Members: David Doan and Michael Foiani (Team: jailpt2)

CSCI 1680: Computer Networks

Project: A Basic Ultrasonic Link Layer

Introduction:

Among the possible methods of data transmission, we were the most intrigued by the idea of transmitting data through sound waves. Our goal for this final project was to implement the wireless transmission of data between computers through ultrasonic sound waves that humans cannot hear. We were able to achieve this goal, being able to send data to and from both our own computers, as well as computers within the same room. We achieve this transmission with a reasonable error rate and speed and through implementing this project, we garnered experience of sending data through 1s and 0s at a low level to better understand how a link layer works.

Design/Implementation:

In Python, we used three libraries within our implementation. PyAudio was used to create a channel that allows for the transmission and reading of audio data. SciPy and Numpy were used to create frequencies, handle numeric processing, and compute Fourier transforms for signal processing. Finally, we used MatplotLib to visualize the fourier transform of the sound wave frequencies.

Our utils.py file contains helper functions that assist in the encoding and decoding of data. This includes functions that map our bits to frequencies and functions that create and send the frequencies. In our visualize.py file holds the code that creates a visualizer of the waveform and frequencies that our computer's microphone picks up, which we used for presentation and debugging purposes.

To implement the link layer, we created two classes, a receiver and a sender. Similar to a radio, our receiver and sender classes must share the same initial frequency setting values such that they send and listen to the correct range of frequencies, sharing the same: starting_frequency, frequency_range, sampling_rate, bytes_per_transmit. These values must be shared in order for the receiver and sender to communicate properly.

The sender class starts a thread that hangs on user input, the data we want to send. It then processes the inputted data, converting each data into its 8 bit representation. It then maps the bits to the frequencies that should be sent. Then, through its stream, it sends the data, or in our case plays the frequencies from our speakers.

Similarly, the receiver continuously listens for frequencies found within the agreed upon frequency range. When it hears data within that range, it proceeds to process it. Our receive process uses a protocol we decided to use in order to receive data more accurately. Our protocol to send waves consists of mapping a bit sequence to fundamental frequencies then summing those fundamental frequencies to build a wave.

To map a chunk of data sequence, we take a frequency range and split it into n (= 8 * # of data bytes per transmission + 2 flag bits) sections. The frequencies mapped to the positions that hold “1”s in the bit sequence are given a max amplitude, and the frequencies mapped to the positions of “0”s are given an amplitude of 0. This is how we encode one or more bytes as multiple fundamental frequencies inside of a soundwave. An example of a mapping is in the figure below.

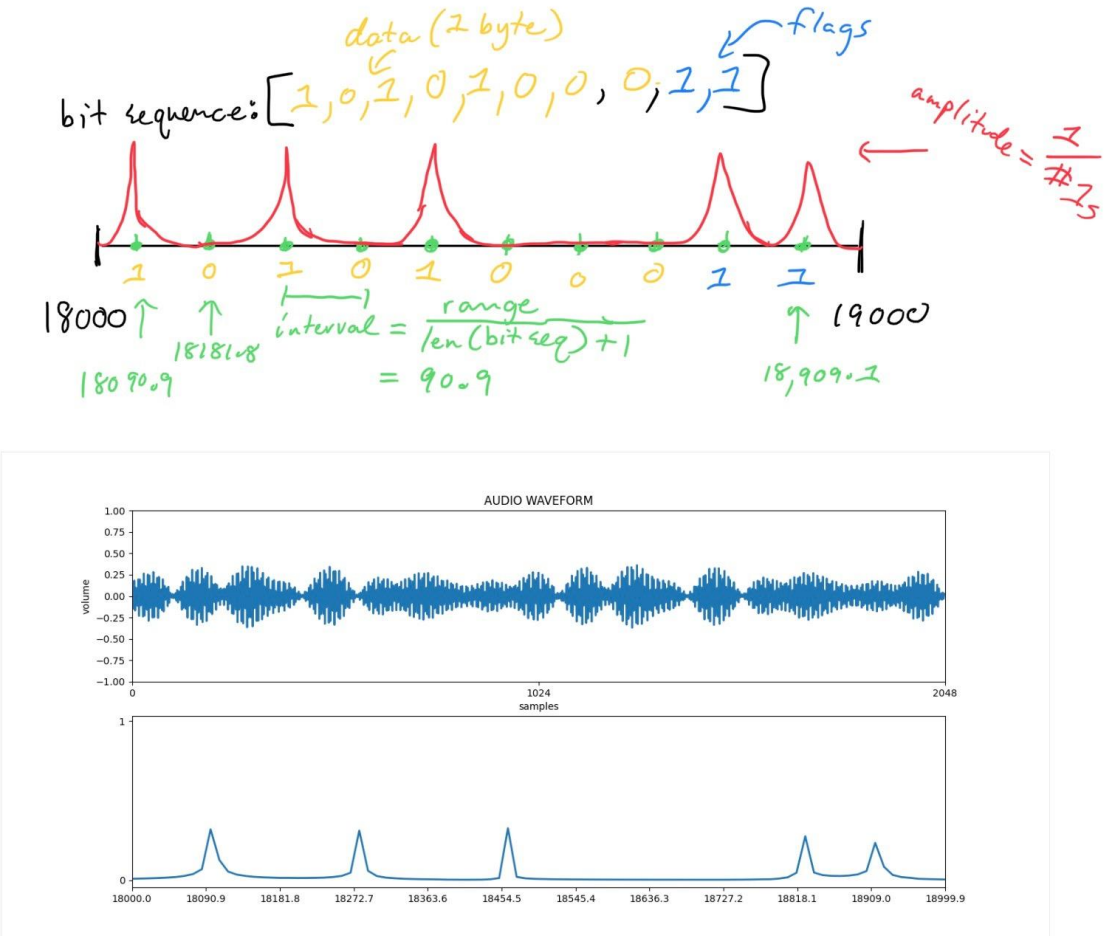

Figure 1: Example of Frequency Mapping the bit sequence 1010100011. The top sketch shows how the bit sequence maps to the fundamental frequencies of a wave. The bottom image of our oscilloscope tool shows the waveform that the sum of those broadcasted fundamental frequencies form.

The first flag assists with an issue that comes with transmitting the same byte back to back, for example “hello”. When transmitting the two “l”s, the data’s bit sequence for that data is the same, so we needed a way to tell the receiver that this same bit sequence actually is a new byte that should be read into the buffer. Therefore, whenever the time to broadcast the next bit in the sequence, this flag is always flipped.

The second flag assists with an issue of knowing if the sender is sending data. If 1, then the receiver should be saving the incoming data. This helps to establish whether a bit sequence of all zeros should be written into the buffer, or if any random ultrasonic wave is actually real data. As a plus, it tells the receiver exactly when the start of a transmission begins and when the last byte of a transmission has been transmitted.

The receiver reads the broadcasted soundwave, applies the fourier transform on it, and then determines the 8 max fundamental frequencies (with amplitude greater than .125) within the frequency range of transmission. These 8 max values are decoded by clamping them to their closest mapped frequency, mapping those frequencies into their resulting bit sequence, then, if the flags are valid, writing the byte data to the buffer.

Discussion/Results:

Demo video link: https://youtu.be/N2Lnv_yb9E8

Along with the demo video above, we ran some experiments to see how the error rate of the transmission changes when we modify the bitrate, frequency range, volume of transmission (with respect to surrounding noises), and distance between the hosts.

Within our experiments, we noticed that instead of seeing a gradual decrease in error rate associated with a gradual change in a parameter, we actually saw a drastic decrease in error rate once a parameter crossed a specified threshold (raw data here). For example, when using a frequency range of 250, we had a low error rate of 98%, but when dipping below 175, the program fell to an error rate of 10%. Similarly, when the bitrate was increased from 6.7 transmissions per second to 10, we went from a perfect transmission to a 50% error rate. Also, if the surrounding noise was equalized to be “louder” than the transmitted ultrasonic wave, then we would get an extremely low error rate, when, if they were equal, we achieved a perfect transmission rate. Unfortunately, we could not get consistent data for distance between the two hosts, but on max volume, we could transmit around 6 feet maximum with very low error rates.

This makes sense to us, given that computers can only be so precise when sending and reading audio waveforms. From this data, we can pinpoint specific “breaking points” in which the transmission begins to fail. Optimal and consistent performance parameters can be achieved by choosing values right before these breaking points.

Given this observation and selecting 250Hz and 6.7 transmissions per second as the optimal parameters, we could use a minimum interval of 250/11 = ~22.72 between each fundamental frequency we encoded as a bit. Therefore, over a frequency range of 2000 (18k-20k), we could theoretically transmit (2000 / 22.72 =~) 88 bits at a time, for a maximum of (88 - 2 (flags)) / 8 = 10 bytes per transmission (of data).

Therefore, in ideal noise conditions, we reach a max bitrate of 10 * 6.7 = 67 bytes per second, applying our protocol on the full available ultrasonic spectrum using these parameters.

We also experimented with bidirectional communication by having each host run both the sender and receiver program concurrently. In a fun way, we viewed the independent sections from the starting and range frequencies as a type of port, where each host needed to be broadcasting or listening on different frequencies. However, this limits the bandwidth in half, so for testing, we stuck with unidirectional data transmission.

Additionally, we ran into issues that we did our best to circumvent. An issue that arose was that sound waves are naturally lossy, which decreases the rate of which we can send data and receive data. This lowers the overall accuracy of reading data from another device, as even though we send the data, it may be lost, requiring the need of retransmission and acks to address this issue. Also, there are device specific issues that can cause popping noises when transmitting ultrasonic waves, corrupting the sound wave that was supposed to be sent over. We tried employing a quick algorithm to clean the signal (safe_byte), but to no avail. More broadly, we realize that communication over ultrasonic waves is not a portable medium among different devices, as it's not standardized.

Conclusions/Future work:

Through the process of implementing and testing our project, we learned a lot about the process of data transmission and better understand the complexities of other methods of transmissions and how they handle some of the pitfalls in sending and receiving data through waves that we came across. From this, we found an appreciation for robust wireless communication by considering the difficult strategies that allow it to be low-latency, high-rated, secure, and lossless. Beyond the necessary conditions to send data well, we can see why we no longer send data through sound waves and use other mediums instead, in favor of a faster and more robust connection.

Relating this project to TCP, and how fast data was being sent, around .003 seconds for a one megabyte file, our method could have its improvements. If we could keep working on this project, we would like to further add optimization to make our data transmission reflect the behavior of current wireless connections, such as data retransmissions, data synchronization, and sending data at higher rates. Nevertheless, we enjoyed and are proud of this project overall, experimenting with something new and fun, while learning more about networks in the process.