Design Document

Data Communications: Assignment 3, Comm Audio

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

January 10, 2015

Table of Contents

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

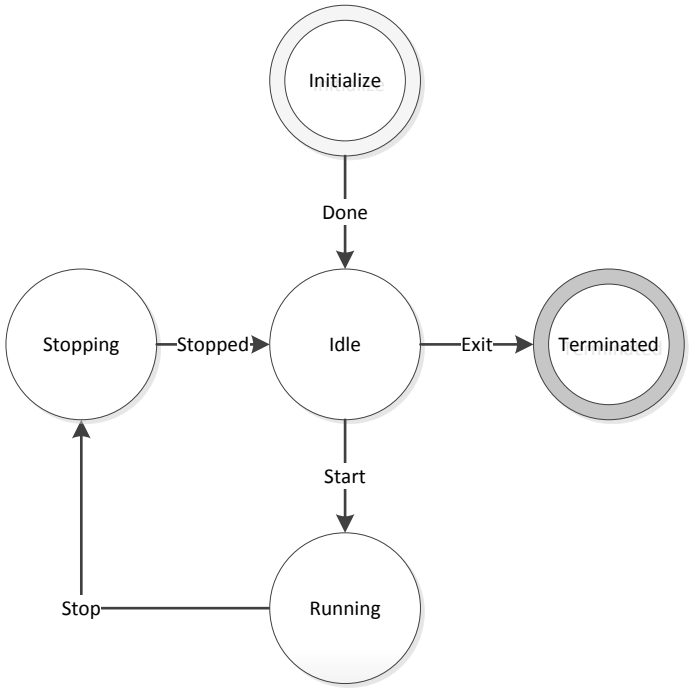
The client application:

- 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.

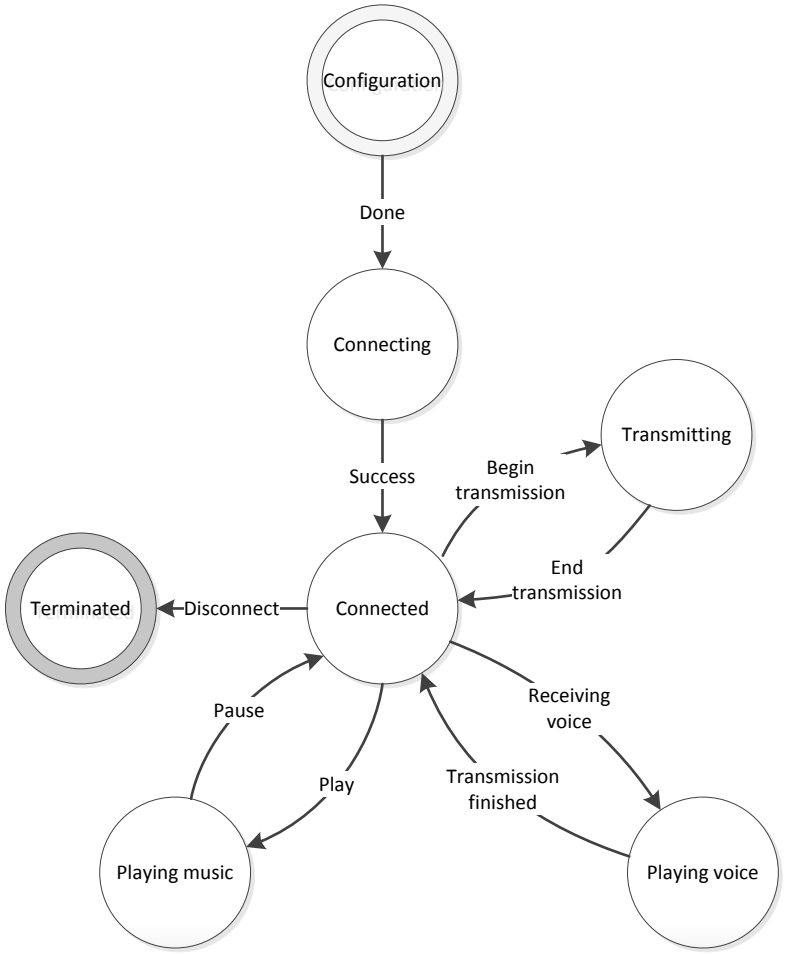
Server



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



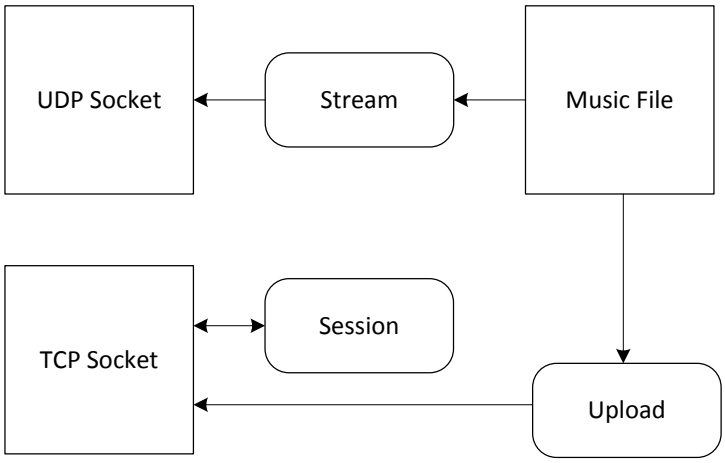
The above diagram illustrates the states of the client application as it appears to the user:

- **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.

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



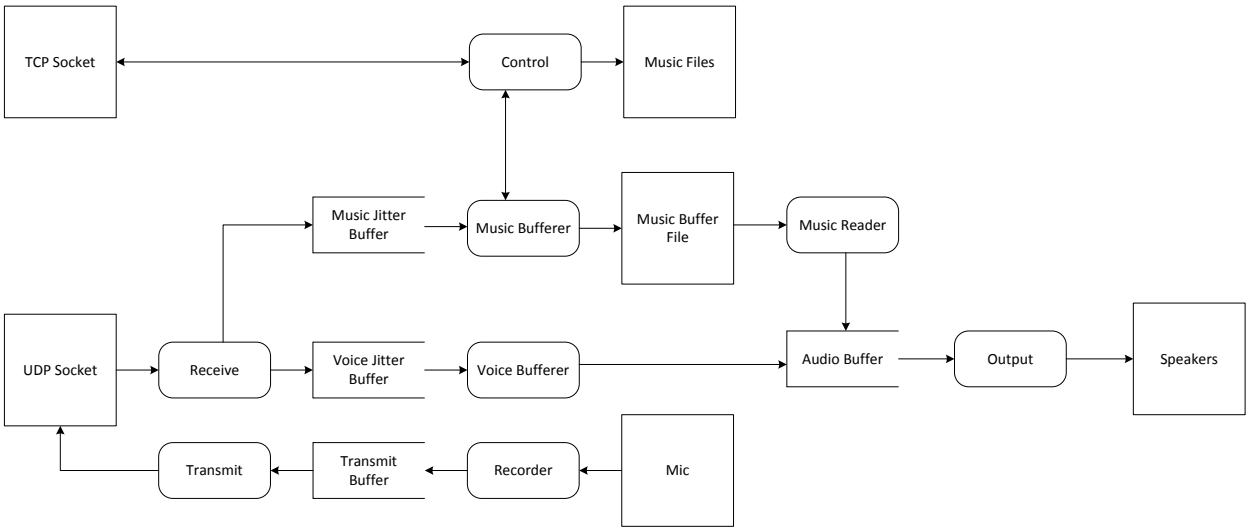
Above is a data flow diagram illustrating how data flows within the server application. The kind of data that circulates in the application includes:

- **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. 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.

As illustrated, 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.

Flow Chart Diagrams

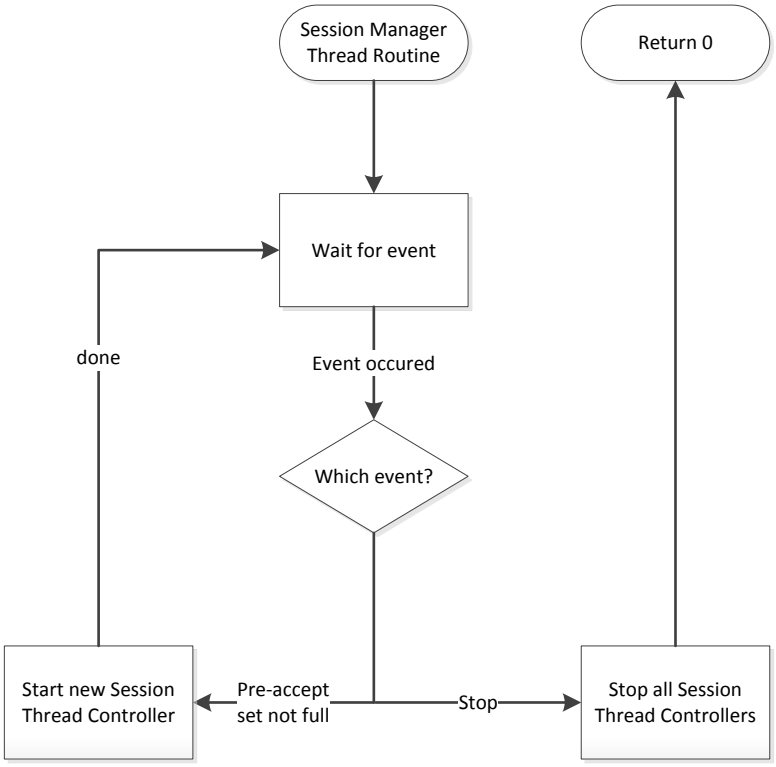
The following flow chart diagrams in this section show how each of the threads identified in the data flow diagrams work from a high level. They show how resources are allocated, used, and deallocated for each thread.

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.

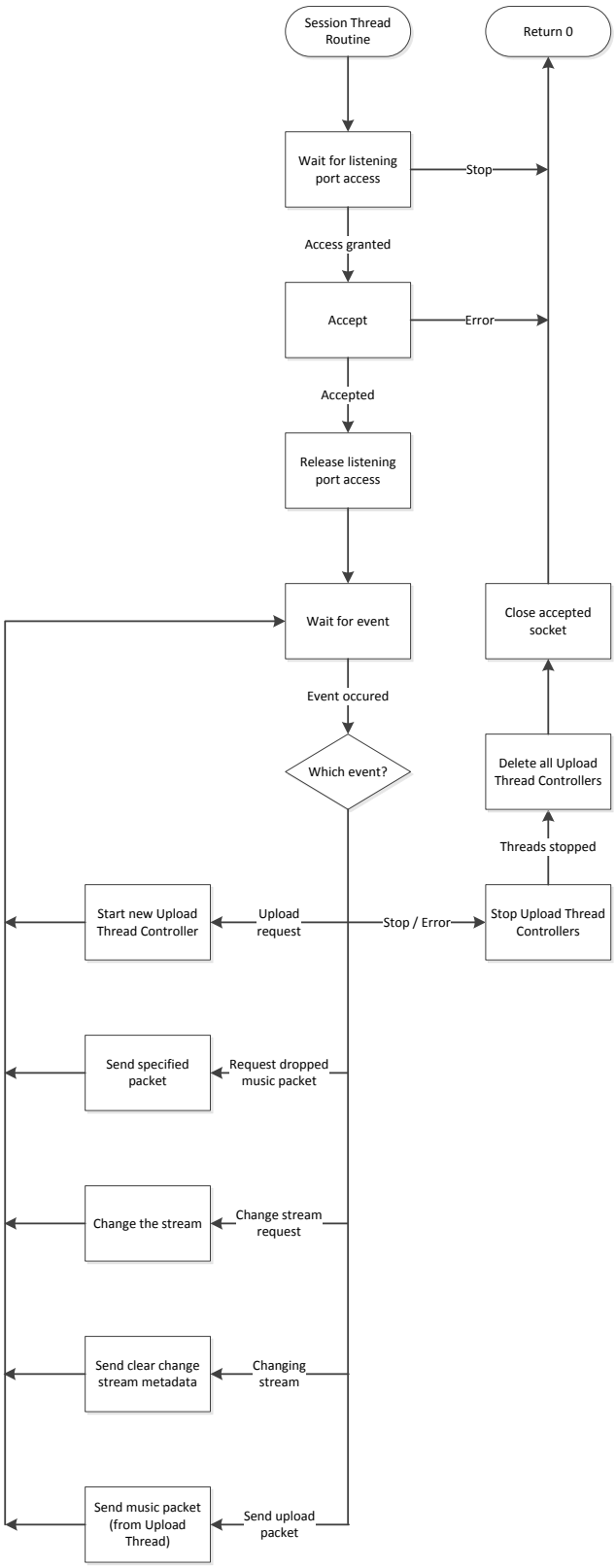
Session Manager



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

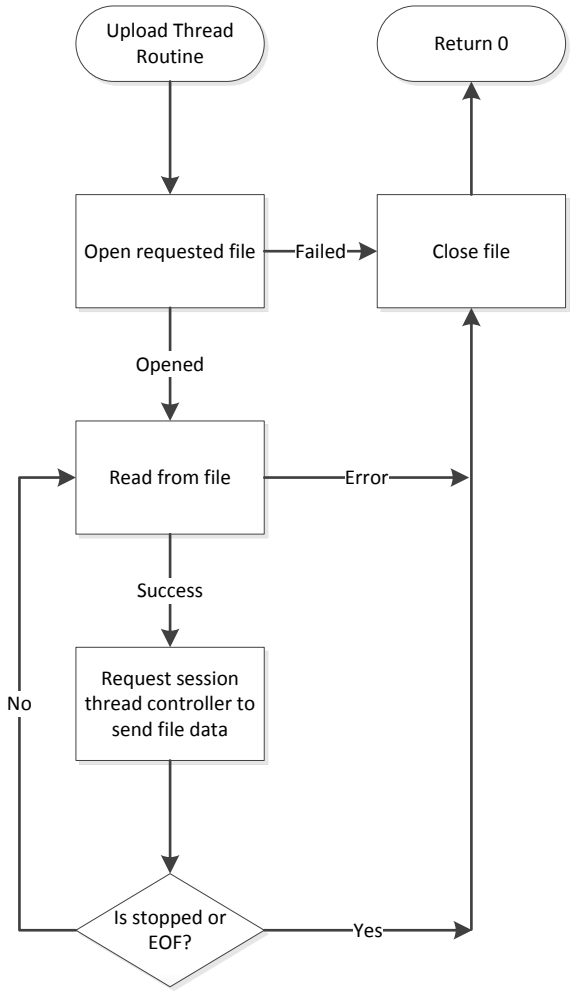


The session thread is used to interact with the client, and listen for its requests, as well as sending the client control information.

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.

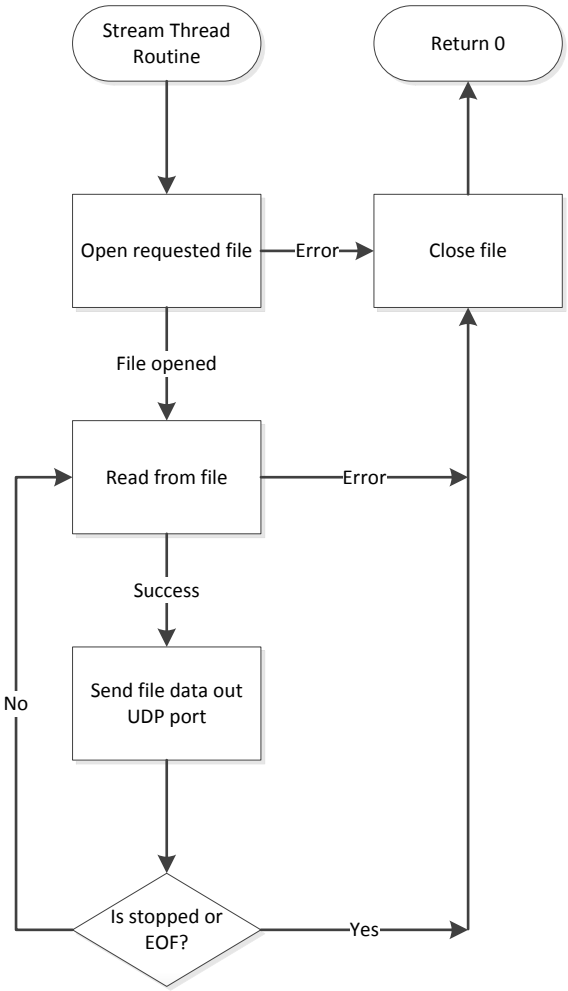
During the thread's operation, it may create upload threads to service the client's download requests. When the session thread is stopped, it stops all the upload threads it created before terminating itself.

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

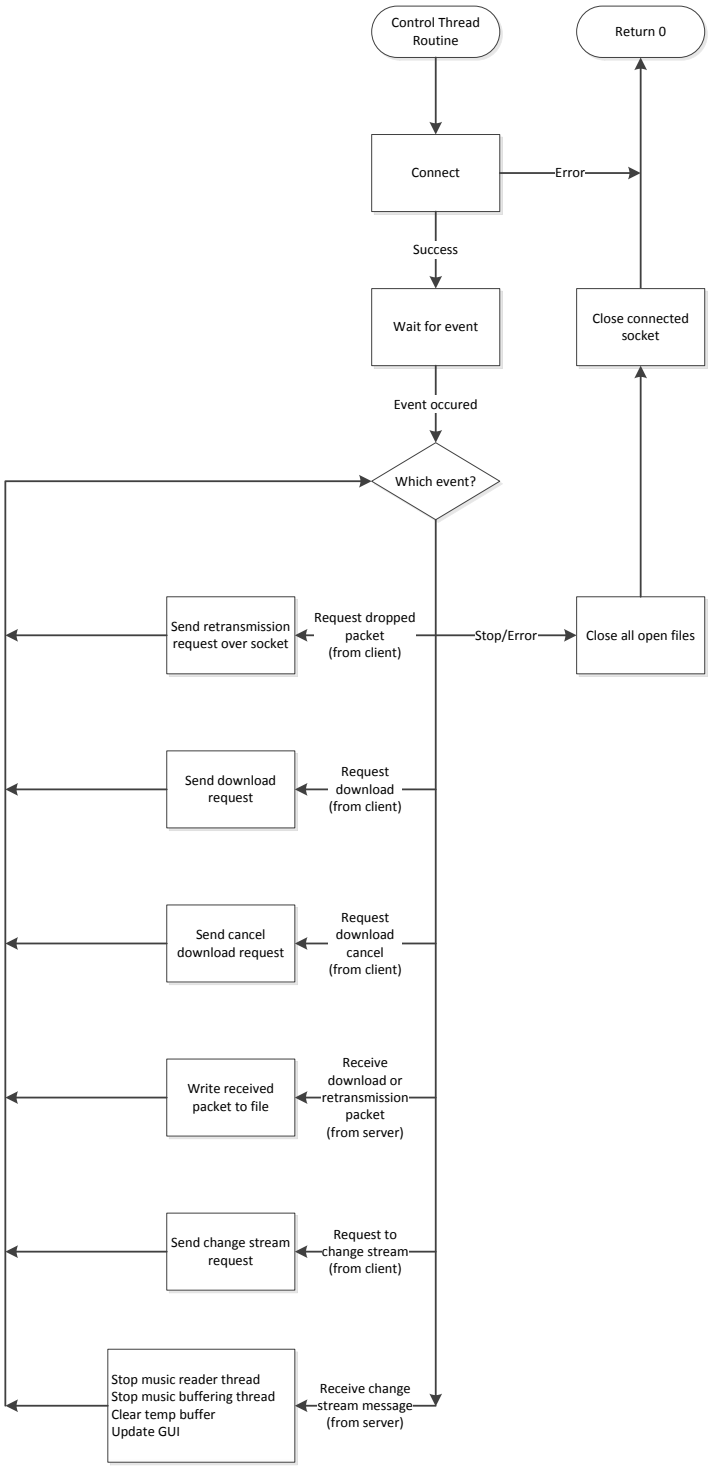


The stream thread is used to multicast a file on the network. There should only be one of these threads running in each server application at a time. This thread is alive when the server is in the Running state. It reads from a file, and multicasts it on the UDP socket until it finishes transmitting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



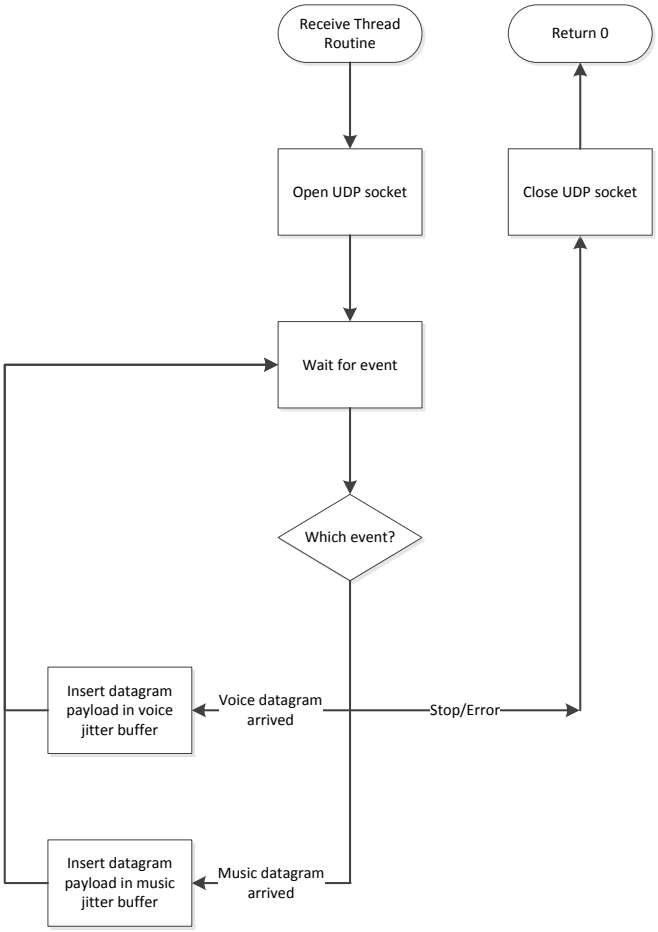
The control thread is used to receive messages from the control line, and communicate with the server. Not all messages that arrive through the TCP socket are control messages.

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

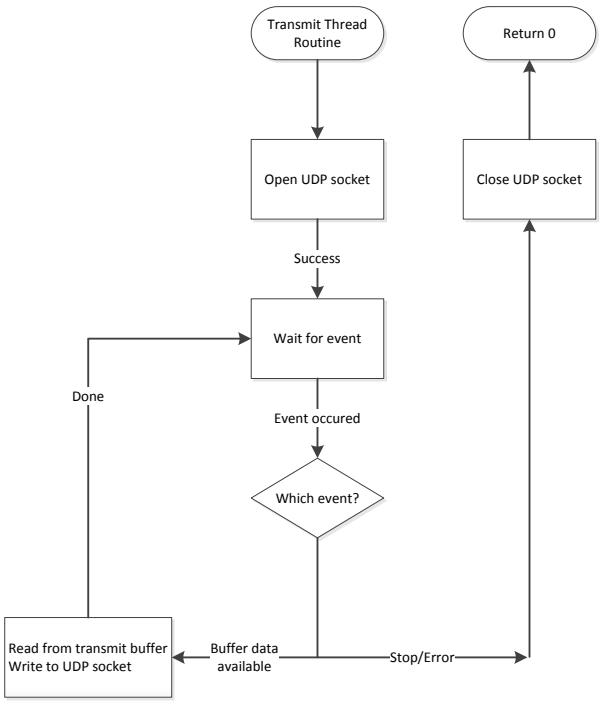


The receive process is used to listen to incoming datagrams on the client's UDP socket. Packets that can be encountered include voice datagrams from other clients, or music datagrams from the server. When such packets are received, they are inserted into their appropriate buffer; music datagrams are 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.

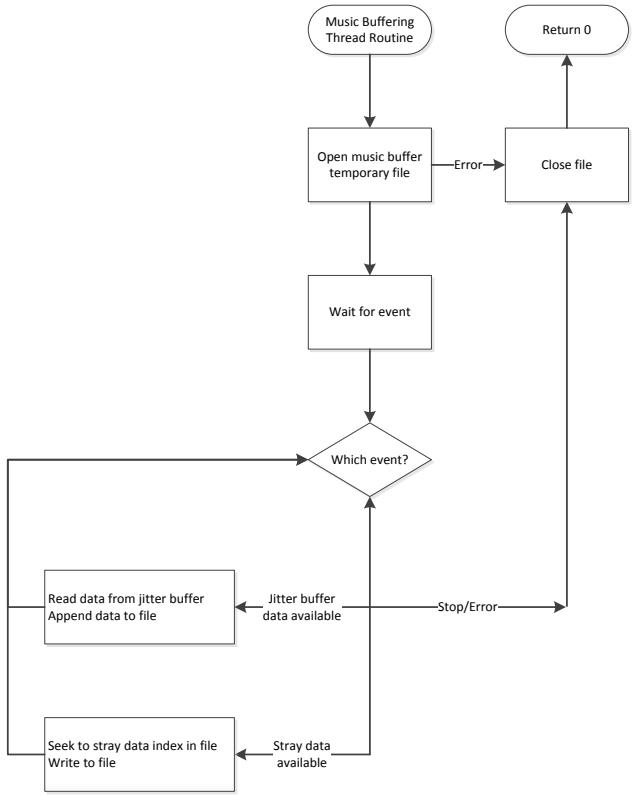
When the thread receives a stop signal, it closes the UDP socket, and terminates.

Transmit



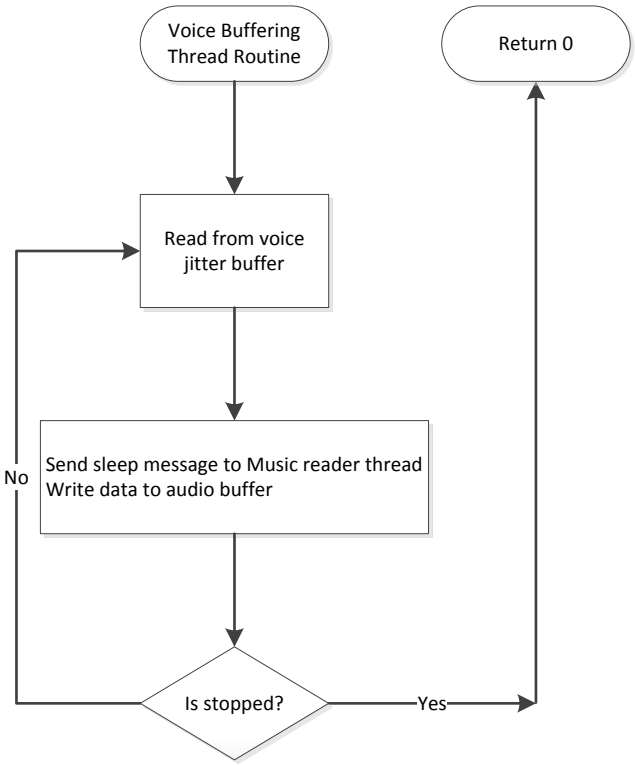
The transmit thread is a simple thread. It continuously reads data from the transmit buffer, and sends it to the user specified address through the application's UDP socket. This thread terminates when it receives the stop command.

Music Buffering



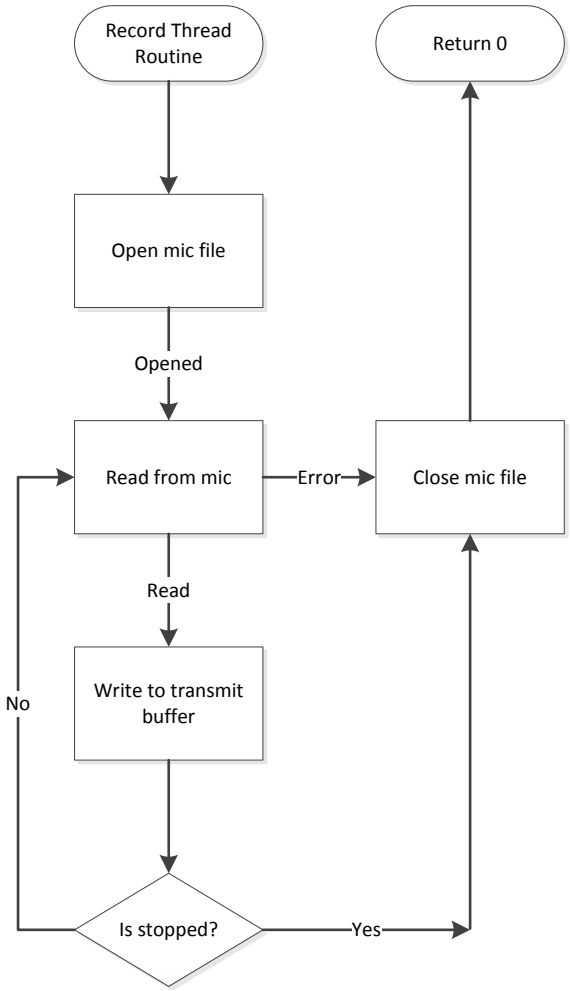
The music buffering thread is used to write data to the music temporary buffer files. It reads stream data from the music jitter buffer, and writes the data to the temporary buffer file. Every once in a while, it may receive a retransmitted packet from the control thread to be written to the temp music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

Voice Buffering



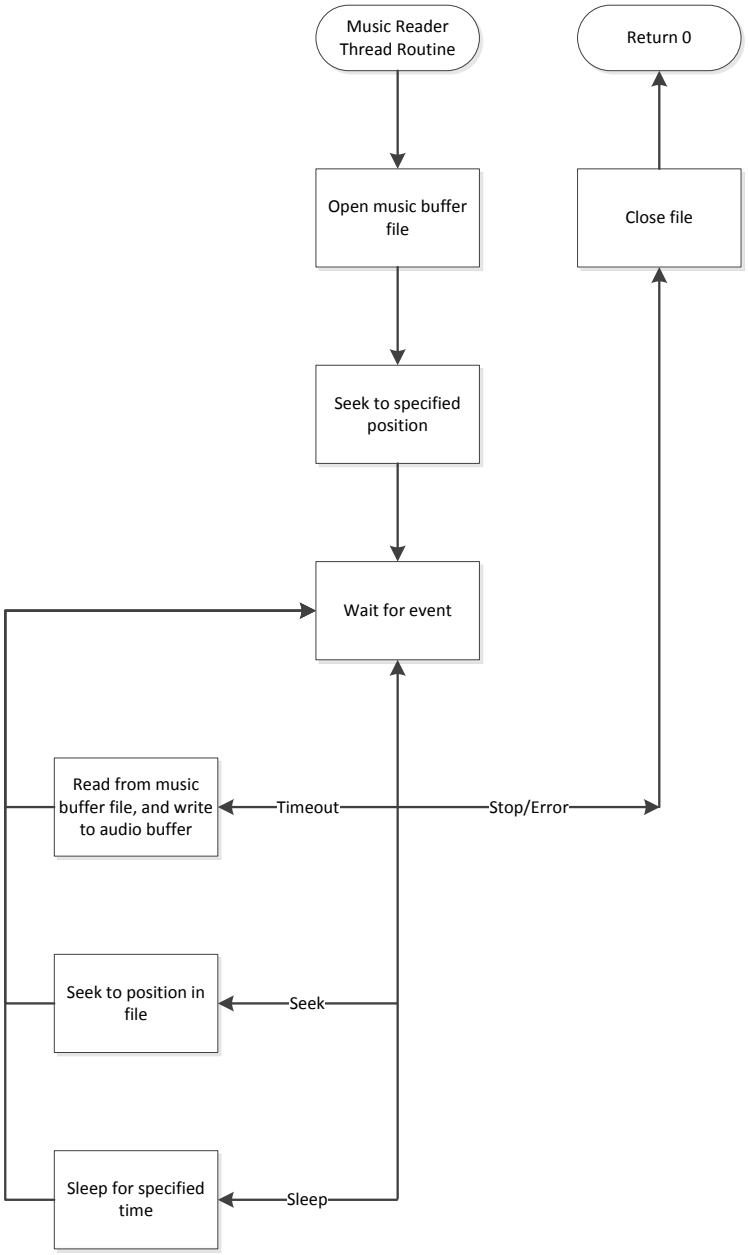
The voice buffering thread is simple. It reads the data from the voice jitter buffer, and writes the read data to the an audio buffer, which is used to play it. This thread continues to do this until it receives the stop event, which terminates the thread.

Record



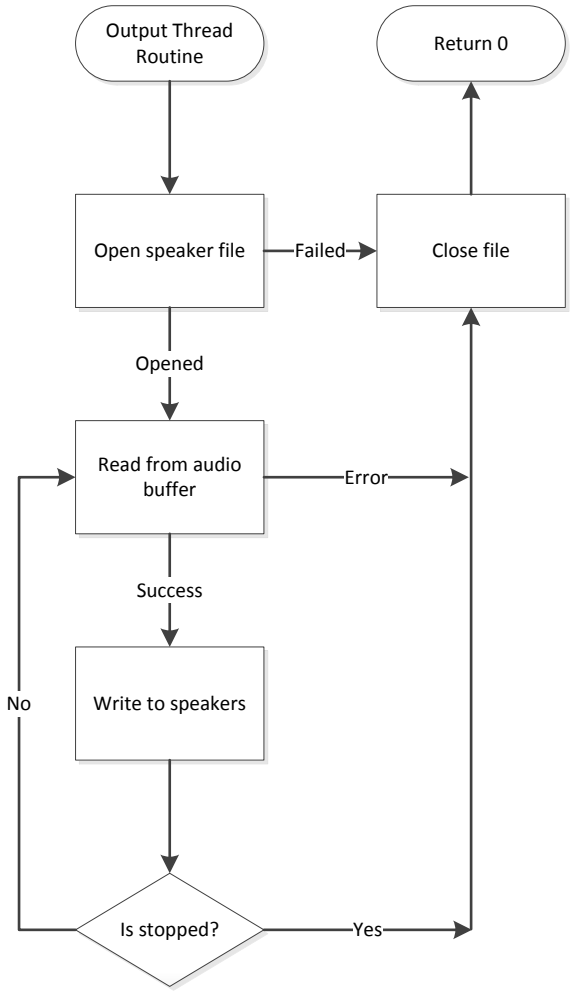
The record thread is simple, and only exists while the client application is in the transmitting state. This thread reads audio data from the system's mic, and places it into the transmit buffer. The thread stops once it receives the stop command, which terminates the thread.

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.

Pseudocode

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