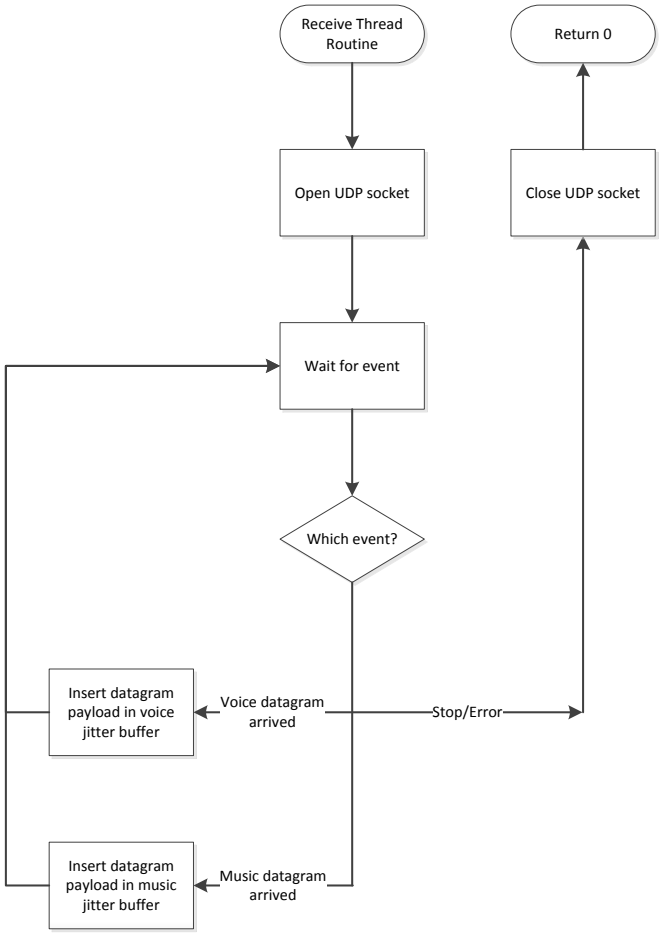
The control thread is used to receive messages from the control line, and communicate with the server. Not all messages that arrive through the TCP socket are control messages.

The control thread is used to process and control messages from the server, and send control messages to the server. These messages include download requests, retransmission requests, and stream change requests.

The control thread is also used to receive dropped music data packets, or music data received because of downloading music. When the control thread receives such packets, it writes it to the appropriate file, or passes it to the music buffering process to be written into another file.

When the control thread is shutting down, it closes all the files that it may have open from writing downloaded music data into music files.

Receive

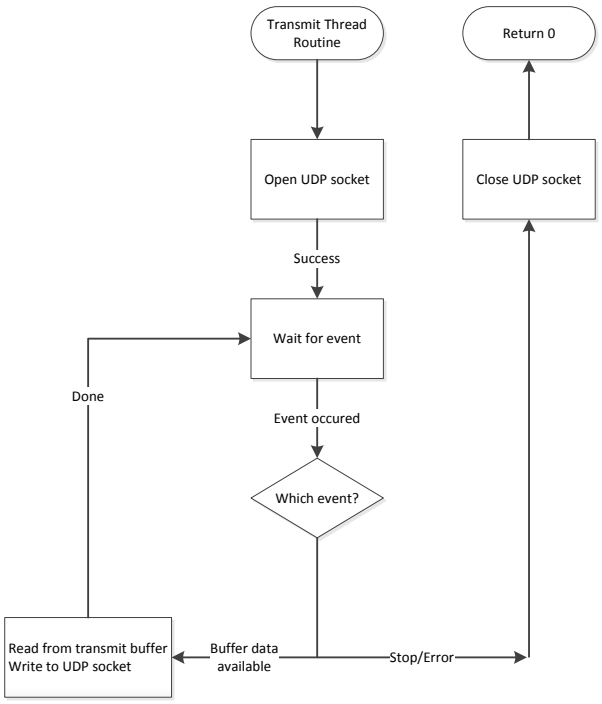


The receive process is used to listen to incoming datagrams on the client's UDP socket. Packets that can be encountered include voice datagrams from other clients, or music datagrams from the server. When such packets are received, they are inserted into their appropriate buffer; music datagrams are put into the music jitter buffer, and voice datagrams are put into the voice jitter buffer. Each voice datagram may be multiplexed into voice jitter buffers depending on their source IP.

Jitter buffers are necessary because the datagrams may not arrive in order. Jitter buffers allow the received datagrams to be put in the order they were sent, so when they are consumed and played by the output process, it is not garbled, and played out of order.

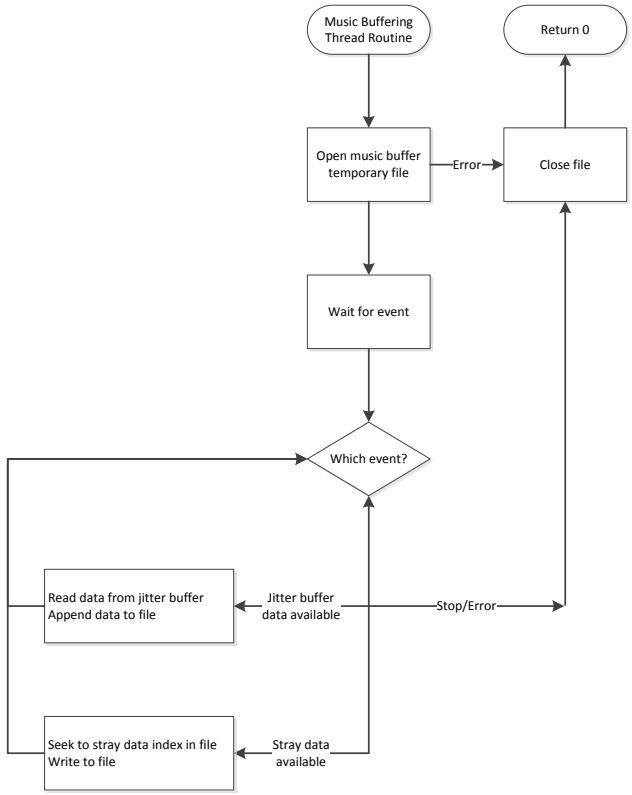
When the thread receives a stop signal, it closes the UDP socket, and terminates.

Transmit



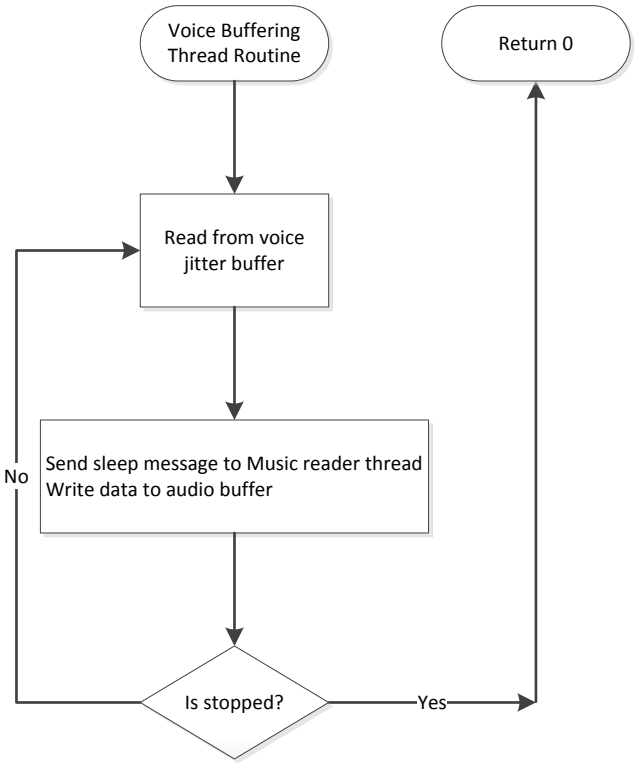
The transmit thread is a simple thread. It continuously reads data from the transmit buffer, and sends it to the user specified address through the application's UDP socket. This thread terminates when it receives the stop command.

Music Buffering



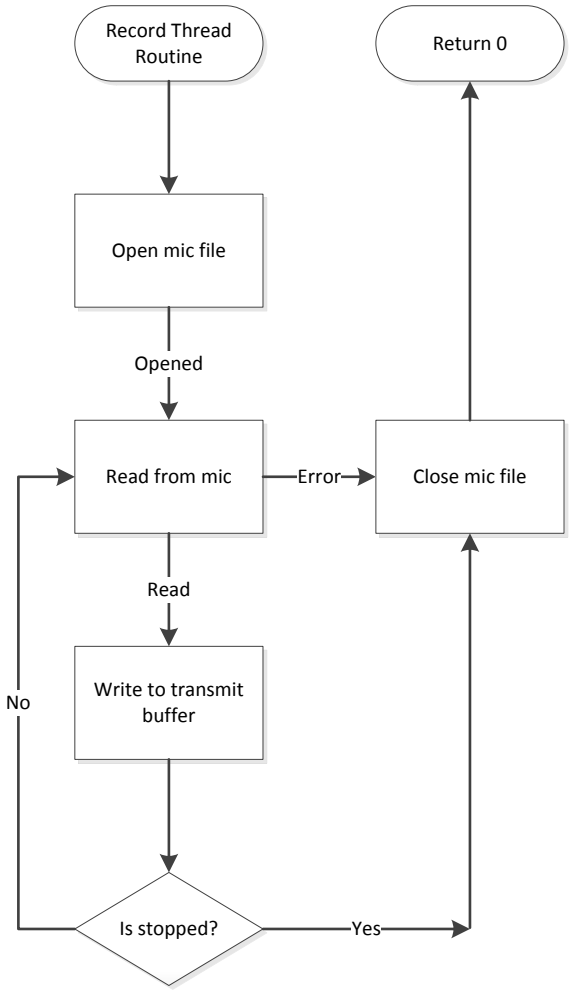
The music buffering thread is used to write data to the music temporary buffer files. It reads stream data from the music jitter buffer, and writes the data to the temporary buffer file. Every once in a while, it may receive a retransmitted packet from the control thread to be written to the temp music file as well. This thread will terminate when it receives the stop command, or it encounters an error.

Voice Buffering



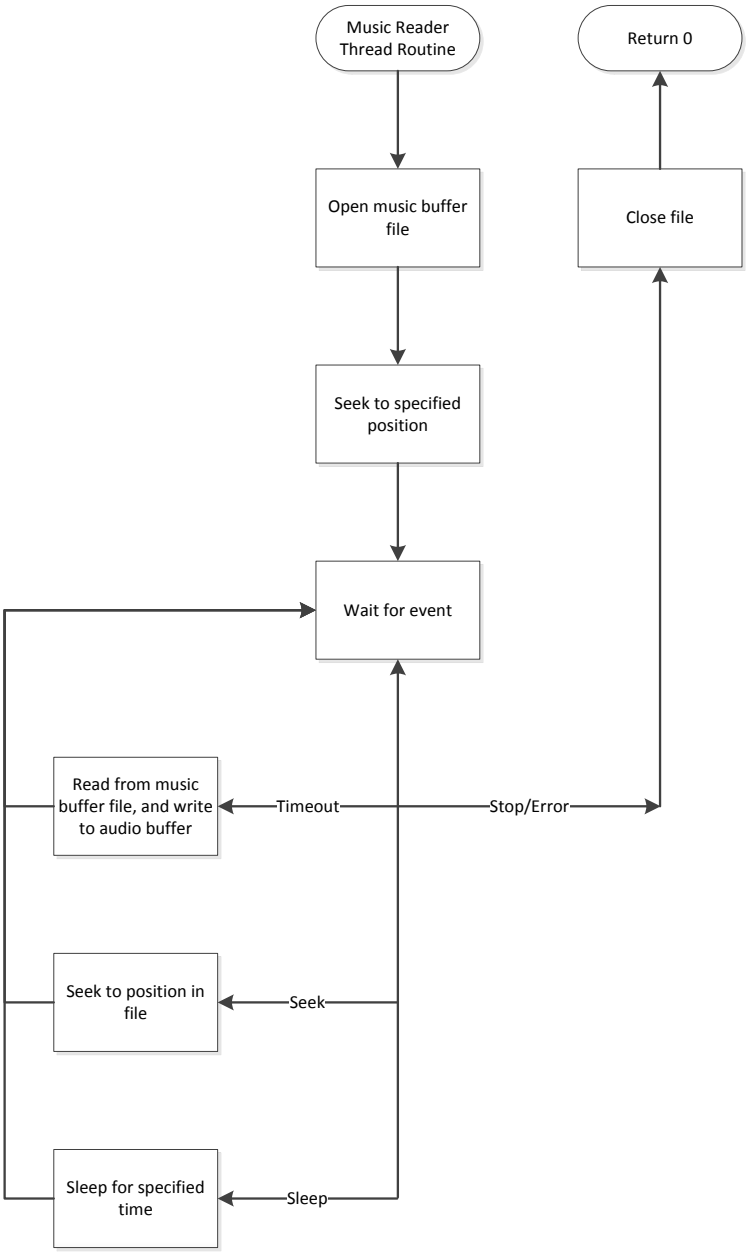
The voice buffering thread is simple. It reads the data from the voice jitter buffer, and writes the read data to the an audio buffer, which is used to play it. This thread continues to do this until it receives the stop event, which terminates the thread.

Record



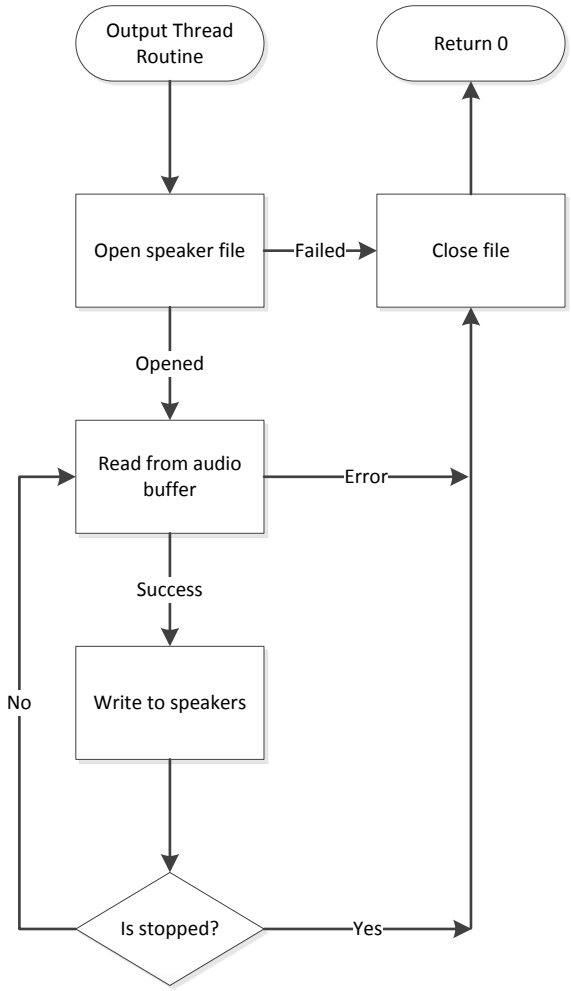
The record thread is simple, and only exists while the client application is in the transmitting state. This thread reads audio data from the system's mic, and places it into the transmit buffer. The thread stops once it receives the stop command, which terminates the thread.

Music Reader



The music reader thread is used to read the temporary files created to hold the streamed audio. It is mainly controlled by the GUI, and lets the user play, pause, and seek in the saved audio, and play from anywhere the user likes.

Output



The output thread routine reads the audio data from an audio buffer, does any decoding as necessary, and then plays it on the system's speakers. This continues until a stop command is issued to the thread.

Thread Pseudocode

Server

```
1  Start Listening for Connections
2  while
3  {
4       if(there is a new client)
5       {
6           Create a new Session.
7           Add session to session List.
8       }
9       if(stopped)
10      {
11          free resources
12      }
13 }
```

Session

Interfaces directly with the client, one is created for each client. This process is used to exchange control data with the client

```
1  Session(socket client)
2  {
3       Send List of Songs to the Client
4       Send Client the list of all the sessions (Clients
   Connected)
5
6       start Listening from the socket
7       while
8       {
9           if(stopped)
10          {
11              free resources
12          }
13
14          Check Client List.
15
16          if(there is an update)
17          {
18              Send update to every client
19          }
20
21          Read 'flag' from the socket.
22
23          if(flag == download request)
24          {
25              read from the socket
26              get the Song out of the message received
```

```
27              Call Upload(Song, client)
28          }
29          else if(flag == stream request)
30          {
31              read from the socket
32              get the Song out of the message received
33              Add Song to Song Queue.
34          }
35          else if(flag == disconnect flag)
36          {
37              Remove client from client list
38              close socket and connection
39              free resources and exit the process
40          }
41          else
42          {
43              Flag Error
44              Let the client know
45          }
46      }
47  }
```

Upload

Interfaces directly with the client, one is created for each client. This process is used to exchange control data with the client

```
1  Upload(string Song, Session client)
2  {
3
4       Open the File(Song)
5       Send File to client using the TCP Socket.
6       Close File
7
8  }
```

Stream

Reads from the currently selected music file, and multicasts it on the network

```
1  Stream()
2  {
3       While
4       {
5           get resources
6
7           if(there are Song requests)
8           {
9               Open the File(Song)
10
11              while
12              {
13                  Start Playing Song
```

```
14                  //Let Client know the song being played
15                  Multicast the audio using the UDP socket
16                  if(Song is complete)
17                  {
18                      Close File
19                      break
20                  }
21
22                  if(there is a new request)
23                  {
24                      Close File
25                      break
26              }
27          }
28      }
29
30      if(stopped)
31      {
32          free resources
33      }
34   }
35 }
```

Client

Client Connects to Server.

Server will send the Client List to the Client.

Control

Exchanges data between the TCP socket, music files, and the music buffering process. If the received data is music data, it will be written to a file, or forwarded to the music buffering process. Otherwise, it is control information, like requesting to download a file, or requesting to change streams. Running as long as the program is in the connected state.

```
1  Control()
2  {
3      acquire resources
4
5      while()
6      {
7          Read from the Socket()
8
9          Wait for events from the socket read AND User Requests
10         switch(event)
11         {
12             case there is an update to client list:
13                 {
14                     Modify Client List.
```

```
15                      break
16                  }
17              case a song file has been received:
18                  {
19                      read the data
20                      write the data to Message Queue
21
22                  }
23              case User wants to request a song to download:
24                  {
25                      Send flag
26                      Send song request trough TCP socket
27                  }
28              case User wants to request a song to play in
   stream)
29                  {
30                  Send flag
31                  Send song request trough TCP socket
32                  }
33              case stopped - disconnected
34                  {
35                      free resources
36                      break
37                  }
38              default:
39                      break;
40          }
41      }
42  }
```

Transmit

Reads from the transmit buffer, and sends it out the UDP socket to a user specified address. Running as long as the program is in the connected state.

```
1   Transmit()
2   {
3       acquire resources
4
5       while
6       {
7           if(transmit buffer has data)
8           {
9               get UDP socket - Client Information
10              Send the data trough the socket
11          }
12
13          if(stopped - disconnected)
14          {
15              free resources
16              break
17          }
```

```
18      }
19 }
```

Receive

Reads from the UDP socket, and writes the read data to the corresponding jitter buffer depending on the type of data that is read from the socket. Running as long as the program is in the connected state.

```
1  Receive()
2  {
3      acquire resources
4
5      while
6      {
7          Read from UDP Socket
8          Wait for an event from the socket read.
9          switch(data)
10         {
11             case voice:
12                         Write data to Voice Jitter Buffer
13                         break;
14             case music:
15
16                         Write data to Music Jitter Buffer
17                         break
18             default
19                         Prompt for unknown type received
20                         break;
21         }
22
23         if(stopped - disconnected)
24         {
25             free resources
26             break
27         }
28     }
29 }
```

Music Buffering

Reads from the music jitter buffer, and writes it the corresponding music buffer file. Running as long as the program is in the connected state.

```
1  Music buffering()
2  {
3      acquire resources
4
5      while
6      {
7          if(there is data in the music jitter buffer) // event
   received
```

```
8           {
9               Open File
10              read the data
11              if(data is music data)
12              {
13                  process data
14                  Write data to the music buffer
15              }
16              Close File
17          }
18
19          if(there is a music file in the message queue) //
   event received
20          {
21              Open File
22              read the data
23              if(data is music data)
24              {
25                  Write data to the music buffer
26              }
27              Close File
28
29          }
30
31          if(stopped - disconnected)
32          {
33              free resources
34              break
35          }
36      }
37  }
```

Voice Buffering

Reads from the voice jitter buffer, and writes the data to the audio buffer. Running as long as the application is in the connected state.

```
1   Voice buffering()
2   {
3       acquire resources
4
5       while
6       {
7           if(there is data in the voice jitter buffer)
8           {
9               read the data
10              if(data is voice data)
11              {
12                  process data
13                  Write data to the audio buffer
14              }
15
```

```
16              if(stopped - disconnected)
17              {
18                  free resources
19                  break
20              }
21          }
22      }
23  }
```

Music Reader

Reads from the temp files created to hold the streamed music data, and writes the data to the audio buffers. Running as long as the application is in the connected state.

```
1   Music Reader()
2   {
3       acquire resources
4
5       while
6       {
7           if(there is data in the music buffer)
8           {
9               get the file
10              read from it
11              Send data to Audio Buffer
12              close the file
13          }
14
15          if(stopped - disconnected)
16          {
17              free resources
18              break
19          }
20      }
21
22  }
```

Record

Reads from the mic, and writes the read data into the transmit buffer. Running as long as the application is in the transmitting state.

```
1   Record()
2   {
3       Start Recording audio from the default mic.
4
5       if(Stop record event received from GUI)
6       {
7           Write recording to transmit buffer.
8       }
9
10      if(stopped - disconnected)
```

```
11      {
12          free resources
13          break
14      }
15  }
```

Output

Reads from the audio buffer, and plays it out through the system's speakers. Running as long as the application is in the connected state.

```
1   Output()
2   {
3       acquire resources
4
5       while
6       {
7           if(there is data in the audio buffer)
8           {
9               Read data
10              Play data- through speakers
11          }
12
13          if(stopped - disconnected)
14          {
15              free resources
16              break
17          }
18      }
19  }
```

Class Pseudocode

The pseudocode in this section presents in a programming-language-agnostic way how threads, and various core classes for this assignment should be implemented.

Jitter Buffer

This section contains pseudo code for the jitter buffer class. The jitter buffer is used to buffer received network packets, and put them into the order that they were sent, since because of jitter, they may not have arrived in order. Then when the data is read from the jitter buffer, it is read in order.

Insert

Inserts the data at **src** into the jitter buffer at index **index**. If the passed data is too late (has already been consumed by the consumer), then the function returns right away, with an error code. If the data arrives too early, then the function blocks until it can be put into the buffer.

Elements can be consumed (even if they're not produced yet) after some milliseconds of elements following it has been produced, or then the element being consumed is produced.

```
20 int JitterBuffer::insert(int index, void* src)
21 {
22     if index is smaller than the last consumed element...
23         return 1
24
25     copy {elementSize} bytes from {src} into index {index} of
   buffer
26     update book keeping variables that keep track of last
   produced and such
27
28     return 0
29 }
```

Remove

Copies an element from the jitter buffer into the **dest** pointer.

This function may block until something can be dequeued if nothing can be dequeued immediately.

```
1  void JitterBuffer::remove(void* dest)
2  {
3      waits until there are elements to remove from the buffer
4
5      copy the data at the end of the queue into the {dest}
   pointer
6      update book keeping variables
7  }
```

Circular Buffer

This section contains pseudo code for a buffer. The buffer is a fixed-sized first-in-first-out queue.

Enqueue

Copies the data from **src** into the circular buffer.

```
1  void CircularBuffer::enqueue(void* src)
2  {
3      blocks until there is room in the queue
4      copies {elementSize} bytes from {src} into an element in
   the message queue
5  }
```

Dequeue

Reads an element from the buffer, and copies it to **dest**.

```
1  void CircularBuffer::dequeue(void* dest)
2  {
3      blocks until there is data in the queue to read
4      copies {elementSize} bytes from the queue into {dest}
5  }
```

Message Queue

The message queue class is a dynamically sized queue used to store a queue of messages for a thread. Events are built into it for synchronization purposes.

Enqueue

Copies **len** bytes from **src** into the buffer, and stores the **type** information as well so it can be returned later.

```
1  void MessageQueue::enqueue(int type, void* src, int len)
2  {
3      blocks until there is room in the queue
4      copy {elementSize} bytes from {src} into an element in the
   message queue
5     record {len}, so we can return it later
6     record {type}, so we can record it later
7  }
```

Dequeue

Reads an element from the buffer, and writes it to **dest,** and writes the number of bytres read from the buffer to **len,** and the type of data copied into **type**.

```
1  void MessageQueue::dequeue(int* type, void* dest, int* len)
2  {
3      blocks until there is data in the queue to read
4      copy {elementSize} bytes from the queue into {dest}
5     write the number of bytes copied from the buffer to {len}
6     write the type of data copied to {type}
7  }
```