CS120 Project Report

Author: Sijie Xu; Email: xusj@shanghaitech.edu.cn

Author: Beiming Zhang; Email: zhangbm@shanghaitech.edu.cn

Project 1

This project basically asks us to implement the physical layer of a network. To be more specific, we are supposed to use speakers and microphones to produce and resceive audios so as to transmit bits. This is demanding because we not only have to design and program the software for our physical layer, we also need to face different issues that come alone with the physical devices.

Our first attempt adopted a unique preamble detection technique with a extended version of PSK modulation. We made the preamble a combination of two notes with specified frequencies, to which FFT will be applied so as to extract it from the signal recorded by the microphone. In terms of modulation and demodulation, we decided that ASK was not practical since the amplitude of the sound will very easily get affected in the air. In order to maximize the baud rate, we express bits with 8 carrier waves with different phases. On top of that, we combined FSK to allow carrier waves with two different frequencies.

However, this attempt didn't work. After multiple times of parameter adjustments, we still couldn't recover the bits sent by the other laptop through sound in the air. Oftentimes the alignment of the preambles failed due to potential external interferences. Possibilities are that our original design had some deficits that caused the failure of our transmission. That said, we may never know the problem in our original design, since later we found out that the speakers and the microphones of our laptops are a mess, to which our failure might be greatly attributed to.

After the failed attempt, we turned to the given `Matlab` code for help. We basically adopted the idea in the `Matlab` code and rewrote our own as the final version of our Project 1 code. We use the chirp signal as the preamble and do dot product calculations to detect it on the fly due to the property that a shift of a single sample will cause the dot product to vary dramatically, which helps in turn gives us precise alignment of the preambles and the data bit signals. The modulation and demodulation are also pretty straightforward. We implement the PSK modulation with `0` represented by a `sin` signal and `1` a `cos` signal. To demodulate the data bit signals, we apply dot products as well to pick the bits that would produce modulation signals that are closer to the recorded ones as the decoded data bits.

It was at this point that we found out it still didn't work. So we began to consider external factors that could possibly influence the performance of our code. We started by plugging in an wired earphone and tried to transmit the bits from the speakers on the earphone to the microphone on it on the same laptop. Then, with this working, we ditched the original version and started working on this version. In order to debug and adjust the parameters, we had to write some scripts to visualize the recorded signal. Finally, this final

version, after great amount of fine-tuning, was able to work between two laptops with the help of an external microphone and a speaker.

One thing I want to say about this project is that it value too much on hardwares. We use various kinds of laptops and hence various kinds of sound cards, speakers and microphones, which inevitably leads to indefinitely many issues with the hardwares. This makes the project tricky already. To add insult to injury, tasks such as the one that challenges the range of the transmission is almost bound to be a lot easier with better external audio devices and much more difficult with shoddy ones or with no external devices at all. I suppose it would better to value less on something that can be easily achieved with a simple hardware upgrade.

Project 2

There's a lot less to do with the hardware and real world environment in this project , since we no longer have to transmit the audio signals through the air. Instead, we can use cables for signal transmission, which is not that prone to external interference.

So, the first thing we do is to redesign the encoding of our data bits. We figured that we can directly encode the data bits as `1` s and `-1` s and send them to the sound card, through which the data bits can be delivered further through the audio cables. The limit of the baud rate would be bound by the Nyquist–Shannon sampling theorem, so we could at minimum encode one data bit as one signal bits where signal frequency equals `fs/2` . However, due to the fact that there's still external factors, such as electrical levels of the laptops and the sound cards' own properties, that can affect the transmission and the reception of the electrical signal, we finally decided to use three signal bits to encode one data bit. This gives us tremendous boost of the efficiency of data transmission. In terms of the preamble, we keeps the idea of the chirp sound, but adapts it to the electrical signal version:
`-1 1 -1 -1 1 1 -1 -1 -1 1 1 1 -1 -1 -1 -1 1 1 1 1 -1 -1 -1 -1 1 1 1 1 -1 -1 -1 1 1 1 -1 -1 1 1 -1 1` . The preamble detection and data demodulation work basically the same as in project 1. As we tried to transmit some random data using this new wired version, it worked out OK.

However, when we were transmitting the data bits in the handout, it failed. We spotted that the error occurred after many consecutive data bits of `1` s or `0` s. This actually caused issues in the transmission since we would end up keeping electrical potential 1 or -1 all the time, which leads to leakage and hardware reset. So, we introduced the 8B10B encoding, which instantly solved this issue.

We also implement subcontracting by including a 4 bytes space for the length of the packet and a flag that indicates whether it's the last packet. So in our design, an physical packet looks like
`| preamble | size | flags | payload | crc |` .

Building on top of the solid data transmission, the other related jobs, like ACKs are just easy to implement.

Project 3

Now that we have a working physical and MAC layer, we figured that we only had to deal with software issues. But, in reality, we were wrong. The hardware issues were far from gone. Every time we connected the laptops with the mixer and plugged in or not plugged in the power, the transmittion would very possibly fail due to varing voltage issues, so we had to adjust the parameters and the shreshold of preamble detection and recompile the whole program again and again before we can move on to implement project 3. To make things better, we let our program read the configurations when it start running just so we don't have to compile it to make parameter modifications. If we had more time, we would definitely work on self-adaptive mechanics that can determine the shreshold of preabmle detection automatically on the fly to make the system more robust.

We implemented the virtual network device in the first place to save our time doing the tasks the hard way. When implementing the forwarding, we faced a new issue on the software side. With more layers in the data transmission, the program ran much slower. And since the task of processing input signal and calculating output signal is performed in the `audioDeviceIOCallback` of the sound card which runs in one thread, the heavier task potentially causing the thread to be block from triggering the next `audioDeviceIOCallback`, and some signals collected in the buffer of the sound card started to be missed out. The key is to prevent the `audioDeviceIOCallback` being blocked. So, once an input arrives, we copy it to a stream buffer to be processed later by another thread. And we took a huge step up and introduced asynchronous semantics with `boost::asio` to our program. It came in handy and the physical layer and the layers above could run asynchronously, which solved the issue perfectly.

Then, we simply have to use `Npcap` to capture the packets that a node receives, and then compose a packet following the IPV4 standard and also the other way around. But when we connected all the cables and starts sending `ping` packets, the channel was a mess. It turned out that loops occurred when node A forwarded everything it received, including the packets sent by itself. So, we implement forwarding tables to both directions of the channel and everything seemed good.

Except that the `ping` packets never get replied. We then spent hours trying to find out what exactly we did wrong. We compared our packets with the normal `ping` packets that the system sent out in Wireshark. We nearly modified everything but still couldn't see what the problem was. We even predict the sequence number of the system's next ICMP packet and sent the exact same ICMP packets that the system `ping` command produced in Wireshark. And the weird thing was that the system `ping` was replied whilst ours was not. After hours of hard work, we conducted an experiment which compares the ICMP request packet that Wireshark captured at sender's laptop with the ICMP request packet that Wireshark captured at receiver‘s laptop through which we finally made a discovery.

We were surprised to find that although the IP checksum of ICMP request packet generated by the system `ping` command at sender's laptop is zero, the ICMP request packet capture by Wireshark at receiver's laptop was calculated to the correct value. So we were misled by Wireshark at sender's laptop all the time! The packets shown in Wireshark are not necessarily the final version of the packets that really went out to the internet through our network interface card. We thought the system `ping` didn't fill in

the IP checksum as shown in Wireshark and did so, however, the IP checksum offload technique allows NIC to calculate the IP checksum instead of CPU and the filled in the `checksum` field before finally went out to the internet, while Wireshark only captured the packet before NIC's calculation. Once we filled in the `checksum` field, which was intentionally left blank, we immediately got the replies. Then, all the work followed was straightforward because the virtual network device does most of the job for us.

Project 4

In this project we are required to implement DNS and HTTP, which corresponds to the capability of forwarding UDP and TCP packets. So all we need to do is to add support to UDP forwarding and TCP forwarding on the basis of Project3 and config the route table of node1 to let DNS server IP and the target IP go through our virtual net interface. Then all the tasks are completed. However, after a double check of the requirement of task2, we found that it requires the TCP sequence num to be chosen from a specific set. And we cannot use the system's `curl` command to generate the TCP packets, instead, we have to manually generate the SYN packet and reply ACK to certain packets. One problem we met is that out SYN packet never get replied. And after consulting for some information, we found that the system won't capture the replied TCP packet unless the port is used by some process. So we write a simple python script to occupy that port and run at background, then start our curl program which first sends a SYN packet by `Npcap` . And we successfully captured a SYN/ACK reply and make the following transmissions. Everything went well then.