Networks Assignment 2:
Audio Streaming Protocol

David Schep

June 2020

1 Introduction

Spotify, YouTube, Mixcloud, Soundcloud, all these website have their own implementation of an audio streaming protocol. Without extremely good implementations these services would be unusable. In this report I will explain how I implemented my own audio streaming protocol, what problems arose and how these problems were solved. I will also explain how I implemented error-detection and data compression.

2 Implementation

For the implementation of the streaming of audio from the server to the client, the UDP/IP protocol was chosen. UDP does not guarantee package arrival or package order. But it does have the advantage of having less overhead than TCP. This is very important for audio streaming as the audio packets do not have to always arrive but they do have to arrive fast and on time.

One of the problems that do arise from using UDP is that if the server sends too many data packets too fast the client cannot keep up anymore. When the server sends a data packet it is stored in the clients recieve buffer until the client processes the packet. When the recieve loop of the client is not fast enough the recieve buffer fills up and the client cannot recieve any more packets. The server cannot tell if a recieve buffer is full so it keeps on sending packets, which are dropped. The solution to this is twofold. First the recieve buffer size of the client is increased from 213 KB to 4.2 MB. This way it can store more packets and the client has more time to process them. Secondly the main recieve loop needs to be sped up. This is accomplished by offloading filling the ALSA buffer to a second thread. This thread handles all the music playing and the main thread

only handles recieve the packets and processing them. These two measures were enough to ensure that the recieve buffer is always processed fast enough.

The client's recieve thread handles the incoming packets. It recieves the packet and stores this in a small buffer. Then first the checksum is checked. After which the data from the packet needs to be copied to the correct place in the samplebuffer. I have chosen to use a single large buffer the size of the entire audio file. This way the audio playing thread does not have to do a lot of communicating with the recieving thread. It also has the advantage of being able to place out of order packets in the right place in the buffer. This implementation means an out of order packet is never dropped. To know where to place the samples, the packet has a field called `sample_number`. This is filled with an `uint32_t` that tells the client which sample the first sample in the data bytes is. Using this it is trivial to figure out where to place the data bytes.

To make sure the audio thread never reads further along the samplebuffer than it is currently filled with samples, a single `uint32_t` is shared among the two threads. The variable stores the highest written sample number by the data recieving thread. The audio thread reads this value when it wants to write more data to the ALSA buffer. It only starts copying a block of samples into the buffer once it knows that all the samples numbers in this block are less than the highest written sample number. Using this method it is plausible that if packets arrive out of order the audio thread will try to write data to the ALSA buffer that has not yet arrived but will very soon. This is however unlikely as the audio thread will always write larger blocks at once. Which means that writing data that has not arrived can only happen if it crosses one of these block boundaries.

2.1 Error-detection

To detect what the quality of the connection is the client tracks how many bytes are dropped during the streaming. Checking the connection quality every time it recieves a packet is very processor intensive and this would slow down the recieve loop so much it is no longer fast enough to process packets in time. This is remedied by only checking the quality every `N` samples. In the implementation `N = ERROR_DETECTION_WAIT_BLOCK` which is currently set to 32896.
On every single loop of the recieve loop the total number of bytes recieved is kept track of. When the error detection starts this total number of bytes is compared with the total number of bytes on the last error detection run. This value is the number of bytes recieved in the last `N` samples. We can calculate how many bytes we would have expected to recieve in the last `N` samples by multiplying `N` by `SAMPLE_SIZE`. Subtracting these values we now know the difference between the expected recieved bytes and the actual recieved bytes. This is the number of dropped bytes in the last `N` samples.

This algorithm is made slightly more complicated by the data compression. To remedy this the compression level is checked before the expected number of bytes recieved is calculated. For every level of compression we know how many bytes we are expected to recieve per sample, which is hardcoded into the error detection algorithm.

Compression level	Compression method	Effective compression
0	No compression	100%
1	Bit reduction	50%
2	Discarding every 4-th frame	75%
3	Bit reduction & Discarding every 4-th frame	37.5%
4	Bit reduction & Discarding every 2,3,4-th frame	12.5%

Table 1: Compression levels

Compression level	Threshold to increase level	Threshold to lower level
0	5000	-
1	15000	0
2	30000	5000
3	40000	15000
4	-	30000

Table 2: Compression thresholds

2.2 Data compression

5 levels in compression are implemented using combinations of 2 methods. The first method is bit reduction. The 2 channels of 16 bit samples are reduced to 2 channels of 8 bits. This is done by simple taking the last 8 bits of the 16 bit value which effectively divides the 16 bits value by 256. When this data arrives at the client the client boosts the 8 bits back to 16 bits and copies this into the data samplebuffer. This method compresses 32 bits per sample down to 16 bits per sample. A reduction of 50%. The second method is downsampling. Here we discard every `n`-th frame. If we discard every 4-th frame we would reduce the data size by 1/4th. Or 25%.
The 5 levels or compression consist of 5 different combinations of these 2 methods as shown in Table 1.

The client wants to know when the connection is dropping more packets so it can tell the server to change its compression level and send less data. To know when the connection quality changes, without reacting every time the connection drops just a few packets, a rolling average of 3 values is used. Every error detection run the rolling average is updated with the new number

of dropped bytes. By using a rolling average there need to be 3 runs in a row of large amounts of bytes dropped before the client reacts. The client only reacts if the average surpasses a threshold variable. These thresholds are stored in an array and are shown in Table 2. The table shows that the thresholds to lower a level are the threshold of it's level - 2. This is to make sure the compression level only drops when the quality of the connection significantly increases. Otherwise the compression level might bounce back and forth between 2 levels if the quality falls right inbetween the thresholds.

3 Conclusion

Implementing an audio streaming protocol is not easy, many hours were lost debugging sockets and wondering why the audio output sounds weird. In the end the result is very good. The server can stream a full 5 minute long Stereo 44.1 Mhz samplerate audio file to the client in just a few seconds. Without any dropped packets. Also it is possible to simulate a bad connection where the server drops and delays random data packets. Using this simulation it is possible to show how well the protocol handles dropped and delayed packets. Not only can the protocol handle this simulation it can also change the level of compression according to the quality of the connection.