Computer Networks, Spring 2021 Assignment 2: Audio Streaming Protocol

Deadlines: Friday April 9, 2021
Friday May 21, 2021

1 Introduction

Just imagine you live in a world where a strange virus is spreading. The governments of the world tell the people to stay at home and no longer visit each other until the spreading of the virus has been stopped.

Little did everyone know, the virus is not easy to contain, and the people have not been able to visit friends for a long time now. Everyone becomes more and more bored, being separated from their friends for so long.

It was then when it came to your mind: An audio streaming service would help the people of this world! Everyone would be able to stream their favourite music and recordings to friends, making everyone happy once more!

Eagerly, you started working on the design of a streaming protocol.

1.1 Formal Introduction

In this assignment, you will design an Audio Streaming Protocol and implement it on the UDP/IP stack. The program must be written in C. You can make use of the skeleton code that is available from the course website (`https://liacs.leidenuniv.nl/~wijshoffhag/NETWERKEN2021/`). Teams of **two people** are allowed.

The program consists of a server and client program. The server reads the audio file (a `.wav` file), and streams it to the client (when it is connected).
During this assignment, you are encouraged to create your own `.wav` files to use for testing. We explain how to create your own `.wav` file in Section A.4. Finally, you are encouraged to form a team and not do this assignment alone, because it is a lot of work for a single person.

2 Installation

For this project, we use the ALSA audio driver to play audio. The ALSA drivers only exists for Linux distributions, where they are installed by default. This means that you *must have* a Linux machine. If you have a Linux machine around, please use that.

If you do not have a Linux machine, you can either install Linux on a device you have, or, you can install a Virtual Machine Manager on your current device, and then run a Linux (e.g. Ubuntu) VM.
In previous years, students tried to solve this problem using their own solutions.
Here is a list of things that actually **do** work:

- Install Linux on a machine and work with that.
- Install a Virtual Machine Manager on your current device, and then run a Linux (e.g. Ubuntu) VM.

Here is a list of things that **will not work**:

- Windows 'WSL' Subsystem.
- Ubuntu on Windows (a collaboration to make an Ubuntu-like environment run on Windows).
- Ubuntu Terminal on Windows.
- Using a MacOS distribution.

If you have no machine running Linux, and you do not know how/want to install Linux or run a Linux VM, then find a partner who has a Linux machine.

3 Implementation

Here, we explain what your implementations should be capable of. We separately address base requirements, report requirements, code quality requirements, and bonuses you may obtain.

3.1 Base

Your program has to satisfy **all** of the base requirements explained here to be able to get a full mark.
The basic program requirements are:

1. You need to build a client and a server application.
2. The programs must work on a standard **Ubuntu 18.04** (or newer) OS. In theory, if you get it to work on that OS, it should work on University machines when grading as well.
3. The programs must be able to communicate over UDP/IP, by default using ports 1234 and/or 1235.
4. 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.
5. The client program has to play audio through the ALSA API.
6. The programs must be built using a Makefile, which you add in your submission files. The makefile must have a `clean` target, which cleans up object files and executables.
7. The client program should, when first connected, fill a buffer with the first part of the audio. This allows for smooth streaming. The size of the buffer should have a sensible default, and it must be configurable with a commandline argument.
8. The client program has to give some indication of how far we are in the played audio. For example, a client program can print how many percents of the audio file has been played until now.
9. For both server and client programs, you must write a documentation file in a plain text format (.txt, .md). The documents should describe **all of the below**:
   (a) how to build and use the server/client program (e.g. make command, commandline arguments)
   (b) describe any significant choices you made in the implementation (e.g. the behaviour of the quality selection, error handling)
   (c) anything you want to bring under attention (which may affect your grade)

   Please note that using .md files is explicitly permitted. Should you decide to use .md, know that both Markdown-style tables and an ASCII-diagram inside a multiline-code block are accepted.

Additionally, functional requirements are:

1. Your program should be able to simulate an unreliable connection. A commandline argument should be available to enable this simulation. The simulation must include **all of the below**:
   (a) Randomly dropped packages on the receiving end.
   (b) Randomly swapped packages before sending on the sending end.
   (c) Random delay times, with **configurable** min and max delay time (in milliseconds).
2. Your program has to implement 5 quality levels (min quality= 1, max= 5). While streaming, your program should continually check whether the connection is reliable enough and, if necessary, adjust its quality level accordingly. So, if the connection is not reliable enough for the current quality level, the quality should be decreased. If the connection is stable enough, the quality should be increased. When the connection is unreliable, you will have to decrease the amount/quality of data you send. See Section A.5.

All of the above requirements have to be satisfied.

3.2 Report

We also expect a report (.pdf). In the report, you are to provide:

1. a short introduction.
2. implementation details. You are required to write down the following:
   - Explain what sort of protocol you made. Is it a burst-protocol, or a sliding window protocol, or something else?
   - Give a RFC 768 packet definition for your packets (see Section A.1).
   - Explain the communication between the client and server. For example, explain how the client asks the server for data, and how the server responds.
   - Explain how you detect faulty/lost/out-of-order packets, and how your protocol fixes these problems.
   - Explain what each quality level does: (e.g. "with quality=1 we reduce sample size from 16 to 8 bits and downsample to 50% of the frames. With quality=2...")
3. a conclusion (short/long depending on how much you have to tell).

Of course, add any additional interesting content in your report.
Should you be interested in a bonus, do some interesting experiments with your framework and make a 'Results'- or 'Evaluation' Section. For more details about the experiments you have to conduct for this bonus, see Section 3.4 below.

3.3 Code Quality

To ensure we get programs of decent quality, we impose a few quality requirements on all implementations. Here we list quality requirements we expect from students:

- Implementations have to compile without any warnings and errors.
- Implementations should adhere to one style only.
- There should be no memory leaks caused by your programs. Note: ALSA internal memory is always considered to be leaked, because ALSA is a driver, and the driver does not stop running when our program finishes. Of course, we will ignore these detected leaks.
- Implementations cannot have huge functions. We define a function as huge when it has approximately 50 to 55 statements (= 50 to 55 lines of code in most styles). We prohibit this, because that kind of function quite commonly is a unreadable blur of code that performs way too much work.
- The purpose of 'non-trivial' functions need to be explained by using a documentation comment above said functions.

In common: Pay attention to good style- and coding practices when programming for this assignment, and try to create code as beautiful as possible.

3.4 Bonuses

For this assignment, we have the following bonuses, which bring extra points once implemented. Note that your grade is at most 10.0. Finally, note that we only apply bonuses when the base grade (the one before applying bonuses) is at least 5.5.

1.0 pt Implement the server program such that it can stream to multiple clients simultaneously. Clients can join and leave the server at any point, and there is no maximum on the number of clients that can connect to your server.

0.5 pt Perform **all of** the following experiments with your implementation, and add them in your report:

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

4 Grading

Here, we publish the way we grade implementations for your benefit. Implementations can get a total of 10.0 points normally, and there are 1.5 bonus points available. Each section starts with the number of points you can get. Your grade is computed as Grade = min(points, 10.0).

Code Implementation - 3.0 points How well your implementation satisfies the base requirements, as given in Section 3.1. This includes simulation of unreliable connections, downsampling/bitreduction implementations, etcetera.

Report - 2.0 points 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 bonuses Section, Section 3.4. Note you don't have to do them to get 2.0 points here, as it is a bonus.
- Conclusion (short/long depending on how much you have to tell).

For more information, see Section 3.2.

Code Quality - 3.0 points Important things here are:

- Code readability (no huge functions, correct usage of types).
- Consistent style.
- 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).

For more information, see Section 3.3.

Protocol Definition - 2.0 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.

5 Submission

Due to the size of this assignment, we split the deadline in two parts.

5.1 Part one

Part one is due on **Friday April 9, 2021**. For part one, we expect a preliminary version of your implementation. 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] Assignment 2 preliminary*" and include your names and student numbers in the e-mail.

5.2 Part two

Part two is due on **Friday May 21, 2021**. You need 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**. 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** "*[CN] 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, and (2) Remove some/all .wav files.

5.3 Final Words

For late submissions, we subtract 1 point from your grade (so out of 10 points) per day.

A Appendix

A.1 RFC 768 Packet Definition & Documentation

When designing your Audio Streaming Protocol, you will decide the 'header layout' of one or more packet types. By agreeing on a header layout, both server and client know how to read incoming bytes from remote machines.

To document this layout, you have to follow the RFC 768[1] standard. Simply put, your protocol should be documented, with the header format in an ASCII-diagram and the values of all the fields explained. Here is an example of an ASCII-diagram for the definition of the UDP protocol (excerpt from RFC 768):

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

A.2 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[2]. Also, the Linux manpages offer a lot of information. You can find some examples of client-server programs at thegeekstuff.com[3], abc.se[4], pacificsimplicity.ca[5] and many more sites.

A.3 ALSA Framework

For this assignment, you will use the ALSA PCM (Pulse-Code Modulation) interface[6] 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

[1]`https://tools.ietf.org/html/rfc768`
[2]`http://phoenix.goucher.edu/~kelliher/cs43/mar19.html`
[3]`https://www.thegeekstuff.com/2011/12/c-socket-programming/`
[4]`https://www.abc.se/~m6695/udp.html`
[5]`https://www.pacificsimplicity.ca/blog/c-udp-client-and-server-example`
[6]`https://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html`

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[7] can be useful.

Note: ALSA is a Linux-only audio driver, meaning: Your client must run on a Linux OS. See Section 2.

A.4 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[8]. Code to read WAVE files is included in the skeleton code.

A.4.1 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
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.

A.5 Quality

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

A.5.1 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 these workarounds is not required.

[7] `https://larsimmisch.github.io/pyalsaaudio/terminology.html`
[8] `http://www.topherlee.com/software/pcm-tut-wavformat.html`

A.5.2 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.