Music Streaming Service

# How to run

Before starting the application put all the songs you would like to use in the **Executables\Server** folder**.** In order to see the application running, first start **WaveyfyServer.exe** from the **Executables\Server** folder.

After that start the client – **WaveyfyFormsApp.exe** from the **Executables\Client** folder.

You can start as many clients as you want, and they will all work with the server

In order to run the client and the server on different machines, change the IP in the files **ip.txt**. Please note that there is one of those files in both **Executables\Server** and **Executables\Client** directories. Do not leave “localhost” in any of the files when testing on different machines. You must supply the IP address of the machine that is going to host the server in both files.

# Functionalities

Music is streamed from the server to the client, using packets with size of 1024 bytes. No music files are stored on the client and the buffer holding the music data is not bigger than that needed for 4 seconds of raw data.

Multiple clients can connect to the same server and get data concurrently

The user can choose a song to be played from a list in the client. Please note that a new song can be chosen at any point (even while the old one is playing) and it will start playing.

The user can pause/play the current song at any point using the respective buttons on the UI

The user can control the volume of the playback using the slider in the bottom left

The user can see the progress of the current track on the progress bar in the bottom

# Testing

## Client Testing

When testing the client I did it in two parts – testing the core services (connection to server, streaming music, playing music) and testing the GUI.

For the core services the main concern was consistency of the communication between the client and the server. The obvious solution was to use “control” packets (instead of just sending a packet with data, send a packet that tells you what the next packet is going to be). This doubles the number of packets being send, but the impact introduced is not that big. On top of that this approach was the way to make sure the client doesn’t crash or enter an endless loop when the server goes down unexpectedly. Testing for this part consisted mainly of shutting down the server and clients with different timings (server first, client first, etc) and making sure the other end doesn’t explode. A bug I found in this scenario was preventing the client from closing if the server was closed down first. This was due to an endless wait for the next package from the server, which never came. The solution to this problem can be seen in the numerous uses of the \_**serverTerminated** variable, which is set to true when a socket error occurs.

When testing the GUI the major problem was the fact that a new thread is now introduced to the system. What this means is that when telling the player to stop playing and directly stop the secondary buffer we might be stuck on a Wait command forever and never be able to actually close the forms application. In order to solve this problem I needed to “lock” parts of the code using variables. In this way when the GUI thread was trying to access any functions of the player object it always knew what state the player object is in. This, of course introduced some more complexity to the code, but removed quite a lot of bugs. The testing for the GUI consisted of issuing commands in varying order and at varying times. For example, there was a bug that didn’t let you choose a new song before the last one was done playing – this was solved by changing the values for the **WaitForMultipleObjects** function so that the code also checks every second if the playback has been stopped from the outside.

## Server Testing

The testing for the server came in two main parts – stress testing (how many clients can be concurrently handled) and reliability testing (can the server be left running for a long time)

## Server Stress testing

The first phase of this test was connecting 25 clients to it in the same time on the same machine. The server application stayed stable, with no memory leaks or CPU spikes. There is, however, some increase in CPU usage when a client disconnects. I believe that comes from the destruction of all the objects used for the streaming of music.

The second phase of the stress testing was connecting the same number of clients but from different machines. Five of the clients were hosted locally, on the same machine as the server, while the other 20 were split on two other machines on the same local network. The behavior of the server was the same as in the previous phase (no CPU usage spikes, except a little increase when a client disconnects).

## Server Reliability testing over long periods

The first test for the server was to leave it running for several hours and then connect a client to it. This worked perfectly, and the client was able to connect, play a song and disconnect from the server.

The second test included leaving the server running for several hours and do a stress test (as the one above) every hour. The behavior was stable and quite predictable. The occasional CPU spikes appeared when a lot of clients go shut down at once, but apart from that no resource leaks were seen