Computer Networks, Spring 2023 Assignment 2: Audio Streaming Protocol

Deadlines: Friday 21st April, 2023
Sunday 4th June, 2023

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!

Unfortunately, you are also great at procrastination, so this reality is all but over. However, undeterred by trivial matters such as the real world, you start working on the design of your 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 on Brightspace, and the slides available from last year's course website (`https://liacs.leidenuniv.nl/~wijshoffhag/NETWERKEN2022/`). Teams of **one or 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 (but not required) 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 SDL2 (`https://www.libsdl.org/`) to play audio. SDL2 is famously cross-platform, and should work on just about any platform where you can get it to compile.
SDL2 can be installed simply using:

```
sudo apt install libsdl2-dev
```

Or any similar command using a package manager of your choice.

The provided Makefile should work on most Linux systems. This includes Windows Subsystem for Linux, and WSLg (`https://github.com/microsoft/wslg`), which is included in the relatively up-to-date Windows 11 builds.

Since SDL2 is not available on the LIACS systems in any meaningful way, a build has been provided at `/vol/share/groups/liacs/scratch/cn2022/`. The provided Makefile includes provisions to build using this, by invoking `make` as:

```
make SDL2_CONFIG=/vol/share/groups/liacs/scratch/cn2022/bin/sdl2-config
```

In case of any technical troubles related to SDL2, you can always ask questions during the weekly labs or at *x.n.lim@umail.leidenuniv.nl*.

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 and relatively modern Linux-based OS.. The assignment won't be graded on University machines, so working on these is not a requirement. As long as any issues that arise due to differing environments are relatively easy to fix, it's fine.
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 16-bit WAVE file with an arbitrary sample rate and number of channels. The sample rate and channel count should be read from the input file, and the input file should be provided by the user using a command-line argument or by direct input.
5. The client program has to play audio using SDL2. A playback framework is provided to do so.
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 (≤ 1 seconds of audio), and it must be configurable with a command-line argument.
8. During playback, the amount of buffered audio should not significantly exceed this buffer size. This means that the sending of audio data may need to be stopped and started multiple times during playback. Your protocol should be able to handle this.
9. The client program has to give a human-readable indication of the total track duration, and how far playback is at the current time. This can, for example, be a percentage, or a `mm:ss` display. This value should update as playback continues, and your protocol will have to be able to communicate the total duration to the client.
10. 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, `command-line` 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 or may not 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. As is common with networking, your protocol should operate using a big-endian (BE) byte order. This means that packet headers, etc. should use BE, but it doesn't apply to the (potentially compressed) audio data being sent. With the exception of M1 and M2 Macs, most PCs and laptops use little-endian x86_64, meaning your protocol fields should be decoded after receiving them but before using them. As such, on every platform you may use the following functions for endianness conversion:

- `htobe64` and `be64toh` for 64-bit integers. Found in the header `endian.h`, which should be included **after** defining the macro `_BSD_SOURCE` (see the relevant man pages for more details).
- `htonl` and `ntohl` for 32-bit integers. Found in the header `arpa/inet.h`.
- `htons` and `ntohs` for 16-bit integers. Found in the header `arpa/inet.h`.

For 32-bit and 16-bit integers, you may also use the functions provided in `endian.h`. For all these functions, `hton` means the conversion is from the **host** byte order to the **network (BE)** byte order, and `ntoh` performs the reverse.
Failing to perform this conversion means your protocol may not be able to communicate to a client or server with a different endianness.
NOTE: as mentioned, you do not have to swap the byte order of the audio data. This is because this is a property of the audio data being sent, not of the protocol. As such, a client on a big-endian system should still be able to play little-endian audio data, and this is what your program should do.

2. Your program should be able to simulate an unreliable connection. A command-line 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).
3. Your program has to implement 4 (or more) quality levels. 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. You can judge connection quality by, for example, detecting dropped packets on either side, or buffer underruns on the client side. See Section A.7 for some information on implementing compression.

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=4 we reduce sample size from 16 to 8 bits and downsample to 50% of the frames. With quality=3...")
3. A conclusion (short/long depending on how much you have to tell).
4. Anything fun or interesting you may have to say about your submission or the assignment itself.

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 can 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 is a list of requirements we expect from students:

- Code may use any version of the C standard starting from C89.
- You may not use external libraries, with limited exceptions (such as using `pthread` for connection handling). If you are unsure, you can ask if a library is fine for what you're using it for.
- Compilation should be done with as many warnings enabled as is reasonable (`-Wall -Wextra -Wpedantic`).
- Implementations have to compile without any warnings and errors.
- Implementations should use a consistent style.
- There should be no memory leaks caused by your programs. This can and will be checked using i.e. `valgrind`. Note: SDL will allocate internal memory that will be considered leaked by tools such as `valgrind`. Of course, these leaks will be ignored.
- No use of uninitialized memory, etc.
- The code style should value readability. Use plenty of blank lines and comments to segment and document your code. There is no hard limit on function size, but a guideline is given at 50-55 lines excluding comments and blank lines. This is to prevent the function from becoming an unreadable blur of code, and to encourage modularity and reduce code duplication.
- 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 theoretical 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, for example, kilobit per second? Taking the audio bitrate of the uncompressed audio into consideration, what kind of compression ratio is achieved, and how does this compare to the theoretical ratio based on your compression algorithm? (This measures the protocol overhead).
- 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. Note that **the implementation, the report and the protocol definition are required to be present**, and omission of any of these means you will not pass this assignment. 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).

Implementation - 3.5 points How well your implementation satisfies the base requirements, as given in Section 3.1. This includes simulation of unreliable connections, downsampling/bitreduction implementations, byte order handling, etc.

Report - 1.5 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).
- No memory leaks or other invalid memory usage.

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, with the first deadline being optional.

5.1 Part one

Part one is due before **Saturday 22nd April, 2023**. For this, we expect a preliminary version of your implementation, on which some feedback will be given. 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 *x.n.lim@umail.leidenuniv.nl*, make sure the subject **is equal to** "*[CN] 2023 Assignment 2 preliminary*" and include your names and student numbers in the e-mail. Note that this deadline is **optional**, but highly recommended to get some feedback on your work.

5.2 Part two

Part two is due on **Sunday 4th June, 2023**. Again, submissions until **Monday 5th June, 2023 8:59** are considered on time. 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 **1 or 2 students**. Ensure that you mention your names and student numbers in your report. Submit your work by e-mail to *x.n.lim@umail.leidenuniv.nl*, make sure the subject **is equal to** "*[CN] 2023 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 25MB for most mail clients), you should (1) perform `make clean` if you still have object files/executables, and (2) Remove some/all .wav files.

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

6 Final Words

If anything in the requirements, explanation or code in the assignment is unclear, or if there are any other issues, do not hesitate to contact me at *x.n.lim@umail.leidenuniv.nl.*

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 SDL2

For this assignment, you will use SDL2. Extensive documentation can be found on it's wiki at `https://wiki.libsdl.org/`.

After opening an audio device with the correct parameters using `SDL_OpenAudioDevice`, you can queue audio to be played simply by calling `SDL_QueueAudio`.

Some notes about usage:

- An audio-player "class" is provided. This should be sufficient to play audio, but of course can be extended upon (or entirely replaced).
- This player includes functionality to sleep until the audio buffer is nearly empty, which can be used to keep the amount of buffered audio below a threshold. Usage of this can be found in the source code of the demo player.
- SDL can take care of the conversion of any unsupported sample rates or channel numbers, so calling `SDL_OpenAudioDevice` with the right parameters is enough to play back audio with the desired sample rate and channel count.
- Audio in your .wav files will be interleaved, and this can be passed directly to SDL.

  interleaved: the samples for each frame directly follow each other (1 frame consists of 1 sample from each channel)

[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`

planar: each channel has it's own memory region containing all that channel's samples. Since SDL2 does not natively support this, you don't have to support it either.

Demo player

A demo player showing how to read .wav files and how to play and queue audio is provided, under the name `wav_player`. Passing a .wav file as a command line argument will print some info and play the file, with a progress bar. Playback can be interrupted with Ctrl+C. You may use this as a reference for your own client and server.

A.4 The WAVE Format

A WAVE file is a RIFF-based (`https://en.wikipedia.org/wiki/Resource_Interchange_File_Format`) audio file format.

A RIFF file consists of chunks with a 4-byte identifier and an unsigned 32-bit size. WAV files consist of a single RIFF chunk, which contains an identifier and subchunks. The first subchunk is always the `"fmt "` chunk (note the space), followed by arbitrary chunks, one of which will be a `"data"` chunk. This chunk contains the audio data we're interested in. A reference for the WAV file format can be found at `http://soundfile.sapp.org/doc/WaveFormat/`.

A.5 Obtaining audio

You may use a program such as `yt-dlp` (`https://github.com/yt-dlp/yt-dlp`) to download audio from a site like YouTube for your own use. When `ffmpeg` (`https://ffmpeg.org/`) is installed, this can also convert the downloaded file directly to a .wav file. An example command to download a song and save it as a .wav file is:

```
yt-dlp -x --audio-format wav -o "Tower of Flower.wav" https://youtu.be/Rwzy6Qt8gq8
```

This will produce a file called "Tower of Flower.wav", which will contain the audio of the video at the url `https://youtu.be/Rwzy6Qt8gq8` in .wav format. Make sure to check that it's a 16-bit little-endian .wav file. The command

```
ffprobe "Tower of Flower.wav"
```

should output a line similar to `Stream #0:0: Audio: pcm_s16le` indicates the audio contained is signed 16-bit little-endian audio, which is what we want. If this is not the case, you may use the steps below to convert it.

A.6 Custom .wav files

Using `ffmpeg` (`https://ffmpeg.org/`), it's trivial to create your own audio files for playback. It supports basically any audio (and video) format under the sun as input, and is available on most platforms.

For example, to convert an AAC `.m4a` file to a CD-quality (16-bit Stereo at 44.1kHz) WAV file, simply run:

```
ffmpeg -i "14 - This Silence Is Mine.m4a" -ar 44100 \
        -ac 2 -c:a pcm_s16le "14 - This Silence Is Mine.wav"
```

Similarly, to create a 48kHz 16-bit mono file from a Vorbis or Opus `.ogg` file, run:

```
ffmpeg -i "2 - Song of the Ancients - Devola.ogg" -ar 48000 \
        -ac 1 -c:a pcm_s16le "2 - Song of the Ancients - Devola.wav"
```

By using the provided framework correctly, you should be able to easily support .wav files with differing sample rates and channel counts.

A.7 Quality

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

A.7.1 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.

A.7.2 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.

A.7.3 Ordinary compression

Normal compression algorithms such as DEFLATE (`https://en.wikipedia.org/wiki/Deflate`) generally don't perform that well on audio data. However, you are free to experiment.