# Netwerken, Spring 2019 Assignment 2: Audio Streaming Protocol

**Deadlines:** Sunday April 28, 2019
Sunday May 19, 2019

## 1 Assignment

In this assignment you will design an Audio Streaming Protocol and implement it. The program must be written in C. You can make use of the skeleton code that is available from the website.

### Protocol

As a first step you should design an Audio Streaming Protocol. This protocol should be designed to be implemented on top of IP, but considering the fact that we have to use the shared lab rooms, we cannot give you the privileges required to use an implementation directly on IP. Therefore, you will use UDP/IP to encapsulate your packets. The protocol should be documented, with the header format in an ASCII-diagram and the values of all the fields explained. A checksum of the ASP frame is required, based on RFC 1071[1]. You are not required to use a pseudoheader for the checksum.

For inspiration, you can look at, for example, the definition of the UDP protocol (from RFC 768[2] ):

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

### Program

Your program should meet the following requirements:

- Your submission should consist of a server and a client program, that communicate over UDP ports 1234 and/or 1235 (these two ports are open to use in room 306/308);
- Your program should be able to simulate an unreliable connection. A `#define`-directive should be available (and documented) to disable this simulation. The simulation should include random dropped packages and a random delay time. Packets that are received out-of-order can be dropped;
- Your protocol should implement 5 quality levels. While streaming, your program should continually check whether the connection is reliable enough and, if necessary, adjust its quality level accordingly. That is, if the connection is unreliable, the quality should be decreased, and if the stability increases, the quality should do so too;

[1] https://tools.ietf.org/html/rfc1071
[2] https://tools.ietf.org/html/rfc768

- Your client program should, when first connected, fill a buffer to allow for smooth streaming. The size of this buffer should have a sensible default and be adjustable with a command line argument (e.g. `client -b 1024` could start the client program with a buffer size of `1024 Kbyte`). You should give the user an indication of the progress;
- Your server program should be able to read a WAVE file in CD quality (Stereo, 44.1kHz, 16-bit) provided by the user as a command line argument;
- Your client program should be able to play the received audio through the ALSA API;
- A Makefile should be included to build the software. Your Makefile should include a `clean` target;
- (BONUS) You can make your server able to stream to multiple clients simultaneously. Each client should, whenever they connect, receive the sound file from the beginning and each connection should be able to switch quality independent of the others. You can earn up to 1.0 point extra by implementing this bonus;

### Documentation

A documentation file has to be written, describing how to use the program, the format and the meaning of all the fields of your protocol, but also describing any significant choices you made in the implementation (e.g. the behaviour of the quality selection, error handling) and anything you want to bring under attention.

## 2 Submission

By April 28, a preliminary version of your implementation is due. This preliminary version should include **at least** a working server-client communication and a version of your protocol definition. Submit this preliminary version by e-mail to *netwerken.liacs@gmail.com*, make sure the subject **is equal to** "Netwerken 2019 assignment 2 preliminary" and include your names and student numbers in the e-mail.

By May 19, we ask you to send a working version of your implementation written in C together with a Makefile and a plain text file containing the documentation of your protocol and programs. You may work in teams of **at most 2 students**. Plagiarism in your submission will lead to a grade 0 immediately. Ensure that you mention your names and student numbers in all files containing your source code and documentation. Submit your work by e-mail to *netwerken.liacs@gmail.com*, make sure the subject **is equal to** "Netwerken 2019 assignment 2 final" and include your names and student numbers in the e-mail. Please send e-mail attachments, Google Drive or Drop-Box links *are not accepted.*

For this assignment, a maximum of 10 points can be obtained. The points are distributed as follows: Code compiles & works (1.0 / 10) Code Layout & Quality, Makefile (2.0 / 10), Network connection & delay and drop simulation (2.0 / 10), Audio Transfer Protocol (2.5 / 10), Dynamic quality switching (2.5 / 10).

Up to 1.0 extra point can be obtained by implementing the bonus extension. Make sure to document the extension. This bonus will only be applied if at least 7.0 points are obtained without the bonus. The final mark for this assignment can never be higher than 10.0.

## 3 Hints

### Sockets

For the communication between client and server, you can use the Socket API. To get an overview of the available system calls, you can read this Introduction to the Socket API[3]. Also, the linux manpages offer a lot of information. You can find some examples of client-server programs at thegeekstuff.com[4], abc.se[5], pacificsimplicity.ca[6] and many more sites.

### The WAVE format

A WAVE file is a simple audio file format consisting of a header and a data part of samples. A description of the header can be found here[7]. Code to read WAVE files is included in the skeleton code.

### The ALSA framework

For this assignment, you will use the ALSA PCM (Pulse-Code Modulation) interface[8] to play the audio on the client computer. After setting up the parameters of the output with `snd_pcm_set_params(...)`, you can directly write samples to its buffer. The parameters you can choose include the sample format, the channel count and layout and the sample rate. The channel layout describes where the samples for the different channels can be found in the buffer. Either the samples are

**interleaved:** for each frame (time step) the samples directly follow each other;

or they are

**non-interleaved:** each channel has an own region in the buffer and the samples for that channel follow each other directly in that region.

In the skeleton code, some sample parameters are used, but you can change these to fit your implementation.

This website explaining common ALSA terminology[9] can be useful.

### Simple audio compression

For the quality selection, you should implement compression. Two recommended ways of (lossy) compressing audio streams are **downsampling** and **bit reduction**.

#### Downsampling

In downsampling, you simply discard every $n$-th frame. For example, if your source audio would have a sample rate of 4Hz, you can discard every 4-th frame to reduce the data size by a quarter. Alternatively, for more agressive compression, you could only pass the first frame and discard the other three to achieve data reduction by three quarters. This compression strongly affects audio quality and all kinds of workarounds exist to minimize this effect. For this assignment, we prefer good audio quality, but implementation of quality optimizations is not required.

[3] `http://phoenix.goucher.edu/~kelliher/cs43/mar19.html`
[4] `https://www.thegeekstuff.com/2011/12/c-socket-programming/`
[5] `https://www.abc.se/~m6695/udp.html`
[6] `https://www.pacificsimplicity.ca/blog/c-udp-client-and-server-example`
[7] `http://www.topherlee.com/software/pcm-tut-wavformat.html`
[8] `https://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html`
[9] `https://larsimmisch.github.io/pyalsaaudio/terminology.html`

### Bit reduction

Another way to cut down on data size, is by reducing the amount of bits every sample needs. By reducing its representation from for example 16 bits to 8 bits, the size of the data is cut in half. This can also cause artifacts in the audio, so a balanced combination is probably best.