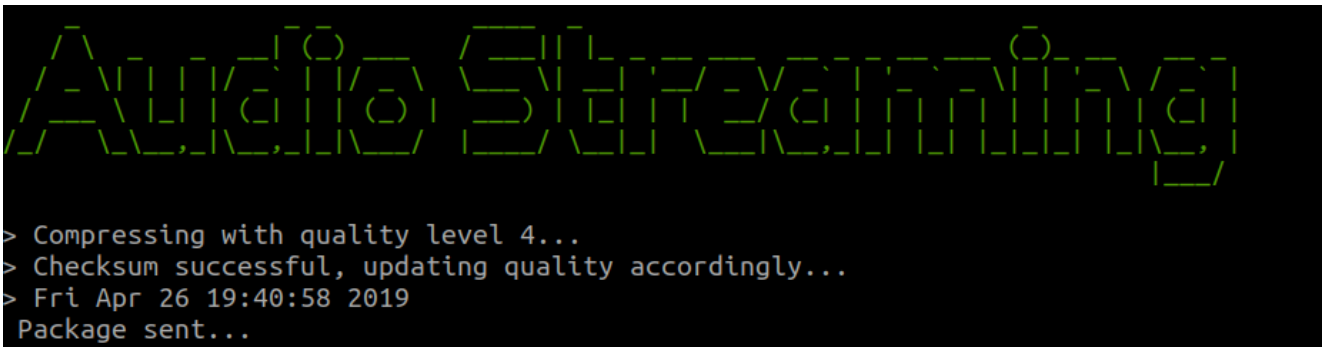
Audio Streaming Protocol

Carlos Bilbao

April 26, 2019

1 Introduction

This is the report of the **Audio Streaming Protocol** on top of TCP/UDP. As such, it can also be seen as a **user manual** and documentation for the executable files.



In here I will give in-detail information on how my protocol works, and how to use it, for both the server and client. The code assumes we are executing both of them in the same machine (i.e. localhost 127.0.0.1) but that could be easily modified with the use of the C macros in the source code. The same can be applied for the port used for the socket connection, which in our case is 2000 but could be any other opened port.

By default the code is set to be in the **simulation mode**, useful for texting the plain connection. This can be changed in the asp.h macro. Anyway, this information is also provided with the option 'h' at the client. Apart from the source code, I also provide a file MUSIC.wav I have used to test the code.

2 HOW TO USE IT?

2.1 Server Program

Sever program establishes a socket in a machine's port and waits for clients to connect in order to send them the corresponding packages. The created socket can be of type TCP or UDP, is just a matter of changing macro **PROTOCOL** at asp.c (STREAM for TCP or DGRAM for UDP).

```
#define PORT 2000
#define PROTOCOL SOCK_STREAM
#define LOCAL_HOST "127.0.0.1"
```

When the simulation mode is disabled, the server program receives as first and only argument the audio file that the server needs to stream. The format for the audio file is expected to be .wav. You can find plenty of waveform files to test in www.wavsource.com.Then it compresses the file and sends it by packages to the clients, and they then compute a checksum, but I will talk about that later. **The server is capable of streamming to multiple clients simultaneously**, thanks to the implementation of the packages and the socket connection system.

2.2 Client Program

The client program support some optional arguments:

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
This is the ASP client.
Options:
You can specify the buffer size with 'b'
You can ask for help with 'h'
The simulation macro is currently ON. Edit the macro at asp.h and
re-compile for music
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

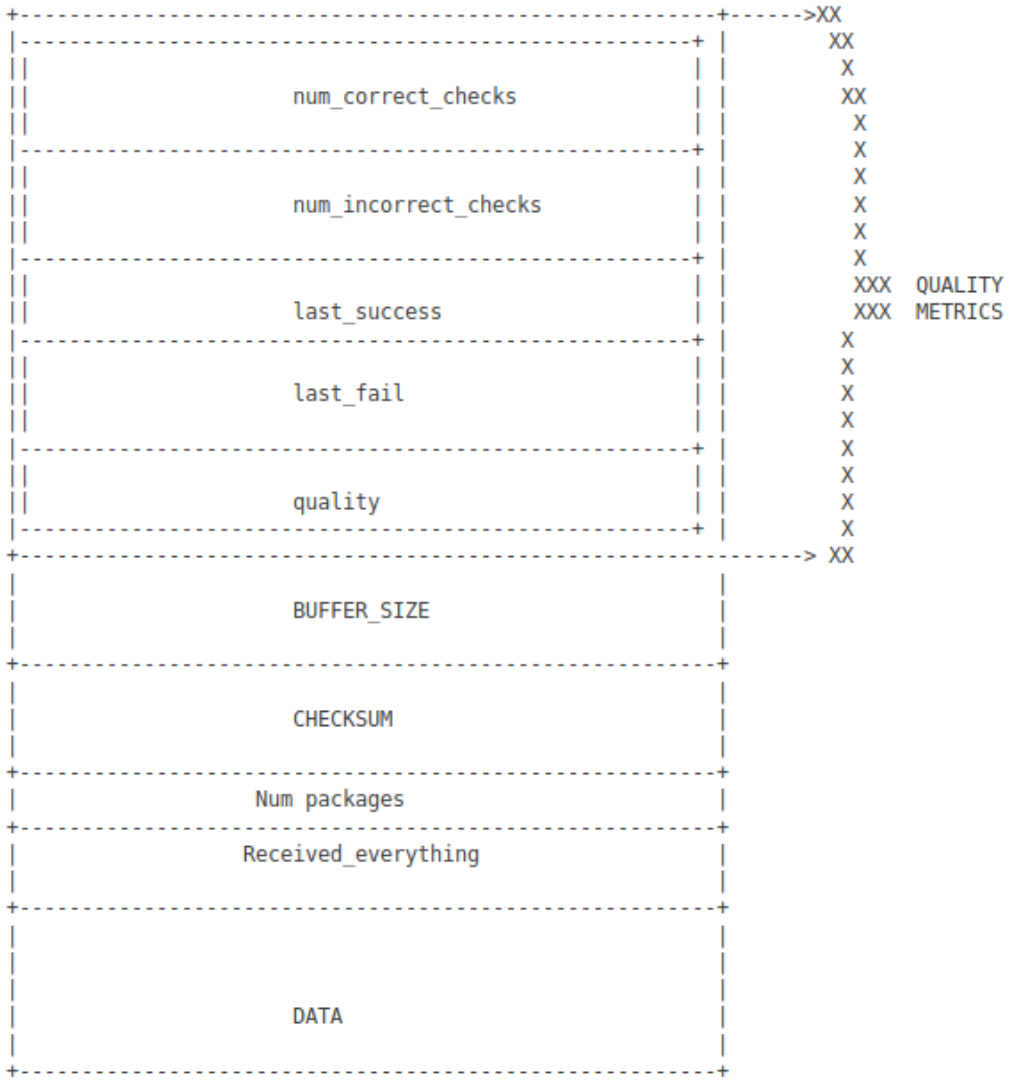
When the simulation mode is disabled, the client program receives the audio by pieces in

the package payload, and opens it using ALSA open-source library.

As the buffer size is specified at the beginning (or not, with the default value), and the song size is known by the server, the client ends the connection when the server flags him that the package received was the last one, with the variable the_package.last_received working as "boolean" (this will be seen in the next section DIAGRAM).

3 PACKAGE DIAGRAM

3.1 ASCII DIAGRAM



The QUALITY METRICS are used to compute the **streamming quality level** (5 quality levels, as explained in the next section QUALITIES). The other sections are the buffer size

(of the payload), a **checksum** used to test the reliability (as later explained), the number of packages already received and the data itself (unit8_t). This package is the only one sent by or received by both parties. The server **does not need to have any record of the clients** thanks to this approach. It simply compute what part of the song should it send next depending on the client's buffer size and the number of packages already received. This design allows **multiple simultaneous clients**.

4 QUALITY LEVELS

```
enum quality_level {EXTREME_LOW, LOW, MEDIUM, HIGH, EXTREME_HIGH};
```

There are five possible quality levels, depending on the reliability of the connection. This value is updated continuously, depending on how many packages have been lost or received successfully, and when both cases occurred last time.

This code is used to update the quality, with the help of the checksum, (which explanation is the next section CHECKSUM):

```
void update_quality(struct package *the_package , struct tm
*current_time){
  if(compute_checksum(MAX_BUFFER, (short unsigned int *)
  the_package->data) == the_package->checksum){
  printf("> Checksum successful, updating quality accordingly\n");
    the_package->num_correct_checks++;
    the_package->last_success = *current_time;

    if (the_package->num_correct_checks >
    the_package->num_incorrect_checks && the_package->quality < 4)
          the_package->quality++;
  }
  else {
     printf("> Checksum fail, updating quality accordingly...\n");
     the_package->num_incorrect_checks++;
```

```
        the_package->last_fail = *current_time;

        if (the_package->num_correct_checks <= 2 *
        the_package->num_incorrect_checks && the_package->quality)
            the_package->quality--;
    }
}
```

```
> Checksum successful, updating quality accordingly...
> Compressing with quality level 5...
> Fri Apr 26 19:41:01 2019
 Package sent...
```

The idea is that, if the connection succeed this time, the quality might improve, depending on how many failures have been and when, and the inverse in the case of a failing checksum computation of the provided data.

5 CHECKSUM AND COMPRESSION

The checksum is a value used to check that the data has traversed without difficulties betwene the client and the server, and viceversa. Here I compute it using the standard formula **RFC-1071**:

```
long compute_checksum(int count, unsigned short *addr){

    register long sum = 0;

    while(count > 1)  {
        sum += * (unsigned short*) addr++;
        count -= 2;
    }

    /*  Add left-over byte, if any */
    if(count)
```

```
        sum += * (unsigned short *) addr;

    /*  Fold 32-bit sum to 16 bits */
    while (sum >> 16)
        sum = (sum & 0xffff) + (sum >> 16);

    return ~sum;
}
```

The checksum is computed after the compression, which depend on the quality, using both **BIT REDUCTION** and **DOWNSAMPLING**:

```
void inline compress(uint8_t *data, enum quality_level quality, int size){

    switch(quality){

      case EXTREME_HIGH:
            downsamp_reduct(data,COMPRESS_LOW,0, size);
          break;
      default:
            downsamp_reduct(data,(quality == EXTREME_LOW || quality == MEDIUM)? COMPRESS_LOW : COMPRESS_HIGH,0, size);
            downsamp_reduct(data,(quality == EXTREME_LOW || quality == LOW)? COMPRESS_LOW : COMPRESS_HIGH,1, size);
        break;
    }
}
```

Depending on the quality it invokes downsamp_reduction with different values, which are used as shown in the next page.
In downsampling, every byte whose position is a multiple of the frequency is not copied. In bit reduction, every byte whose position is a multiple of the frequency is divided by half.If the connection is unreliable, the quality will be decreased and the compression will bigger.

```
void downsamp_reduct(uint8_t *data,int freq,int downsap,int size)
{
            uint8_t aux[MAX_BUFFER];int i = 0, m = 0;
            assert(freq == COMPRESS_HIGH || freq == COMPRESS_LOW);
            memset(aux, SENTINEL , sizeof(aux));
            while(m < size){
                        if(data[m] == SENTINEL) break;
                        if(downsap){
                                    if(m % freq)
                                                aux[i++] = data[m];
```

```
                        }
                        else {
                                if(!(m % freq))
                                        aux[m] = data[m]/2;
                                else
                                        aux[m] = data[m];
                              data[m] = aux[m];
                        }
                m++;
        }
        if(downsap){
                size = i;
                for(i = 0; i < size; ++i)
                        data[i] = aux[i];
        }
}
```

The macros COMPRESS_LOW and COMPRESS_HIGH are defined at asp.h and can be easily modified as desired. You can then recompile with the **provided Makefile**.