Netwerken, Spring 2020 Assignment 2: Audio Streaming Protocol

Deadlines: Saturday May 9, 2020
Saturday June 6, 2020

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); *Due to the* ***unusual*** *events and closed university this semester*, we cannot expect from you that your program works on the lab room computers. For that reason, **we accept solutions that work on a Ubuntu 18.04** (or newer) OS. In theory, if you get it to work on that OS, it should work on the lab computers, too. Also, it works on my machine in that case.
- 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, in a plain text format (.txt, .md) describing how to use the program, the format and the meaning of all the fields of your protocol, and 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 (which may affect your grade). Please note that .md files is explicitly permitted. Should you pick use .md, know that both Markdown-style tables and an ASCII-diagram inside a multiline-code block are accepted.

Report

We also expect a report (.pdf). Here, you are to provide implementation details. Some examples for you to get inspiration:

- Why you chose a protocol with bursting packets?
- Why did you make that ringbuffer implementation?
- What error-detection function you used (and in short why it works)
- What you do to your audio when quality should drop? (and how much space (in percent) does your method free up?)
- How do you measure when quality should drop?
- How did you make sure the music of your packets are placed in-order in the ALSA buffer?

Also, we expect it to contain an introduction and conclusion. Should you be interested in a bonus, do some interesting experiments with your framework and make a 'Results'/'Evaluation' section. For more details about this bonus, see Section 2.1 below.

2 Submission

Part one is due on **Saturday May 9, 2020**. For part one, we expect a preliminary version of your implementation from you. 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 *s.f.alvarez.rodriguez@umail.leidenuniv.nl*, make sure the subject **is equal to** "*[CN] 2020 assignment 2 preliminary*" and include your names and student numbers in the e-mail.

Part two is due on **Saturday June 6, 2020**. We ask you to send a working version of your implementation written in C together with a Makefile and a text file (.txt, .md) 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 your report. Submit your work by e-mail to *s.f.alvarez.rodriguez@umail.leidenuniv.nl*, make sure the subject **is equal to** "Netwerken 2020 assignment 2 final" and include your names and student numbers in the e-mail. Please send e-mail attachments. Using online services like Google Drive or Drop-Box links *are not accepted.* If you find your zip is too large (larger than 10MB for most mail clients), you should: 1) perform make clean if you still have object files/executables. 2) Remove some/all .wav files.

2.1 Grading

Below is the way of grading. You can get a total of 100 points normally, and there are 15 bonus points available. Each section starts with the number of points you can get. Your grade is computed as Grade = min(score/10.0, 10.0).

Program Quality - 30 points

Important things here are:

- Code readability (no 300 line functions, correct usage of types)
- Usage of structs and enums, and const, static, inline keywords
- Shared code between server and client
- No magic numbers/chars (use defines)
- Useful comments (e.g. so it is clear what each function does)

Protocol Definition - 25 points

Important things are:

- Well-definedness (aka clear, no ambiguous stuff)
- Robustness (aka TA cannot find a way to make your protocol fail)
- Good documentation of protocol

Report - 15 points

Report. Import things are:

- Short introduction
- Implementation section, containing explanations of what you did (and of course why and how you did it)
- Experiments? See the bonusses below. Note you don't have to do them to get 15 points here, as it is a bonus
- Conclusion (short/long depending on how much you have to tell)

Misc - 20 points

Things like Commandline functionality, quality of downsampling/bitreduction implementations, quality of simulation of unreliable connections.

Spell name correctly - 10 points

Bonuses

10 points if score>70, and implemented the bonus: multiple clients per server.
05 points if score>55, and you performed some experiments with your implementation:

- What is the average speed of your network in MiB/s?
- Does speed vary if you run server and client on different machines? (if you can only work on 1 machine, come up with a different experiment)
- Come up with some other experiment, preferably something you can plot

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.

Sending your own music

This assignment is more fun if you can send and receive your favorite music. Here are the steps to get your favorite music in a WAV file, which the skeleton code can read and play:

1. Find some music you like with length between 5 seconds and 30 minutes
2. Get it on your computer as mp3 file
3. Open that file in Audacity (a free audio editor tool, available on all major platforms). Just start Audacity, click on with file > open..., navigate to your file
4. In Audacity, go to file > export > export as wav
5. Set the type to WAV (Microsoft) signed 16-bit PCM, pick a folder and press save

[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`

6. In the "Edit metadata tags", press "clear", then "OK"

Now, you have suitable WAV file to send over the network!

The reason not every WAV file works out of the box is: A WAV file consists of 1 header, followed by music data. The more tags, categories, and other meta information you store, the larger the header becomes. This results in shifted music data. The framework is simple, and expects the WAV header to not contain any meta information. This is why you have to strip this information with Audacity.
Also, the framework handles only WAV files which are stereo channel, PCM-encoded, at 44100kHz.

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.

Note: ALSA is a Linux-only audio driver, meaning: Your client must run on a Linux OS. If you don't have a Linux OS installed on your machine, you can: 1) Use only University computers. 2) Install a virtual machine manager (e.g. well-known 'virtualbox').
Also note: the Windows Linux subsystem module will not work, because this subsystem does not provide the required ALSA audio drivers. You must fully emulate a Linux OS with e.g. 'virtualbox'.

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.

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.

[8] `https://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html`
[9] `https://larsimmisch.github.io/pyalsaaudio/terminology.html`