CS120: Computer Network Project Report

田鲁岳 2022533065
Team member
Shanghaitech University
Shanghai, China
tianly2022@shanghaitech.edu.cn

陶潘鑫 2022533112
Team member
Shanghaitech University
Shanghai, China
taopx2022@shanghaitech.edu.cn

***Abstract*—In this four-phase project for CS120, we implement a toy network `Aether.NET` with acoustic signal to transmit information through sound card. These four phase can be roughly summarized as the Physical layer, the MAC layer, the IP layer and the Application layer. We employ C# with `NAudio` and bunch of other libraries to complete the whole project. We chose C# since it can balance the low level control and high level abstraction. We also use `WinTUN` to create a virtual network interface to simulate the network environment. The project is a good opportunity for us to learn about the network protocol, and we have gained a lot from it.**

I. CS120 Project Report

A. Project 0&1: Acoustic Link

Project 0 offers us a chance to learn about how to code with the sound card. Project 1 is the footstone of the `Aether.NET` project, we implement basic modulation and demodulation framework, and work flow like preamble detection, packet encode/decode and file read/write, etc.

a) *Audio API:*

We utilize WASAPI instead of ASIO as provided in this two project, because WASAPI is built into Windows, providing native support for audio without requiring additional drivers, making it more accessible than ASIO. And WASAPI supports connecting two sound cards at the same time, allowing us to debug on one computer. WASAPI has the disadvantage of not being able to freely adjust the buffer size, but project1 has no requirements for latency, so we choose WASAPI. We will discuss the implementation of the physical layer in the following order.

b) *Modulation and Demodulation:*

We implement the modulation with PSK as provided in the example matlab code. PSK is more suitable for this task because the open environment will lead to unnecessary and thorny challenges to handle the noise and amplitude for FSK and ASK.

As for PSK, we should handle the problem of aligning with preamble signal. We use chirp signal as preamble and implement an async task of demodulator to monitor the preamble. The demodulator needs to maintain a small buffer to store the received sample and calculate whether it is a preamble by dot-multiplying and comparing with the threshold. If the product is over the threshold, this moment is marked as the starting point of the data. For demodulating of the data, we calculate the dot product with the symbol of 0, and if the result is over 0, then it is 0, otherwise 1.

It is worth mentioning that, in order to avoid potential synchronization problems, our modem as well as the leading code detection procedure is implemented to be completely stateless, and its combination with `ReadOnlySequence<byte>` greatly improves the readability of our code.

We fix the length of the payload in the packet and pad zeros, and add a length field to the package to remove the padded 0′s.

c) *Error Correction:*

FEC code (we use RS code) is added to check whether there is any error in the demodulated signal. However, it turns out that error-correcting codes are not decisive in open environments, and additional error-correcting codes increase the number of BERs, so we added only a small amount of error-correcting codes to serve as a checksum and expect to correct a very small number of errors. Each packet contains 32 bytes payload and 5 bytes ECC, and so it allows the received packet contain 2 errors.

d) *Higher Bandwidth:*

We use OFDM to improve the bandwidth of our toy network, which utilizes two orthogonal carriers overlapping without interfering with each other. This allows for high spectral efficiency and resistance to interference, making it ideal for high-speed data transmission.

OFDM is based on transmitting with multiple frequencies at the same time, so that the signals of each frequency are orthogonal, which means that the integral value within the length of time of each symbol should be 0. The plainest way to realize such is to determine a base frequency, multiply this frequency by 2, and multiply the number of periods by 2 as well, so that the integral value within each of our symbols is 0, which also fulfills the requirements of OFDM.

As for why we chose only two frequencies, it is because we found that different frequency sound signals propagate differently in the air, only the middle frequency sound is better,

and high frequency sound is damaging to the human ear, so we set the frequencies at 4000Hz as well as 8000Hz.

e) *Challenges and Suggestions:*

Since this project utilizes external sound signal to transmit, the performance is subjected to the device and environment to a certain extend. A good speaker and microphone can help, while the built-in one in laptop behaves terrible. A quiet and open environment is required to determine the parameters such as frequency and threshold, and the narrow dormitory is not a good choice.

We also found that longer signals still produce offsets after alignment, so we set the packet length to 32 bytes, allowing us to achieve a 100% transmission success rate.

B. Project 2: Manage Multiple Access

In this project, we implement the MAC layer. To be more specific, it mainly involves the sliding window algorithm and the CSMA state machine. Wired connection through USB sound card and audio cables. For the sake of performance, we switch to ASIO API instead of WASAPI, which provide more flexible access to twisting the buffer.

a) *Base Band Transmission:*

We started with a direct square wave signal for transmission, but the number of sample points per symbol could never be less than 3. After recording and observing in Matlab, we found that the waveforms were not perfect, probably due to the fact that the digital to analog conversion was not perfect, so we considered changing the sample points for modulation to a sawtooth waveform, with each symbol starting at 0 to avoid overshoots, and we did get a much better waveform, but due to the fact that the ASIO has from time to time an offsets, which led to errors. We ended up going with a square wave, but turning the volume down amazingly made the signal quality better, and also managed to reduce the number of sample points per symbol to two.

b) *Sliding Window Algorithm:*

The sliding window algorithm is designed to manage the reliable transmission of data packets between two endpoints using a window-based flow control mechanism, and is nothing new than the one in class. A sequence number field is added to each packet, and note that the bits for sequence number field should cover the send window and receive window to avoid overlap.

We modeled the TCP packet ACK mechanism and designed the sliding window algorithm. In our implementation, ACK indicates that all packets before this number have been received, which can effectively prevent ACK packets from being lost or duplicated leading to state confusion. We have both seq and ACK fields in the packet header, which can effectively prevent a large number of retransmissions caused by the loss of ACK packets when sending to each other.

When there is a packet to be sent, a sequence number is first taken out of the queue, and this operation accomplishes both blocking and sequence number assignment. The packet is then written to the PHY and sent, and when it is finished, a Task is requested for the synchronization primitive, which is returned when the ACK is received in another thread. Next, this task waits with a timeout, if it receives the sequence number it adds it back to the queue, if it times out it retransmits it, and after each timeout or ACK is received the RTT estimation is updated to dynamically adapt to the state of the network.

```
func send(packet p) {
   seq = seq_queue.dequeue()
   p.seq = seq
   task = sync.wait(seq)
   while (true) {
      phy.send(p)
      if task.wait(timeout) {
         seq_queue.enqueue(seq)
         update_rtt_estimate()
         break
      } else {
         rtt_estimate += 10ms
      }
   }
}

func receive(packet p) {
   if ack_in_window(p.ACK) {
      acks = ack_in_window_before(p.ACK)
      sync.signal(acks)
      seq_queue.enqueue(acks)
      update_window(p.ACK)
   }
}
```

Although we successfully implement the sliding window algorithm, we set the window size as 1 which makes it equivalent to a stop-and-wait mechanism due to the low-bandwidth, and such a frame-by-frame acknowledgment can better service the integrity of data transmission.

c) *CSMA:*

We implement the CSMA protocol by sensing the communication medium before attempting to transmit data. If the medium is detected to be idle, the transmission proceeds; otherwise, the transmission is deferred to avoid collisions. We incorporates a mechanism to detect the presence of a carrier signal, which indicates whether the medium is currently in use. This is achieved through a carrier sensing component that continuously monitors the medium's energy levels. If the energy level exceeds a certain threshold, it indicates that the medium is busy, and the transmission is postponed. This helps in minimizing the chances of data collisions.

In addition to carrier sensing, our CSMA design does not have a fallback mechanism, but is designed with MAC, our MAC uses a timeout value based on the estimated RTT time, when the MAC layer has a timeout it will pair increase the estimated RTT (10ms), and when a new ACK is subjected to the estimation, we use the sliding average algorithm to esti-

mate it. This design allows our connection to be transmitted adaptively.

d) *Challenges and Suggestions:*

This project must be the most time-consuming one due to the dependence of the devices. We encounter following problems.

The actual rate of the USB port may not be consistent with the one you set in the Windows setting, so try print the rate before debugging. The reason for this problem was that we were using WASAPI and couldn't set the frequency manually in the code, we solved this problem by switching to ASIO.

At the same time, we incorporate a warmup preamble to alleviate the influence of the device start up. This problem is related to the topology of the sound, and only occurs when two sound cards are directly connected, whereas a topology with a mixer added in the middle does not, but we have nevertheless added a warm-up precursor to our code to avoid possible problems.

One tricky problem also puzzles us a lot, that is we don't know whether the audio API is blocked or not, so it's hard to set the backoff time. Chances are that the timeout happens before the packet is retrieved from the buffer and actually sent. In this case, the sender endpoint will keep plugging the same packet to the sender buffer multiple times, no matter whether it's received successfully or not, which waste a lot of time. The reason for this problem is because we were previously buffering our packets all through a queue, which proved to only make sense on reliable channels, and we switched from a queue model to a blocking model based on individual packets by refactoring our code, allowing us to solve the problem.

During this time we were also confused by the quality of the signal transmission, we found out that it was the volume, too much volume would cause distortion of the signal, too little would cause the preamble not to be detected, so we added a loudness parameter so that we don't need to debug the volume level of the mixer and the computer.

All the parameters for project 1&2 are given in the very front of Program.cs for easy finetuning and reference.

In addition, we have designed the packing and unpacking process of packets, we use `ReadOnlySequence<byte>` to store the packets so that we can easily slice and concatenate the packets without copying the data, which reduces the use of memory and improves the efficiency.

C. Project 3: To the Internet

This project requires us to implement some IP layer protocols on the top of the previous physical and MAC layer, including ICMP echo, router and NAT. In the end of this project, our toy network can be integrated into a broader network, leveraging IP layer for effective inter-network communication.

a) *ICMP Echo:*

For ICMP packets, we just need to filter the IP packets with protocol ICMP through the virtual interface, and the system will generate the corresponding reply or forward it for us. On windows, we also need to turn off the firewall.

b) *Router:*

Since we are using a virtual interface, we didn't write the routing code, instead we let windows use the routing feature that comes with windows, so that we can maximize the reuse of the existing features. We used `Set-NetIpInterface -Forwarding Enabled` to enable routing for both the virtual interface and the wireless interface, thus completing the Star task. In other projects, packet forwarding is done internally by NAT and no additional settings are needed.

c) *NAT:*

We use the NAT function that comes with windows to realize NAT, on NODE2, we execute `New-NetNat -Name -InternalIPInterfaceAddressPrefix` to set it up, as long as the internal IP is set as a subnet, then the NAT operation can be performed automatically, however, this NAT comes with some limitations, for example, there is no way to set up a static NAT for the ICMP packet, so we didn't finish the relevant Task.

d) *Challenges and Suggestions:*

Since this project, the influence of the physical device becomes minor, and it's a good news that we don't need to take too much effort to twist with the unreliable devices and tune the parameters. The challenges lies in ensuring the effective implementation of the MAC layer and understanding the principle of these IP layer protocols.

Since the WinTUN we are using is a Layer 3 device that transmits Raw IP packets, and Arp works on top of Layer 2, there is no way for us to implement the standard ARP protocol and have it captured by Wireshark, so we did not complete the related tasks

D. Project 4: Above IP

We dive into the application layer in this project, and revisit the IP layer implementation to support DNS and HTTP. Our main job is to filter the packets in the virtual interface and set up the host correctly, mainly by setting the virtual interface IP and mtu.

a) *DNS:*

The DNS protocol is implemented to resolve domain names to IP addresses. Since we are using a virtual interface, we don't need to implement DNS lookups, only filtering.

We use the `ARSoft.Tools.Net.Dns` library to parse DNS packets for filtering, we designed a heuristic whitelist filtering. We first check if the port of the udp packet contains port 53 as port 53 is the default DNS service port, then we parse the payload as a DNS packet and check that the query is of type A as we only support ipv4, then we check that the if the query is the domain name we set (`example.com`).

Then we directly transmit the whole IP packet as payload over sound and send it out from the virtual interface at the other end and the OS takes over the subsequent processing.

b) *HTTP:*

In the implementation of http, we also mainly perform packet filtering, filtering TCP packets is simpler, only accepting those packets whose IP is set for us (`example.com` IP address).

In Task2, we need to modify the sequence number of the TCP packets, since we are using a virtual interface, we don't implement a TCP stack, instead we hijack the TCP packets, on NODE1, when the SYN packet is sent out, we record the original sequence number, and then modify it to the number that we have set, so that we only need to subtract the subsequent sequence number from the original sequence number and subtract the returned ACK from the sequence number we set, the relative sequence number and ACK can be calculated, and the correct TCP connection can be restored. And no additional operations are needed on NODE2.

c) *Challenges and Suggestions:*

We did not encounter any major difficulties in this project, because the previous project has provided us with a more reliable foundation. Our main time is to set up relevant network settings and further optimize the sound connection to improve the stability of the network.

II. Lib & Reference

`ARSoft.Tools.Net`
Offering us the ability to parse DNS packets.

`AuroraLib.Core`
Provide an `Int24` implementation to receiving PCM24 samples.

`CommunityToolkit.HighPerformance`
Provide useful methods to do span operations and casts.

`DotNext` & `DotNext.Threading`
Provide bunch of async prmitives and useful methods.

`MathNet.Numerics.Data.Matlab`
Provide a matlab lib to export test data to matlab for debugging.

`NAudio`
Provide audio API, and audio utilities.

`Nerdbank.Streams`
A mutable version of `ReadOnlySequence<byte>`, allow us build stream and packet based on it.

`NWaves`
Provide wave builders like sin and chirp.

`PacketDotNet`
Provide network packet lib to parse and manage network packets.

`STH1123.ReedSolomon` A Reed-Solomon error correction code implementation.

`System.CommandLine`
Provide a command line parser to parse the command line arguments.

`System.IO.Pipelines`
Provide a high performance buffer management lib, useful for buffering audio samples.

`Unofficial.WinTun`
A WinTUN wrapper to create virtual network interface.

`profetia/rathernet`
A reference implementation of the project 1&2 written in rust, our main gain was to design a stream and packet based hierarchical structure, which allowed us to design the MAC, CSMA and PHY separately, greatly simplifying the development burden.

`Nyovelt/Native-Modem`
A reference implementation of the project written in C#, our main takeaways were a blocking send interface for the audio API, which allowed us to better control the sample stream, and they also inspired a volume adjustment that became the basis for passing the tasks, even though it wasn't caught in the first place. We also find that they use same report template in **typst**