Athernet: A Toy Network

Yifan Cao Weitian Wang
ShanghaiTech University
{caoyf, wangwt}@shanghaitech.edu.cn

I. Introduction

As the project for CS120, We implement Athernet, a toy net that uses acoustic signal or sound to transmit information through the audio connection. We implemented the physical layer and MAC layer of the network. We also implemented a NAT for Athernet to make it communicate with internet and an FTP client for Athernet nodes on UDP protocol so we can download files for the off-net Athernet node via the audio connection. In this project, we use C++ for Athernet implementation and use python to implement part of NAT for Internet transmission.

II. Project 1: Acoustic Connection

In this project, we work on physical layer of Athernet. we choose ASIO, a computer sound card driver protocol for digital audio that provides a low-latency and high fidelity interface between a software application and a computer's sound card to generate and receive sound wave, and develop our project based on JUCE, a C++ application framework which provides powerful support for audio transmission. We encode the transfering bitstream as samples of sound wave by PSK method, packed with header wave as a physical layer frame. As a result, we successfully transmit 10000 bits by our own microphone and speaker in 15 seconds with acuuracy of 100 percent. However, the transmission is fragile and susceptible to noise from surrouding.

A. Modulation

We choose PSK to modulate bitstream. In original design (without OFDM), we use the wave $1\times cos(2\pi\times 5000x+\pi)$ to represent 1, and $1\times cos(2\pi\times 5000x) = -cos(2\pi\times 5000x)$ to represent 0. We choose the sample rate of 48000Hz, and by fully test the harwares, we find that 400 bits per frame and 48 samples per bit best fit our devices, giving consideration of both accuracy and bandwidth.

As for header wave, it is also a sin wave that start with frequency 200Hz, and continuously increases its frequency until 10000Hz, and after that, its frequency decreses at the same speed to 2000Hz, which is shown in Fig.1

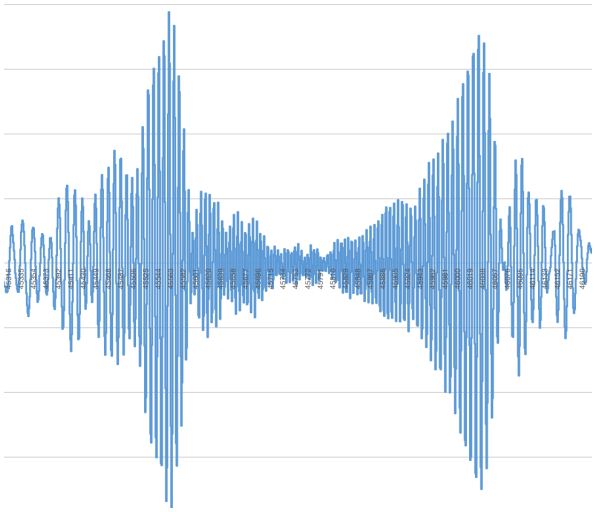

Fig. 1. header wave

B. Demodulation

We use an algorithm to demodulate the audio samples to bits. There are two states: synchronizing and decoding.

1) Synchronization: In the synchronizing state, the program is reading the samples and finds if a header arrives. If it ensures that it hears a header, it switches to the decoding state with the accurate start of this frame recorded.

When synchronizing, we are looking at a number of samples, which is as many as that of a header. We expect to decide how much could these samples fit the header. A variable syncPower is used to estimate the goodness of fit. Given header H and samples S with length N, this value is defined as

$$\text{syncPower} = \sum_{i=1}^{N} H_i \cdot S_i.$$

The larger syncPower is, the better the samples fit the header. We set a threshold value that syncPower must

exceed if the samples are considered as a header. We use a variable called maxSyncPower which records the maximum syncPower value that exceeds the threshold in the near past. If maxSyncPower has not been changed for a period of time, and the current value of it does exceed the threshold, we conclude that the time maxSyncPower appears is exactly the accurate start of the packet. Here we need to wait for a period of time, in case the best fit is used.

To be stricter, we apply an additional strategy to the synchronization algorithm. When we arrive at the header at some time, the syncPower should be obviously large than that of even 1 sample deviation. Taking advantage of this feature, we allow the update of maxSyncPower only if the syncPower exceeds a fixed multiple of power, which roughly estimates the power of the recent samples. Initially it is 0, and it is updated according to the equation

$$\text{power}_i = \frac{63}{64} \cdot \text{power}_{i-1} + \frac{S_i^2}{64},$$

where i is the sample number and S_i is the sample value. In this equation, the most recent sample S_i is merged into power with ratio $1/64$, while the previous power is reduced to $63/64$ of the original, since it represents some older values.

2) Decoding: In the decoding state, the program has already found the start of this frame. If a bit is represented by s samples, and this frame includes n bits, the number of samples we should receive after the header is $n \cdot s$. We split the samples into groups of length s, where each group contains samples that represent a common bit. These samples should roughly have the form

$$s(t) = \begin{cases} A\sin(2\pi f_c t), & \text{binary 1,} \\ A\sin(2\pi f_c t + \pi), & \text{binary 0,} \end{cases}$$

which is similar to those the sender generates. To decode the samples, multiply their values with $A\sin(2\pi f_c t)$. The form then becomes

$$s(t) = \begin{cases} A\sin^2(2\pi f_c t), & \text{binary 1,} \\ -A\sin^2(2\pi f_c t), & \text{binary 0.} \end{cases}$$

In other words, if the bit is 1, the values should chiefly be positive, otherwise negative. We sum these values up, and conclude the bit is 1 if the sum is positive, and 0 otherwise.

C. Higher bandwidth: OFDM

OFDM is a method to encode bits on multiple carrier frequencies. 'O' is orthogonal, and if we use two or more sin waves to transmit simultaneously, the superimposed wave is able to carry information from all channels, as long as they are orthogonal. A simplest orthogonal case is $cos(x)$ and $cos(2x)$, and we can modulate two channels information in the form of

$$A(\text{bit}_1 \times 2 - 1)cos(fx) + B(\text{bit}_2 \times 2 - 1)cos(2fx)$$

and in this case we can trnasmit 2 bits at once. In other words, the bandwidth doubles.

In demodulate stage, we multiply the received waves with two carrier waves, and therefore decode two coreespoding bits. In practice, we adopt a more reliable approach for the case of two frequencies. Firstly, 4 carrier waves are generated at the very beginning, they are:

$$\begin{cases} A(\text{bit}_1 \times 2 - 1)cos(fx + \pi) + B(\text{bit}_2 \times 2 - 1)cos(2fx + \pi) \ldots 0 \\ A(\text{bit}_1 \times 2 - 1)cos(fx + \pi) + B(\text{bit}_2 \times 2 - 1)cos(2fx) \ldots 1 \\ A(\text{bit}_1 \times 2 - 1)cos(fx) + B(\text{bit}_2 \times 2 - 1)cos(2fx + \pi) \ldots 2 \\ A(\text{bit}_1 \times 2 - 1)cos(fx) + B(\text{bit}_2 \times 2 - 1)cos(2fx) \ldots 3 \end{cases}$$

and each of them is corresponding to a digit of 0,1,2,3. Then, we multiply each carrier wave with received wave respectively, track the carrier wave with the largest product, and decode the two bits as the corresponding digits of the waves.

III. Project 2: Multiple Access

In project two, our hardware devices change from commercial speakers and microphones to sound cards and calbles, which allows us to transmit in baseband and use a few samples to encode a bit (we choose 3 samples per bit), and we mainly work on MAC layer of Athernet. We redesign the frame header, adding src_addr, dst_addr, sequence number, type, length fields, as well as a CRC checksum. For Athernet nodes, we establish a state-machine for each node, using multithreding to archive transmitting and receiving at same time, and also ACK and retrasmission mechanism. We also design amd support MacPing and MacPerf command to test the latency and bandwith of Athernet. Finally, to avoid conflict, we implement CSMA to detect the channel and backoff when transmission detected.

A. The Packet Structure

In the packet, we reserve some bits to store necessary fields.

- The first 8 bits are type filed, i.e. they type of this packet, such as data frame, MacPing(request, reply), MacPerf, ACK...
- The second 8 bits are sequence number field, which is essential for ACK and retrasmission.
- The thrid 8 bits are dst_addr field.
- The fourth 8 bits are src_addr field.
- The fifth 8 bits are length filed, which tells the length of the frame's data.

B. ACK and retrasmission

ACK and retransmission are fundamental to archive a reliable transmission. Due to the fragileness of hardware trasmission, packets in Athernet suffer from a series of failure, like lost in cable and reversal for bits. Take data packets for example, when Athernet nodes receive a data packet with sequence number K and its CRC checkusm is 'good', the receiver node will reply an ACK packet with sequence number K, and when the trasmitting node receive the ACK of this packet, the transmission of packet K is treated as successfully, else retransmission is needed. Trasmission occurs in four situations,

- The data packet is lost, i.e. the receiver does not receive packet K.
- some bits are changed during the transmission, i.e. the CRC check does not pass.
- The ACK packet is lost.
- Both data packet and ACK packet are transmitted correctly, but timeout occurs.

In transmitting node, we either maintain a thread pool that keeps the duration of all current transmitting packets using std::condition_variable.wait_until method in C++, or just iteratively check the duration in sending thread. When timeout occurs, the transmitting thread abort the previous transmission and retransmit the abrted packet.

For data transmission in project 2, we found that stop-and-wait strategy does not meet the requirement in performance, so for the relatively small data file, we just transmit all packets at beginnning, and after receiving ACK packets, it continuouly transmit the rest packets without ACK replies, until all packets are successfully transmitted.

C. CRC

To ensure the correctness of transmission, we introduce CRC check. We choose CRC8 with CRC polynomials

$$x^8 + x^2 + x + 1 \ldots 0\text{x}07$$

We compute the CRC checksum when constructing data packets, appending the result in the end of packets, and in receiver node, we recompute the CRC checksum with same algorithm and compare with the appended one to do the corectness check.

D. The macperf Utility

We implementation perf packet in MAC layer, which is a utility to measure the throughput of Athernet. We design the MACperf packet with almost same header as data packet(no length field), and tramit 1000 random bits as payload for each MACperf packet. The MACperf controller sends packets with best effort, while counting the ACK received for each second, and print the measured throughput on the screen.

E. The macping Utility

We also implement ping packet in MAC layer to measure the latency. The MacPing packets' header is same as MacPerf packets, but the random payload is only a byte long.

When node1 tries to do the MacPing meausrement to node2, it sends one MacPing request packet per second, and begin a timer when the request packet is delivered to physical layer. For node2, the MacPing request packet is replied as soon as possible with a MacPing reply packet, and when node1 receive the MacPing reply packet, it stops the corresponding timer, printing the duration which is just the RTT of the packet.

Note that in both MacPerf and MacPing packets, adrress is taken into consideration and need to be handled carefully.

F. CSMA

Since it is possible that the channel (i.e. audio cable) is used by someone else, sending a packet may fail. To avoid this, we listen to the channel before we send a packet. The protocol is as follows.

- Initialize the safe bit to false.
- At a given time, the packet detection thread reads the power value. If it exceeds a threshold value,

someone else is using this channel, and set the safe bit to false. Otherwise, set it to true.
- When a thread wants to send a packet (a normal one or an ACK), it checks the safe bit. If it is false, wait (back off) for a few milliseconds and check the safe bit again, until this bit is true and then it may send a packet since the channel is considered free.

IV. Project 3: NAT

In project three, we build a gateway for the Athernet and in this way, Athernet devices is able to communicate with Internet. With NAT, we successfully transmit data by UDP packet between Athernet nodes and Internet. We further support ICMP protocol packets so that Athernet devices can ping Internet devices, and vice versa. In our project, the ping latency is about 140ms.

A. Forwarding packets between Athernet and Internet

We use different languages to establish connections in the Athernet and Internet. For the gateway, we need to share data between the two connections, so we use temporary files on the disk. To send a packet from Athernet to Internet, the working flow is as follows.

- A sends a packet to B via the Athernet.
- The Athernet connection of B stores the packet to a file F, and creates a notification file ATH_NOTIFY in the current directory.
- The Internet connection of B has a thread waiting for the notification file from the Athernet, ATH_NOTIFY. As soon as it finds this file in the current directory, it deletes it, changes the source of this packet to B, and sends the packet in F to the destination (i.e. C), which is also specified in the packet.

To send a packet from Internet to Athernet, the working flow is as follows.

- C sends a packet to B via the Internet.
- The Athernet connection of B stores the packet to a file F', and creates a notification file INT_NOTIFY in the current directory.
- The Athernet connection of B has a thread waiting for the notification file from the Internet, INT_NOTIFY. As soon as it finds this file in the current directory, it deletes it and sends the packet in F' to the destination (i.e. A), which is also specified in the packet.

B. ICMP ping

An important feature of Athernet is it support ICMP ping between Athernet and Internet. We implement both ping from Athernet devices to Internet devices and vice versa (ping from external devices). The working flow of ping from Athernet devices is (here we specify node1 is the Athernet node, node2 is the NAT node, and node3 is an internet device):

- node1 translates node3's IP address into bits, creates ICMP request packet, and starts a timer.
- node1 delivere the packet to physical layer and transmit it to node2.
- node2 check the packet's TYPE field, and if it is a ICMP ping request, node2 translate the IP address and write it to a notifying file.
- node2's python process detects the file, reads the IP address and sends a Internet ICMP ping request packet to node3.
- node3 automaticly reply with ICMP ping reply packet.
- node2's python process receive the reply, and notify C++ process to continue.
- node2 creates and sends an ICMP ping reply packet to node1.
- node1 receives the packet and counts the duration.

The working flow of ping from external is basically similar, but we need more operations like:

- we ban the system ICMP reply by modifying the Windows firewall.
- we use scapy library to catch ICMP packets and store the IP payload of the ping request.
- Athernet nodes should be able to reply ICMP request as soon as possible.
- to reply the Internet ICMP request, we need to modify the TYPE filed of ICMP header, and recalculate the ICMP checksum.

V. Project 4: FTP

In this project, we implement an Athernet client of the file transfer protocol (FTP), so that the user may retrieve files from the Internet. This depends on the gateway implemented in the last project.

The Athernet client node may specify the address of the FTP server, and is able to use FTP control commands. Among these commands, some simply sends short messages and receives short responses, such as USER, PASS, PWD, CWD, PASV. Some can initiate data transmission, such as LIST, RETR. In

order to simplify the behavior, at the client side, LIST prints the data to stdout, and RETR saves the data as a file in the current directory.

Assume that there are three nodes: *A* (Athernet), *B* (gateway), and *C* (Internet). The working flow is as follows.

- *A* specifies the address of the FTP server and tells *B*.
- *B* establishes an Internet connection with *C*. If success, there may be a response from *C*.
- *B* tells *A* the connection result and optionally with the responses from *C*.
- *A* prints a message saying if the connection is established or not, optionally with the response.
- *A* sends the FTP control command to *B*.
- *B* forwards the command to *C*.
- *C* sends the response to *B*, optionally with data if the command is LIST or RETR.
- *B* forwards the response to *A*, optionally with data.
- *A* receives the response, optionally with data. If the command sent is LIST, print it out; if it is RETR, save to a file. Then it prints the response.

Note: All the packets in Athernet are sent under a stop-and-wait protocol. The next packet may be sent only if the ACK of the current packet is received.

References

[1] Stanford University EE102B. Analog Transmission of Digital Data: ASK, FSK, PSK, QAM. https://web.stanford.edu/class/ee102b/contents/DigitalModulation.pdf.

[2] Stack Overflow. Translate CRC8 from C to Java. https://stackoverflow.com/questions/25284556/translate-crc8-from-c-to-java.

[3] Python 3.7.2 Documentation. socket — Low-level networking interface. https://docs.python.org/3/library/socket.html.

[4] GitHub Gist. A pure python ping implementation using raw socket. https://gist.github.com/pklaus/856268.