CS120 Final Report

Linjie Ma malj@shanghaitech.edu.cn
Daqian Cao caodq@shanghaitech.edu.cn

1. Physical Layer – Project1

1.1 Basic Part

1.1.1 Modulation

We choose PSK as the modulation method, using two different carrier waves, with phase shift 0 and phase shift π representing 0 and 1 respectively. We use an array of fixed-length sampling points to represent 1 (we use 16 sampling points for a bit). For 0, we just the opposite number of the corresponding value in the array.

1.1.2 Transmission

We choose the recommended chirp signal as the package header. Before transmitting a package, we first send a chirp signal to indicate the beginning. Then we transmit the remain modulated payload. Since we fix the length of the payload of every package, no package tail or information about package length is needed to be transmitted.

1.1.3 Demodulation

To demodulation a piece of message, we first need to find the package header. We use a fixed size queue **f** to find header. For every sampling point **v**, we pop the head of the queue and push **v** into **f**. Then, we calculate the inner product of **f** and a fixed-size array containing the chirp signal. If the result is larger than a threshold, a header is believed to be found. Otherwise, we wait for the next sampling point.

After finding a header, the following sampling points are the modulated data. Since the number of sampling point used to modulate a bit is fixed, we then gather fixed number of sampling points to demodulate one bit. To demodulate a bit, we calculate those sampling point with the carrier array. If the sum is larger than 0, we think the modulated bit is 1. Otherwise, we view it as 0. We continue demodulating after finding a header until enough bits are received, at which time we think we reach the tail of a package and will go back to the header finding phase.

1.1.4 Implementation

To implement the whole physical layer, we use JAVA and JAsioHost library. We implement a Listener Class as a wrapper of JAsioHost, which contains an input buffer to store all sampling points and an output buffer to store all sound signal to be sent. For physical layer, we implement PHY_Layer, PHY_Receiver, PHY_Sender Classes. The PHY_Layer is the interface for physical layer. It exposes the send and receive functions. The PHY_Receiver is a component of PHY_Layer. It will start a thread to keep demodulating sampled sound signal from the Listener and store demodulated information into PHY_Layer's buffer. The PHY_Sender is also a component of

PHY_Layer. It will modulate the given payload and store them into Listener's output buffer.

1.2 Optional Part

1.2.1 FEC

We implement Convolutional Code in the physical layer. We use 2 generators and 3 shift registers, which indicates that the length of encoded message is twice that of the origin message.

We use a state machine for encoding messages. For each bit in a message, we update the next state and generate the output using the current state and the input and then gather all output as the coded message.

We use Viterbi Algorithm to decode received message. The Viterbi Algorithm will go through all possible paths and record the hamming distances between paths and the received message. The path with minimum hamming distance is considered as the correct path and we get the input based on the path. For 2 paths which meet at the same state, we choose the one with a smaller hamming distance and discard the other. Since there are 4 states in our state machine, there will be 4 paths at most time and these 4 paths will split into 8 possible paths. For each state, there will be 2 paths which lead to it. Thus, we need to discard the path with larger distance. After discarding, 4 paths remain and the process continues until the end of the received message.

1.2.2 OFDM

We use 3 sine signals with frequencies 3000, 6000 and 9000 Hz respectively. We use the 3000Hz, 6000Hz, 9000Hz and 6000+9000Hz for 00,01,10,11, respectively. For each bit, we use 32 sampling points. For demodulation, we apply FFT on the received signal and analyze the amplitude of 3000Hz, 6000Hz, and 9000Hz to determine the existence of these frequencies.

1.2.3 Chip Dream

To avoid complex functions like sine and cosine, we use lookup tables for modulation and demodulation. When modulating bits, we just assign the lookup table of 0 or 1 to the hardware buffer. When demodulating, we also use lookup tables of chirp signal and carrier for calculating inner products.

To avoid floating point number, we magnify the received sampling points 1000 times and convert the result into an integer. When checking header and bits, all lookup tables contain magnified integers only. The threshold will also be magnified to fit the inner products.

1.3 Challenges

Modulation is easy and straightforward, but demodulation is hard and tricky. The hardest part in demodulation is how to precisely find the header. Although we will

traverse all sampling points to find headers, we could also find a location which is several points ahead or behind. For PSK, misaligning can directly destroy everything. To avoid this, we write a Matlab program to analyze all sampling points. The program performs the same with our java program but can give out plots of sound signals, inner products and some other debug information. We choose the threshold based on plots and information provided by the Matlab program and check again whether this threshold can help us find the most accurate index of headers.

Convolutional Code has some adjustable parameters which control the characters of it. However, since the decoding phase needs to go through all possible and discard suboptimal paths at run time, it is hard to implement a general decoder for all parameters. Due to the time limitation, we only implement a decoder with fixed parameters.

When implementing OFDM, we originally try to use 3 sine signals for 3 bits. But we find that 000 will be represented by no sending no signals using this method. If we stop sending sound occasionally during transmitting, the sound wave will still remain during period which should be no sound. Thus, the FFT will generate unexpected results. To solve this problem, we can only use a signal for all zero bits and use other signals each for a bit.

2. MAC Layer – Project2

2.1 Basic part

2.1.1 State Machine

We only implement ACK mechanism in the MAC layer using stop-and-wait mechanism. For sender, it will send out messages with source, destination, sequence number and payload using physical layer and wait for the ack message. If ack is not received after a fixed time slot, sender consider the receiver does not receive the message and will resend last message. Sender will only resend one message for 16 times. If ack is received, sender will continue to send remaining messages. For receiver, it will keep listening to the sender. Once a message is received, it will check the sequence number and send ack message back to indicate which message is received.

2.1.2 Implementation

We implement MAC_Layer Class as the interface of MAC Layer. It exposes send, receive_data, and receive_ack functions.

For send function, MAC_Layer will ask for source, destination and payload and pack them together with sequence number. Then, it will call the send function of PHY_Layer to send out the packed message. The send function of MAC_Layer will also record the send start time and last sent message for resending.

For receive_ack function, it will ask for a value indicating timeout and call the receive function of PHY_Layer. If a message is received, it will check the destination and

discard messages that not aims to the node. If no messages are received when reaching the deadline, it will resend the last sent message for up to 16 times, after which the function will terminate and return null.

For receive_data function, it is similar with receive_ack function except that it will not resend messages.

3. IP Layer – Project3

3.1 Basic Part

3.1.1 ICMP Echo

We are required to implement ICMP echo and reply in the athernet. Thus, beyond the MAC layer, we implement a "slim" IP layer, due to that the MTU of our athernet is limited to 256 bytes in order to balance the delay and bandwidth and keep the pack-drop rate around 1%. Meanwhile the project only required a few functions that a athernet Host must have, which causes us make the decision to discard the option part in conventional IPv4 packet. A maximum size of 200 bytes for the data is enough for all tasks' requirement. We send and receive the ICMP echo package using MAC_Layer handler.

3.1.2 Router

We are required to implement a router on NODE2 with multiple ports and interfaces. We maintain a simple static routing table that only have one entry for athernet host NODE1, which allows we to forward ICMP echo request to NODE2_3 where the athernet host is connected.

3.1.3 NAT

We are required to implement NAT (Network Address Translation) feature in the NODE2. We implement a simple NAT table which stores the information for ICMP service. NODE2 will listen to NODE1, NODE3 and the public server at the same time. It acts as a forwarder and forward ICMP packages between public server, NODE1, and NODE3.

3.1.4 Implementation

We use Pcap4j library to handle ICMP Echo packages capturing and sending. We implement Loop and SendEcho Classes as the interfaces for IP layer.

In Loop Class, we implement a simple static routing table to store information of NODE1. The Loop Class will run 2 threads for athernet port and routing port respectively. The thread for athernet port is based on the MAC_Layer handler and is used for communicating with NODE1. The thread for routing port is based on Pcap4j and is used for communicating with NODE3.

In SendEcho Class, we implement a simple NAT table. Similar with Loop class, it will run 2 threads. Due to the requirement that we only need to manage ICMP echo

packages between NODE1 and 1.1.1.1, the NAT table can only support forwarding ICMP echo request from NODE1 to port NODE2_2 and forward ICMP echo reply from 1.1.1.1 to port NODE2_3.

3.2 Optional Part

3.2.1 NAT Traversal

This task is similar with the NAT task. The difference is that we need to retrieve payload from the ICMP Echo request packages and forward the request to different devices based on the IP address in the payload.

3.2.2 IP Fragmentation

This task requires fragmented ICMP packages. We check the size of the payload in ICMP Echo request packages in the router and divide the package into fragments which are not larger than MTU of athernet. We create fragments by modifying a helper function in Pcap4j. Then, we send out fragments and forward the reply packages to NODE4. Reassembling is done by NODE4.

3.3 Challenges

When implementing NAT Traversal, we have problems caused by hardware. In NODE2, the network interfaces are different for hotspot and WLAN, but we fail to use Pcap4j to open 2 network interfaces simultaneously. Thus, we use java library to open another thread which runs commands in the powershell to forward packages to NODE4. It's a tricky solution but it has a good performance that only cost an additional 60ms in average for ICMP echo from NODE4 to NODE3.

When implementing IP Fragmentation, we forget to divide packages into fragments whose sizes are a multiple of 8 at the beginning and cannot get any packages from NODE4. Later, by checking the functions for fragmenting in Pcap4j, we realize this problem and fix the bug.

4. IP Layer – Project4

4.1 Basic Part

4.1.1 DNS

For each DNS query from NODE1, the NODE2, as a router, will retriever domain name and send a DNS query to the DNS server. When receiving the DNS request from server, NODE2 forwards it back to NODE1. Then NODE1 can send ICMP Echo to the IP address.

4.1.2 Implementation

We use java library to run a command in the powershell to send out DNS queries and use Pcap4j library to capture the DNS reply packages. The communication with NODE1 is completed by using MAC_Layer handler similar with Project3.