CS120 Athernet Report

1st Wang
ShanghaiTech University
Shnghai, China
wangjw@shanghaitech.edu.cn

1st Qin
ShanghaiTech University
Shnghai, China
qinfr@shanghaitech.edu.cn

I. Project 1

A. 音频基础

本次 Project 包括后续 Project 采用了 C# 的 NAudio 库,其包含了对 ASIO 的驱动封装。ASIO可实现低延迟、高同步、高吞吐率以及对声卡的直接访问,而不用承担系统音频接口的 overhead. C# 相比如 Rust, Go, Python 的语言可以提供适合的运行时与编译时的性能差异,而且更加灵活,更加容易维护。

对于收到的每一个音频信号 (sample),该音频库可以通过事件的方式来激活一个函数,并传入对应的浮点值。

B. 调制与解调

我们采用了 Passband Modulation 作为调制方式。更具体来说是采用了 Phase Shift Keying(PSK). 由于是无线传输,我们接收到的信号响度难以控制。PSK对响度没有要求,因此较为合适。

1) 调制: 对于每一个bit,我们将其转换为对应的波形。0代表的波形和1代表的波形相差180度的相位。而对于标志信号开始的preamble,我们采用了调频连续波(FMCW)。这种波形优点在于非常独特,识别错误率低且精准,缺点在于占用了太多的频谱资源。不过对于项目来说,频谱资源并不稀缺,因此它是一个合适的选择。

2) 解调: 解调器需要时时刻刻去计算前一段时间内接收到的的信号与 preamble 是否一致。因此他需要维护一个小的 ring buffer (即一个支持随机访问的 queue),每收到一个 sample ,就添加到 buffer末尾,并将最老的 sample 从 buffer 中取出,丢弃。然后他将 buffer 与 preamble 进行点乘(元素相乘再相加),若达到一定阈值则认为收到了这一信号,并以这一时刻作为信号开始的时刻。除非在一定时间内,又出现了能量更高的时刻,则以新的时刻为准。(与 matlab 演示代码逻辑相同)

Identify applicable funding agency here. If none, delete this.

在接收到 preamble 之后,就开始解调数据了。和刚才一样,将收到的一串 sample和 0 代表的波形做点乘,若大于0则认为这一位是 0,反之认为是 1. 在读取了预先设定的 bits 数量后停止读取,或者通过接收到的数据中的长度数据判断读取长度。我们最终采用了第二种方法,但由于在后续项目中被重写,在此就不多做叙述了。

3) 参数设置与性能: 调制解调过程中有许多需要设定的参数。经过各种测试,我们大致采用了 4000Hz 频率的载波,1000Hz 到 10000Hz 的调频连续波,用 24 个 sample 表示一个 bit.

这样理论瞬时 Throughput 最高能达到 $\frac{48000}{24} = 2000bps = 2kbps$.

4) 其他挑战: 由于采用的是无线传输,收到的音频信号响度很不稳定。因此如何设定能量的阈值是一个挑战。如果收到的音量较小,阈值应该低,反之阈值应该较高。

另外,外界干扰也较多,在房间内还极其容易受到回声干扰,因此总体可靠性很难得到保证。

C. 校验与纠错

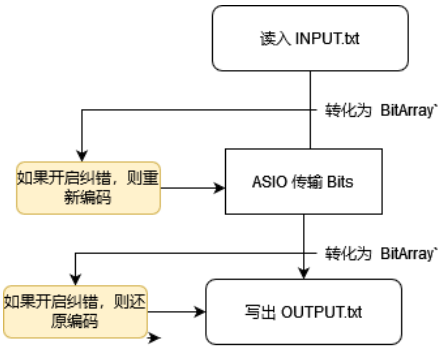

图 1. 校验与纠错示意图.

对于传入的 bit array, 我们采用自制与里德-所罗门码混合的校验与纠错机制。

其中,对于要传入的数据,我们按照 $14 \times 7 = 98$ bit 将其分成若干 frames。对于每个 frame, 将其分成 14 个 7 bits。

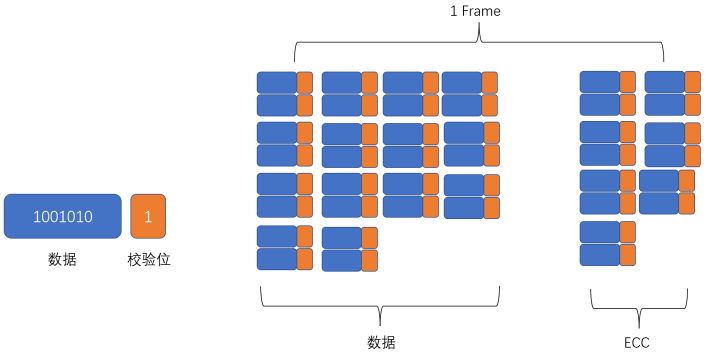

图 2. 校验与纠错结构设计.

对于这 14 个 7 bits, 我们使用里德-所罗门码生成 7 个 7bits 的 ECC 块。对于这 20 个 7 bits,我们规定其最后一位为前 7 个 bits 中 1 的个数是否为偶数的 0/1 值。

这样对于收到的 frames,只要接受到对的数据块至少 14 个,就可以还原出原本的 98 bits。在完成这一步之后,数据传输正确率可以做到几乎每次 100%。

通过观察,我们发现 Athernet 的错误特点是传输过程中会有连续数据的错误,因此对于我们的设计,横向数据(即每个 frame 中的数据)连续错误的概率较大。因此我们设想如果我们能“竖着”对其做里德-所罗门码可以提高对错误发生的针对性。

由于我们的传输效率较高,因此我们还采用了在规定时间内两倍发送数据的方式来增加冗余。当里的所罗门算法发现当前 frame 出现错误后会采用第二次收到的 frame 值。

D. 错误与改进

我们的 frames 必须保证严格对齐,因为我们是根据收到的顺序进行还原。未来的改进可以设计一个协议,在传输的 frame 中加入对应的 index.

II. Project 2

A. 功能概述

在Project 2中,我们实现了一个基于有线传输的软件Modem(调制解调器)。它所承担的工作与真正的以太网调制解调器类似,区别在于它额外地通过ACK机制提供了可靠传输的能力。因此,整个Modem就成了一个可靠的2层设备,在后续的项目中可以方便地使用。

对于上层,Modem提供了简洁的API (例如 Transport-Data(destination, data)),另外在此次项目中也实现了一个简单的命令行方便测试。

B. 物理层(PHY)

与之前不同,这次我们可以依靠较为可靠的有线传输,因此为了更高效的传输我们对物理层进行了重做。Frame总体来讲模仿了 以太网的格式

图 3. Frame设计.

1) Preamble: 由3个10101010组成,用以让接收方获知即将到来一个Frame。(1代表正电平,0代表负电平)

2) Start frame delimiter: 是10101011,用以让接收方准确知道Frame的开始点。(1代表正电平,0代表负电平)

3) Frame body: 物理层的 Payload. 在这次项目中,我们采用了 Line coding 即基带传输,使用 4B5B 与 NRZI 的结合来使得音频信号足够稳定,不会长期保持在一个电平上,进而增加传输的可靠性。另外,由于 NRZ 的特性,传输过程中电平不会保持在零电平上。

详细的讲,给定 Payload,首先通过 4B5B 的转换,将其转换为不拥有超过 3 个连续 0 的串,然后设定一个初始相位(例如正电平),开始传输。过程中每遇到一个 1,就改变相位。这样同一个电平最多只会持续 3 个 bit 的长度。

4) 性能: 在硬件状况良好的情况下,我们可以靠 2 个 sample 来表达一个 symbol. 同时采样率为 48000Hz,算得波特率是 24000symbol/s. NRZI 没有 Overhead,4B5B 的效率是 0.8bit/symbol. 这样理论瞬时数据传输速率可以达到 $24000\ symbol/s \cdot 0.8bit/symbol = 19200bps = 19.2kbps$.

5) 其他尝试: 我们也尝试过使用 MLT-3(Multi-Level Transmit of 3 levels)Encoding,这是一种拥有4个相位(-1,0,+1,0)的Line coding,虽然传输效率不会得到提升,但是会让信号的整体频率下降至波特率的四分之一,对线材的要求更低。但实际测试后,发现对+1、0、-1的采样识别错误率较高,不如+1、-1实用,因此被废弃。

C. 链路层(MAC): 基础

在这里根据项目要求,我们在 MAC 层做了 ACK 机制。由于被迫工作在自干扰环境下,且由于声卡采样率限制导致带宽极低,即便做了滑动窗口想必也很难将窗口设置在1以上,因此我们采用停等协议作为ACK机制的工作方式。MAC 层 Frame 内容图示:

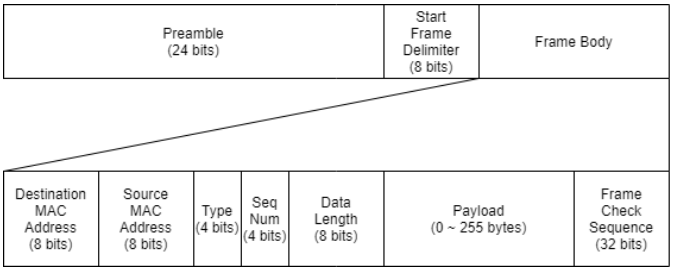

图 4. MAC 层 Frame设计.

1) Destination & source MAC address: 用以标志发送方和接收方。项目中用 8bit 表示 MAC 地址, 因此最多支持 256 台设备。

2) Type: 用以标志 Frame 的类型。4bit 表示,因此最多有 16 种。目前已经使用的共有如下几种:

```
public enum FrameType
{
    Data = 0,
    Data_Start = 1,
    Data_End = 2,
    Acknowledgement = 3,
    MacPing_Req = 4,
    MacPing_Reply = 5
}
```

图 5. Type 设计.

3) Sequence number: 用以标志当前包的序号,或者 ACK 的包的序号。4bit 表示,因此范围是 0 15.由于采用的是停等协议,sequence number 仅用于接收方确认是否收到了重复的包,因此这一范围已经足够。

4) Data length: 用以标志 Payload 的长度(bytes)。8bit 表示,因此 Payload 长度只能在0 255 bytes之间。值得一提的是,除了 Type = Data 的 Frame,其余 Frame 都不应该有 Payload,即 Data length = 0。

实际上,为了防止包过长导致传输时间较长从而导致受干扰或时钟不同步,Payload 长度会被更进一步限制。不同电脑之间传输时,一般限制为 128 bytes 以内。

5) Payload: MAC 层的 Payload.

6) Frame check sequence: 即一串 CRC-32 的校验码,校验范围是整个 MAC 层的包中,不包括 Frame check sequence 的范围。

7) Performance: 根据测试,当 ASIO 的 buffer size 设置为 128 或 64 时,RTT可以做到 40 50 ms。我们假定 RTT 为 45ms. 除此之外,我们设定最大 Payload 长度为128 bytes. 每发完一个包都需要等待一个 RTT 来收 ACK.

其中,传输128 bytes需要的时间是$128 \cdot 8/19.2 = 53.33ms$.

这样,整个 MAC 层的 Throughput 可以达到 $128 \cdot 8/(53.33 + 45) = 10.41kbps$.

实际上,最大包大小对速率影响极大,因此还是要以实际需要为准。例如当没有干扰、硬件状态良好时,可以采用更大的包大小。当要求可靠性或有干扰时,可以采用更小的包大小。

D. 链路层(MAC): CSMA/CD

我们通过模仿半双工以太网的工作机制实现了 CSMA/CD.链路层通过统合 Rx(物理接收)和 Tx(物理发送)来达到提高传输效率的目的。

1) 接收端: Carrier sense,顾名思义,其核心在于感知。因此在接收端我们需要增加感知信道的功能。

在项目中,我们规定,当接收端连续收到一定数量(例如 16 个)的安静(绝对值小于某个阈值)的 sample 后,认为信道是空闲的。反之,一旦收到1个不安静(绝对值大于某个阈值)的 sample 后,认为信道是被占用的。其中阈值可以是,例如, 0.2倍的标准电平。

额外的,当收到1个特别响(绝对值大于某个阈值,例如 1.5 倍的标准电平)的sample 后,会触发一个 Collision 事件。此时我们认为 Collision 发生,因为只有 Collision 才会产生高于标准电平的信号。

2) 发送端: 由于不是随时能够发送,因此发送端有一个 Queue 用来缓存准备发送的包。在有新的包准备发送时,首先检查信道是否空闲。如果空闲则立即发送,如果不是则等到空闲再发送。

当包正在发送过程中,监听到了 Collision 事件时,立即取消发送。此时其他接收者会认为信道变得空闲,但包并没有接收完成,因此会立即认为接受失败。这里利用了 NRZ 的好处,即只要是零电平就是没有在传输,来避免了需要发送 Jamming 通知其他设备的必要。同时,发送端会选择 Backoff 时间后重新传输,或放弃传输。具体机制和以太网类似,仅仅是参数上有一些区别。例如,timeslot 有所不同(理论上应该设为大于 RTT,即 50ms 以上),最大的幂次应该降低(避免太长时间的等待),最大的尝试次数应该降低(避免太长时间的等待).

3) 问题: CSMA/CD 是对延迟极其敏感的。Athernet 延迟太大,会导致接收端的感知太过滞后,变得不准确,甚至会降低效率。而我们又有 ACK 机制,大大提高了 Collision 的可能性。因此我认为在这个项目中实现这些并不是实用角度合理的,但确实可以学到很多。

E. 其他细节

这次 Project 还涉及到很多其他技术细节,例如管理接受进程、发送进程,物理发送、接收端的具体工作细节等等。

除此之外还有一些方便使用的设计,例如支持把收到的信号作为音频录制下来方便调试,例如所有的参数都写入了配置文件,在运行时会进行读取,不需要重新编译,更方便调试。另外还有较为完善的程序使用体验,例如引导输入 MAC 地址,选择、配置声卡驱动,以及通过命令行操作等。对于这些细节,在此就不多做剖析了。

III. Project 3

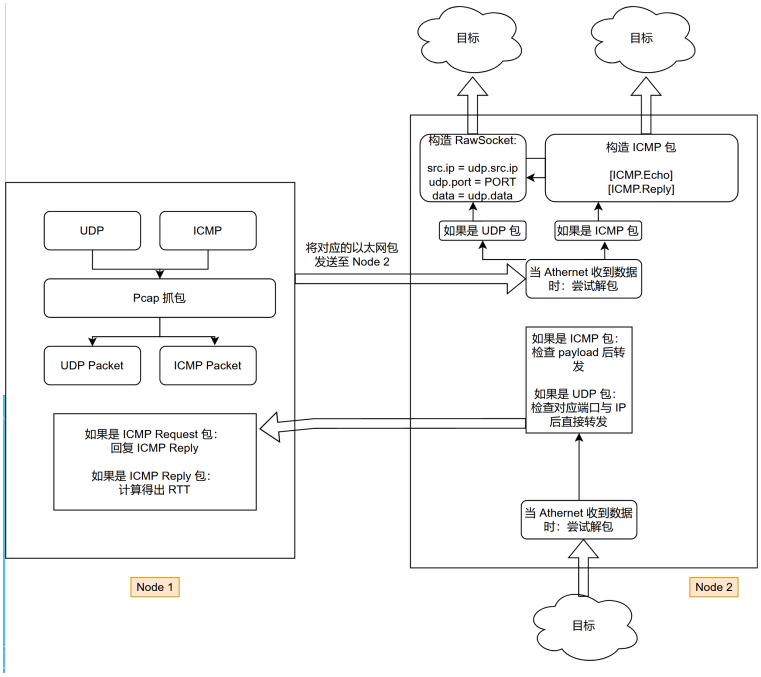

图 6. Project3 设计图.

A. UDP Forwarder

对于发送 UDP,我们使用 PacketDotNet 进行构建包。

```
public EthernetPacket ConstructUdp(byte[]
    dgram, string destination)
{
   //construct ethernet packet
   var ethernet = new EthernetPacket(
      PhysicalAddress.Parse("112233445566"),
      PhysicalAddress.Parse("665544332211"),
      EthernetType.IPv4);
   //construct local IPV4 packet
   var ipv4 = new
       IPv4Packet(IPAddress.Parse(IP),
       IPAddress.Parse(destination));
   //construct UDP packet
   var udp = new UdpPacket(12345, 54321);
   //add data in
   udp.PayloadData = dgram;
   udp.UpdateCalculatedValues();
   ipv4.PayloadPacket = udp;
   ipv4.UpdateCalculatedValues();
   ethernet.PayloadPacket = ipv4;
   ethernet.UpdateCalculatedValues();
   // Console.WriteLine(ethernet);
   return ethernet;
```

之后通过 Athernet 将 ethernet.data 传输至 Node2. 当 Node2 接收到 bytes 之后会进行解析,如果是一个 UDP 包,将会用 Socket 的方式将接受到的 UDP.PayloadData 包入 Socket 由系统的网络栈发送,这样可以复用系统的数据链路层实现。

```
if (udpPacket != null)
{
   // begin to send to Node 3
   var server = new Socket(
      AddressFamily.InterNetwork,
      SocketType.Dgram, ProtocolType.Udp);
   server.Bind(
      new IPEndPoint(IPAddress.Parse(IP),
          12345));
   var endpoint =
      new
          IPEndPoint(ipPacket.DestinationAddress,
          54321);
   server.SendTo(udpPacket.PayloadData,
       endpoint);
   server.Close();
}
```

反之,当 Node2 接受到一个 UDP 包的时候将会进行判断,如果 UDP 端口是预先确定的(12345 ¡-¿ 54321),则将整个 Ethernet 包通过 Athernet 转发至 Node1. Node1 将会进行解包。

B. ICMP Forwarder

ICMP 的实现与 UDP 一致

```
public EthernetPacket ConstructICMP(string
    destination, IcmpV4TypeCode icmpV4TypeCode)
{
   //construct ethernet packet
   var ethernet = new EthernetPacket(
      PhysicalAddress.Parse("112233445566"),
      PhysicalAddress.Parse("665544332211"),
      EthernetType.IPv4);
   //construct local IPV4 packet
   var ipv4 = new
       IPv4Packet(IPAddress.Parse(IP),
       IPAddress.Parse(destination));
```

```
    ethernet.PayloadPacket = ipv4;
    const string cmdString = "Hello CS120";
    var sendBuffer =
        Encoding.ASCII.GetBytes(cmdString);
    var headerBuffer = new byte[8];

    var icmp = new IcmpV4Packet(new
        ByteArraySegment(headerBuffer));
    ipv4.PayloadPacket = icmp;
    icmp.TypeCode = icmpV4TypeCode;
    icmp.Checksum = 0;
    icmp.Sequence = 1;
    icmp.Id = 1;
    icmp.PayloadData = new byte[]{0xff};
    byte[] bytes = icmp.Bytes;
    icmp.Checksum =
       (ushort)ChecksumUtils.OnesComplementSum(
          bytes, 0, bytes.Length);
    ipv4.UpdateCalculatedValues();
    ethernet.UpdateCalculatedValues();
    return ethernet;
}
```

这里有一个问题在于 PacketDotnet 库对于 ICMP 的UpdateCalculatedValues() 函数实现有误,需要手动更新 ICMP 中的 Checksum. 我们组将会在课程结束后向作者提交一个补丁以修复此问题。

之后通过 Node2 发出

```
if (icmpPacket is {TypeCode:
    IcmpV4TypeCode.EchoRequest or
    IcmpV4TypeCode.EchoReply})
{
    Console.WriteLine(icmpPacket.ToString());
    var s = new
        Socket(AddressFamily.InterNetwork,
       SocketType.Raw, ProtocolType.Icmp);
    var a = new Random().Next(20000, 65536);
    var b = new Random().Next(20000, 65536);
    s.Bind(new IPEndPoint(IPAddress.Parse(IP),a
       ));
    Debug.Assert(icmpPacket != null,
        nameof(icmpPacket) + " != null");
    s.SendTo(icmpPacket.Bytes,
       new
           IPEndPoint(ipPacket.DestinationAddress,
          b));
    s.Close();
}
```

在 Windows 中,需要以管理员权限运行,并且在高级防火墙设置中禁止 ICMP inbound 以防止系统回复 ICMP.

IV. Project 4

在 Project 4 中,我们组进行了多种方法的讨论,最后选择实现了其中较为简单的方式,在此将会记录三种不同的思路。

A. FTP message forward

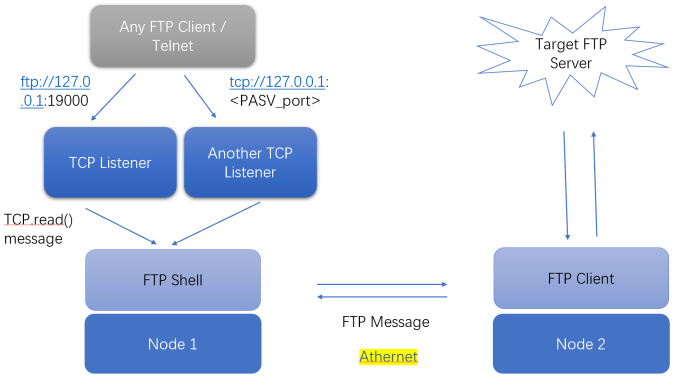

图 7. Project4 的第一种实现设想.

首先,在 Node1 上使用系统自带的 TCP Listener 进行端口监听。当可以从中读到信息时,进行 Parse. 我们实现了一个不完全的 FTP 指令 Parser,将 Parser 过的 FTP 指令通过 Athernet 发至 Node2. 在 Node2 上使用我们自己实现的 FTP 客户端进行与目标 FTP 的通信。将收到的信息通过 Athernet 发往 Node1. Node1 再通过 TCP Listener 写回。

特别的, PASV 指令将会开启一个新的 TCP Listener 线程。通过对 PASV 返回消息的 Parse,我们可以一对一的开启一组端口映射,同时在 Node1 的 TCP Listener 和 Node 2 的 FTP 客户端中的 TCP Client 中。

关于如何区分 Athernet 中 FTP Message TCP Listeners 的问题,我们使用 Magic byte 进行区分,当首位 byte 为 0xff 时,就放入队列中供 Another TCP 调用。

好处在于可以略去 TCP 协议栈,并且兼容 FTP Client。在实现过程中,因为 TCP Listener 写的过程中无法一次性将栈中的数据全部写出而鲁棒性不佳。

B. 其它实现思路

1) TCP 传输: 通过改写路由表,route add 10.11.135.168 mask 255.255.255.255 10.11.135.1,使 loop back 的包经过网卡,得以被 npcap 捕获。

在实现过程中,抓包 TCP 中出现多个返回包,并且无法使用 Socket 发送,因此该方案被搁置。

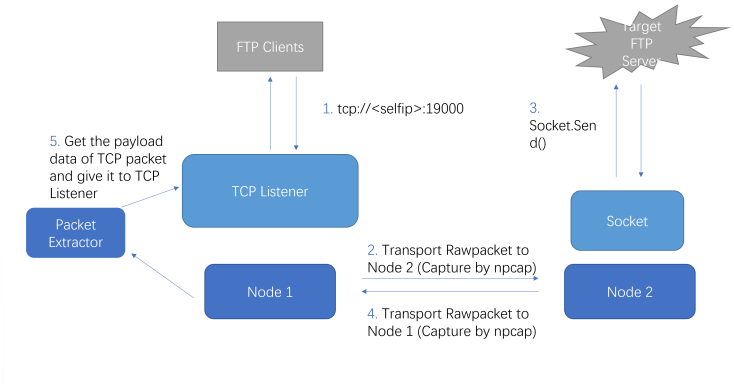

图 8. Project4 的第二种实现设想.

C. 网卡驱动

微软在 Windows10 中引入了一种简化的网络驱动编程方式在 netcx 和 NetAdapter-Cx-Driver-Samples, 可以实现基于 AtherNet 的网卡驱动。