Beepmission
Data Transmission using off-the-shelf Audio Hardware

Luc S. de Jonckheere & Elgar R. van der Zande

January 6, 2020

Abstract

In this paper, we provide a technical description of Beepmission. Beepmission is an application to transmit data using inexpensive audio hardware. By utilizing multiple frequency encoding and Go-Back-N ARQ, we achieve a fast, full duplex and errorless connection. We also provide graphical user interface to show the capabilities of our implementation. We built the back end as a separate library, which makes it independent of the GUI. Because of this, the library can easily be adopted to be used in a different setting.

1 Introduction

The goal of this project is to implement an application that can be used to transmit data using inexpensive audio hardware. The application should be able to transmit data through air, but an audio cable may also be used. Because air is an unreliable medium, we also implemented a system that is able to cope with packets that are damaged or lost during transmission.

At the time of writing, practical use cases for this application are absent. The first thing that comes to mind is impromptu file sharing, however, better alternatives are available nowadays (e.g., Bluetooth, QR-codes, Whatsapp file sharing etc.). Not only are these alternatives much faster, they are also more stable and less annoying.

Because our transmission is slow by nature, a educational use case arises. Our application can be used to get a good understanding of how data transmission works. The reason is that the techniques used in our application are similar to the ones used by other communication methods. Because of the speed we transmit data at, it is possible to analyze the data packets either by listening to them or using a microphone and free recording software. Moreover, our user interface provides a number of encoding settings, which allows a user to analyze the effects of changing the window length, encoding frequency, payload size etc.

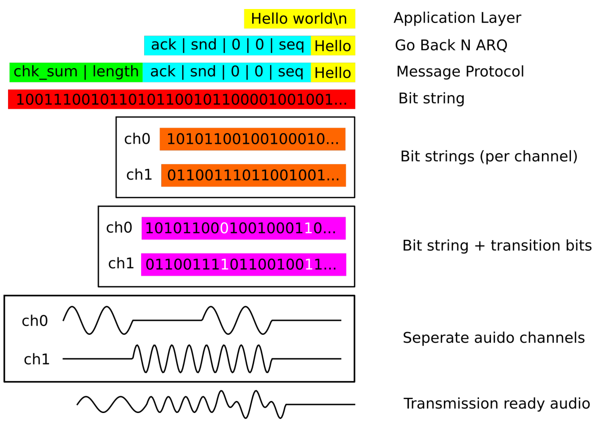

Figure 1: The encoding/decoding stack that our application uses. The different channels indicate different transmit frequencies.

2 Design

There are a number of design criteria we want to meet when developing our application. The transmission scheme is stacked in multiple layers (see Figure 1). Starting from the lowest layer, we implement:

- The audio layer, responsible for encoding a bit stream into audio.
- The message layer, a protocol responsible for adding a header to a payload and converting this to a bit stream.
- Go-Back-N ARQ (sliding window), responsible for errorless full-duplex communication.
- The application layer, which provides a certain interface so humans can use it. In our case, this is the chat application.

2.1 Audio Layer

Figure 1 shows we can encode multiple bit streams into a transmission. This is done by using different carrier frequencies. Before we encode any data, we first append a start sequence. This start sequence consists of a 0 followed by nine 1s and another 0. We use nine 1s, because this can never happen during data transmission (transition bits are inserted at byte boundaries). This start sequence is encoded in all frequencies, because the start sequence is also used for calibrating the receiver threshold value. This threshold determines if a recorded sound is data or noise.

To encode the data we use amplitude modulation. 0 is encoded as no signal (minimum amplitude) and 1 as the carrier frequency with maximum amplitude. Moreover, we have to choose some time period to encode a single bit. We refer to this time as the window length[1]. The effect of this choice is depicted in Figure 2. This figure shows that by increasing the window length, the sensitivity for other frequencies gets smaller.

Using Figure 2, we also conclude that separating frequencies by 20% results in enough signal strength difference to be able to successfully separate different channels. Ideally, we want to choose the frequencies such that they are all orthogonal (i.e., zero points in the graph) with respect to the selected window length. However, we derived this figure numerically and we did not examine the mathematics enough to dynamically (varying window size, different sample rates etc.) derive a set of frequencies at runtime. For this reason, we opted for the much simpler, and in our case sufficient, method of separating the frequencies.

In our implementation, the user can control the base frequency and the number of channels. By setting these values, we generate the following set of frequencies.

$$f_n = f_b \cdot (1 + 0.2 \cdot n) \tag{1}$$

Where n is an integer between 0 and the number of channels (N) and f_b is the base frequency. On top of sending multiple channels at the same time, we also want to support sending and receiving data at the same time (full duplex). In our implementation, the two communicating parties are are called master and slave. To support full duplex communication, they require a disjunct set of frequencies. This is done by separating the frequency set f_n into f_n^m and f_n^s for the master and slave respectively.

[1]We try to be consistent when naming variables in our code, however, name collisions are bound to occur between the different layers of encoding. When we are talking about a *length*, the unit is always in seconds. When we are talking about a *size*, the time period is expressed as a number of samples (hence, the unit is frames).

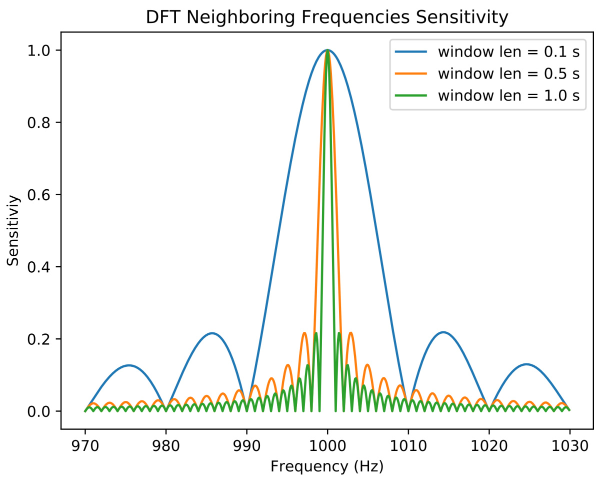

Figure 2: Sensitivity of DFT in neighboring frequencies for different window lengths (δt) at 44.1 kHz sample rate. In this case the measurement frequency is 1000 Hz. From the figure it is clear that if we increase the window length, we can choose frequencies that are closer together. In fact in the limit $\delta t \to \infty$ (i.e., we do a normal FT) the graph changes to a δ-spike.

$$f_n^m = f_{2n} \tag{2}$$

Where n is an integer between 0 and $\lceil N/2 \rceil$.

$$f_n^s = f_{2n+1} \tag{3}$$

Where n is an integer between 0 and $\lfloor N/2 \rfloor$. The equations show that the master gets one frequency more than the slave when an odd number of channels is selected. When this is the case, the master is faster when transmitting.

Our implementation splits a single bit stream into multiple bits stream and encodes this on the different frequencies. Because we can send multiple bits at the same time, we increase the bit rate significantly. We split the bit stream bit-by-bit so that the first bit comes in the first stream, the second in the second etc. We repeat this process until all bits are divided over the bits streams, adding padding if necessary.

Looking at Figure 1, we see that transition bits are introduced in the bit streams. These transition bits serve a number of purposes. Most importantly forcing at least one transition per byte. This extra transition enforces that the start sequence can never occur during the transmission of data (easing the decoding process). Moreover, having multiple transitions is important for decoding because it allows the software to dynamically adjust its sensitivity.

Another practical use of the transition bits is that they can be used to do an extra sanity check on the received data, or to do small one bit corrections. However, at the time of writing, we did not implement any of the features because they are not a strict necessity. How one should approach the implementation of such a feature is discussed in Future Work.

The parity scheme we use for the transition bits is based on even parity. Even parity alone does not satisfy all the criteria we set. When we send eight 0s, this scheme tells us to append an other 0. However, we now have nine 0s following each other without a transition. Because this can only happen in the case that there are eight 0s, we change the scheme slightly to use a 1 in this place as well. The disadvantage is that we cannot detect all one bit errors anymore.

Up to this point we only discussed encoding of the bit stream, however, decoding is equally important. The first problem that arises during decoding is that we need to detect the beginning of a message. We do this by selecting exactly the number of frames that the start sequence takes up (start sequence size times window size) from the record buffer. These frames are then further split up into eleven (the length of the start sequence in bits) equal parts. On each part, we then do a Fourier transform on the lowest receiving frequency. This gives us eleven Fourier components, one for each bit. Because we know our start sequence starts with a 0 followed by a 1, we can use the first two values to determine a threshold that is independent of the background noise level. This threshold is simply the average of the first two values. Finally, using the calculated threshold value, we can convert the array of Fourier coefficients into bits. If this array of bits is equal to the start sequence we can start a decoding attempt.

The search for a start sequence requires Fourier transforms which are quite expensive, so we cannot simply seek through every frame and try to find a start sequence. Because of this, we seek through the frames with a stride of a quarter of the size of a window. If we then find a valid start sequence we attempt to decode the frame. The decoding process can either succeed or fail. If the decoding succeeds, we can start decoding the next message. However, if the decoding fails, we can try again but from a slightly moved position. By increasing the cursor a tenth of the window size, we fine tune until decoding is successful. The advantage of this approach is that if at some point a message is not decodable, we simply move the cursor beyond the start sequence (albeit in steps of a tenth of the window size instead of a quarter), which skips the message.

At the time of writing, we only try to find start sequence if we are sure that there is enough data in the buffer to decode a message of maximal length. This guarantees we can always decode a full message, however, the latency between receiving the sound and delivering the message to the application layer is enormous (especially for short messages). The latency is solved by looking ahead in the packet header to determine if we have already received enough data to decode the message. If this is the case, decode the message without awaiting further data. When we implemented this 'low latency' decoder, we ran into many unforeseen problems, resulting in worse performance then the simple decoding approach. In Future Work, we will describe a theoretical method that uses low latency decoding and mitigates the problems we encountered.

The process of decoding the message is as follows. We once more split the audio data up into chunks of window size and do a Fourier transform for each frequency. Then, we calculate the accompanied bit stream for each frequency. To calculate the bit stream, we keep track of a low and a high value that we use to calculate the threshold value (i.e., their average). Because we know the message data starts with the start sequence, we can initialize the low and high values. We can now iterate over the Fourier components and compare against the threshold value to determine if are receiving a 1 or a 0. After we determine the bit value, we can update the low and high values. This allows us to compensate for varying noise levels during transmission of a message.

Now that we have the bit streams reconstructed, we drop the transition bits and stitch the streams back together into one stream. The importance of separating the bit stream alternatingly during encoding becomes clear. Reconstruction would be impossible if we were to split the bit stream into chunks, because the length of the message is not yet known, so the of a chunk cannot be determined.

2.2 Message layer

In our transmission stack, the message layer is responsible for encoding arbitrary binary data and transforming this into a bit stream. At this layer, we also add the length of the data and we add a checksum (see the green block in Figure 1) so we can verify that a message is unaltered during transmission.

The checksum field is a 16 bit unsigned integer stored in big-endian format. We found that 8 bits is not enough and would regularly yield false positives. The length field is an 8 bit unsigned integer, hence the theoretical max payload size is 255 bytes. However, we limited this to 63 bytes for performance reasons. The two high bits in this field are at the time of writing unused and could be used in the future for sending other information.

Our checksum works on the entire message, including the header. When calculating the checksum during encoding, we initialize the header with the payload length and set the checksum field to 0. We then sum every 16 bits of our data modulo 0x10000 (adding 0s as padding if necessary). We then add a constant to the sum we just calculated, and finally subtract the sum from 0x10000. This value is then substituted into the checksum field of the message. The advantage of this method is that it is symmetric. To verify the checksum we do the same calculations (on a received message) and check that the return value is 0. We add the constant to checksum to prevent triggering on noise that looks like the start sequence of a message with an empty payload. Now that we add a constant to the checksum, an empty message requires a specific checksum value (and not just zeros) solving this problem.

Finally, we need to convert the message into a bit stream. This is done by just interpreting each byte and reordering the most significant bit first.

2.3 Go-Back-N ARQ

We have now established a way to send messages and verify their integrity. However, the communication is far from perfect, because packages can still get lost during transmission. Moreover, sending just messages is not the same as having a full duplex data stream connection (comparable to the differences of UDP and TCP in normal networking). To convert the message protocol to a streaming protocol, we use Go-Back-N ARQ (automatic rerequest).

Our implementation of Go-Back-N, requires one extra byte in the payload (as can be seen in the blue field of Figure 1). The seventh bit (left most) contains the ACK bit, which is set when acknowledging a frame. The sixth bit contains whether the origin of the message is the master or the slave[2]. The lowest four bits represent the SEQ-number, hence the maximal window size is fourteen (this needs to be one less then the maximal sequence number for technical reasons).

2.4 Application Layer

The final piece of the puzzle is the application layer that couples a 'human' to the data stream provided by the Go-Back-N implementation. As discussed in the introduction, the application layer and the rest of the implementation are separated into a chat application (front end) and a library (back end).

We provide a simple chat application that shows how one can use our library. The GUI of the chat application allows for easy experimentation of the encoding/decoding parameters.

3 Implementation

We implemented our design in Python 3.8. We tried to use libraries that are readily available. The GUI is implemented in GTK+ 3 and some calculations use Numpy. The audio library we use is sounddevice, which in its place uses the Portaudio's C bindings. Finally, to use our unit tests, Matplotlib should also be available (however, this is not a requirement for the normal application).

In the `src` directory, a number of files can be found. `audio_stream.py` provides a asynchronous audio stream object that can be used to play and record sound. The file `message_protocol.py` implements the audio layer and the message layer. These layers are implemented in one file, even

[2]This bit field is not required for Go-Back-N ARQ, however, we use it to mitigate receiving our own messages in a echoing environment. Arguably, this field had been better suited in the message layer. However, when we implemented this, we had not yet decided that we wanted to limit the payload size to 63, so there was no space to encode this field.

though the layers can in theory be separated nicely. In practice, some knowledge from different layers is needed at the same time while decoding. For example, to do a bit level parity check, we need to know the length of the payload. However, this length is only not yet known at the audio layer.

The file `sliding_window.py` implements the Go-Back-N ARQ protocol on top of the audio stream message protocol. Finally, `ui.py` and `ui.glade` provide the user interface. The entire application can be started by running the `main.py`.

Our implementation of the Go-Back-N ARQ protocol requires periodic ticking/polling to process its data. This can be a rather slow rate (e.g., 10 Hz) and critical timing is not required. For this reason, the ticks are scheduled in the GTK main loop of the user interface. Processing the raw audio data does require accurate timing, so this process is offloaded to a separate thread in the `AudioStream` class. Likewise, the decoding process also runs in a separate thread because this is an CPU intensive task and would otherwise interfere with the responsiveness of the UI.

In order to make experimenting with the audio/message layer easy, we also provide some rudimentary unit tests. These are located in the `tests` folder.

4 Findings

We experimented with transmission through air and transmission over a cable. When transmitting through air, we found that a base frequency of 1000 Hz works well in combination with the normal base settings we provide. These settings are: 8 channels; window length of 0.1 second; payload size of 12, although this can be increased somewhat and a max sequence number of 3. When using these settings the maximal data bit rate is around 20 bits/s. However this does not take latency/errors into account, so the actual throughput is much lower.

When using a cable to connect two machines together, we achieve a surprisingly high speedup. Because of the high signal to noise ratio, we can increase the base frequency to 2000 Hz, which increases the sensitivity in the Fourier spectrum. This allows us to increase the number of channels to 16, and lower the window length to 0.4 seconds. Finally, because there is virtually no noise on the line, the max payload size can be set to a much higher number (e.g., 31 bytes). With these settings a data bit rate of 120 bits/s is easily achievable.

5 Further Work

There are many improvements that can still be made to speed the connection up or to make it stabler. However, due to time constraints, we were unable to implement these. Below we will quickly discus them and suggest a possible implementation.

5.1 Low Latency Decoding

We briefly discussed that latency is an issue which can be solved by smarter decoding. However, when we implemented an approach that tries to decode a message, even if there may not be enough data available at that time, we ran into issues. The first problem is that the number of messages that we try to decode becomes much larger compared to our current approach. As a result, the application can not keep up with the rate at which we record audio. Another problem is that we trigger on noise and read a header with a large payload size field. However, we cannot discard this message yet because the integrity check can only be done when all data is received. Because we process messages in order, these 'fake' messages fill up the pipeline and caused the receive buffer to grow unbounded.

We managed to get this system partially working by decreasing the max payload size to 63 bytes[3]. However, performance is still worse than the simple approach we initially used.

So, the question remains how we can solve these problems. The answer is simple, reduce the amount of times we decode a message. This can be achieved by introducing an extra buffer to keep track of partial messages. This buffer is filled as follows: in the same way we are now searching for start sequences in the audio stream, we search for the start of message. However, initially, we only require that the header is received. We can then decode the header and read the payload. The start index and the length can then be stored in the buffer. After we

[3]Even though we reverted the changes made, we opted to keep this change because encoding a longer message can take quite a long time (up to 3 minutes of audio data, for 255 bytes).

are done searching for start sequences (i.e., there is no more data in the buffer), we can look at the array and try to decode messages for which we currently have enough data available in the receive buffer.

After a successful decode attempt, we need to update the array such that the indices match the updated receive buffer (because we remove the frames containing the just decoded message). Moreover, any data in front of a successfully decoded message can be removed as long as we decode the messages in order.

The downside of this approach is that we lose the ability to fine tune when receiving a message. However, this can be mitigated by searching for start sequences with a smaller step size (this does raise the CPU intensiveness slightly tough).

5.2 Phase Encoding

It is still possible to increase the throughput significantly by also using phase modulation. However, at this point in time we have not implemented this for a number of reasons. The first reason is that when where testing we noticed that sudden changes in amplitude (i.e., the ones caused by a π phase rotation) produced a loud popping sound when using speakers. Secondly, to be able to detect that a phase change, we always need to send a signal. However, at this point, there is no signal when we are sending a 0. To mitigate this, we can send a low volume signal when sending a 0, however, this lowers the signal to noise ratio.

Our conclusion is that phase encoding is not well suited for through air transmission. Because we focused on air transmission, we opted to leave this transmission method out. It would be interesting to give an option to enable/disable phase encoding because it is a nice feature to experiment with.

5.3 Error Correction

As discussed, we do not do anything with the transition bits at this moment. These transition could be used to narrow the location of an error down to an eight bit window. By combining this knowledge with the checksum, we can try to correct one bit errors. The correction process is as follows: we find the eight bytes where a bit flip occurs and we then flip each bit and check the checksum. If this becomes 0, we have correctly fixed the error.

Whether this method will work in practice is unknown, because the chance that we accept invalid messages is increased. The implementation would also be messy, because this error check breaks the audio/message layer boundary.

5.4 Use of Orthogonal Frequencies

In Figure 2, it can be seen that a smart choice of frequencies yields less crosstalk between different frequencies. These frequencies are called orthogonal frequencies and using these particular frequencies results in a higher signal integrity. However, to use this, we need to be able to derive the frequencies mathematically, because they are dependent on the window length, base frequency etc.

In the same figure, we can also see that separating the frequencies 20% apart is much more than necessary. If we have a good mathematical understanding of the theory, we can fit the channels in a much narrower frequency band. From the figure we can see that at 1000 Hz, we need around a 160 Hz band to fit 16 channels when using a 0.1 second window length. If we manage to achieve this it may become possible to have more than two client in close proximity. Moreover, having a small frequency band makes it also easier to evade certain background noises (e.g., a central heating system).

5.5 Time Domain Start Sequence Detection

The start sequence detection is a CPU intensive task, especially if we want to search in small steps. The reason is that we need to do a expensive Fourier transformations on at least eleven windows. We buffer the sine and cosine functions, however, we still have to iterate a large array multiple times.

A much cheaper check can be done in the time domain. We can simply sum over the bins of window size check if we see a start sequence. If this is the case, we can either still do an expensive Fourier analysis to determine whether we found an actual start sequence, or we can immediately try to decode the message as it will fail decoding if it is invalid because of the checksums.

A Installation on Ubuntu

To run the software on Ubuntu install the following packages:

```
$ sudo apt-get install python3-numpy \
    python3-pip libportaudio-ocaml
```

The sound device library is not available in the standard Ubuntu repositories, however it can be installed using pip.

```
$ pip3 install sounddevice
```

To start the application run:

```
$ ./main.py
```

B Derivation of Figure 2

In Figure 2 we plot the sensitivity $S(f)$ versus the frequency of the 'received' signal frequency (f). The calculations we did are as follows.

$$S(f) = r(f)/\max(r(f)) \tag{4}$$

where:

$$r(f) = \sqrt{a^2(f) + b^2(f)} \tag{5}$$

with a and b the 1000 Hz Fourier component

$$a(f) = \sum_{i=0}^{n} \sin(\frac{2\pi f i}{44100}) \cdot \cos(2\pi \frac{1000i}{44100}) \tag{6}$$

$$b(f) = \sum_{i=0}^{n} \sin(\frac{2\pi f i}{44100}) \cdot \sin(2\pi \frac{1000i}{44100}) \tag{7}$$

Where $n = \delta t \cdot 44100$, the number of frames in a window length.