DREXEL UNIVERSITY
College of
Computing & Informatics

Network Programming Primer

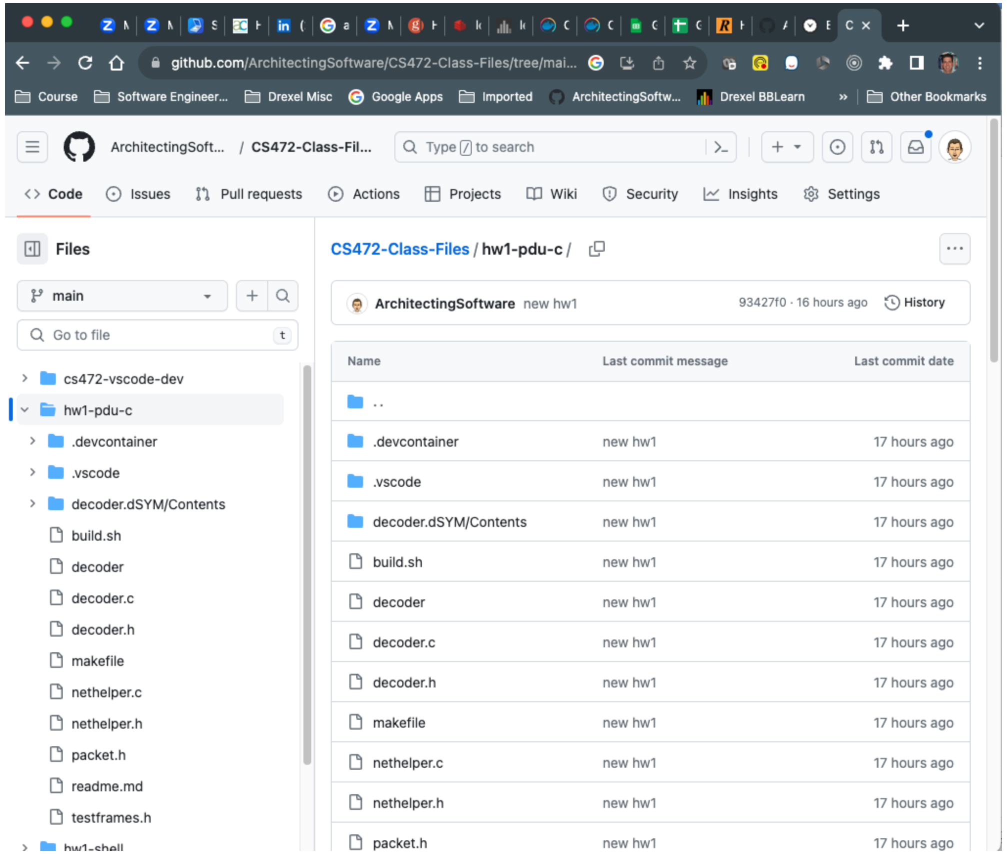
Brian S. Mitchell

Programming Assignment #1

For this class, all programming assignments will be provided via the course GitHub link which can be downloaded to your machine by running:

git clone https://github.com/ArchitectingSoftware/CS472-Class-Files.git

The first assignment is under the /hw1-pdu-c folder



Programming Assignment #1 - Objectives

- Refresh your C programming skills, assuming you have not used them in a while
- Gain some experience working with a few real-world network protocols in the TCP/IP family, without having to understand them (yet)
- Gain some experience using systems programming in C that are common practices when doing network programming
- Use Wireshark to capture some network traffic and decode it using your C program

Programming Assignment #1 – What You Will Be Doing

- If you are proficient at C this will be a REALLY EASY assignment. If you are not, it will be valuable to refresh your C programming skills
- Total solution is about 40-50 lines of basic C code.
- Entire solution is based on understanding of how to use buffers, and overlay them with structures to decode them.
- We will be decoding raw network streams/frames/packets encoded with:
  - The ARP protocol (simplest)
  - The ICMP protocol (a little more involved), specifically an ICMP ECHO frame which is used by the ping utility

Programming Assignment #1 – ARP

```
uint8_t raw_packet_arp_frame78[] = {
0xc8, 0x89, 0xf3, 0xea, 0x93, 0x14,
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50,
0x08, 0x06, 0x00, 0x01, 0x08, 0x00,
0x06, 0x04, 0x00, 0x01, 0xa0, 0x36,
0xbc, 0x62, 0xed, 0x50, 0xc0, 0xa8,
0x32, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xc0, 0xa8, 0x32, 0x63,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

```
Packet length = 60 bytes
Detected raw frame type from ethernet header: 0x806
Packet type = ARP
ARP PACKET DETAILS
    htype:   0x0001
    ptype:   0x0800
    hlen:    6
    plen:    4
    op:      1 (ARP REQUEST)
    spa:     192.168.50.1
    sha:     a0:36:bc:62:ed:50
    tpa:     192.168.50.99
    tha:     00:00:00:00:00:00
```

FOR the ARP protocol, you will detect that the raw frame data (left) is an ARP Request packet and then decode it into its components as shown on the right.

Programming Assignment #1 – ICMP (ECHO)

```c
uint8_t raw_packet_icmp_frame362[] = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
0x45, 0x00, 0x00, 0x54, 0x2a, 0xec, 0x00,
0x00, 0x40, 0x01, 0x89, 0x31, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0x08,
0x00, 0x7b, 0xda, 0x48, 0x59, 0x00, 0x00,
0x65, 0x0e, 0x01, 0xee, 0x00, 0x00, 0xe1,
0xcc, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22,
0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
};
```

```
Packet length = 94 bytes
Detected raw frame type from ethernet header: 0x800
Frame type = IPv4, now lets check for ICMP...
ICMP Type 11
Error: Expected an ECHO REQUEST or an ECHO response
ERROR: We have an ICMP packet, but it is not of type echo

-------------------------------------------------
TESTING A NEW PACKET
-------------------------------------------------
Packet length = 98 bytes
Detected raw frame type from ethernet header: 0x800
Frame type = IPv4, now lets check for ICMP...
ICMP Type 8
ICMP PACKET DETAILS
    type:      0x08
    checksum:  0x7bda
    id:        0x5948
    sequence:  0x0000
    timestamp: 0x650e01eee1cc
    payload:   48 bytes
    ECHO Timestamp: TS = 2023-09-22 21:06:54.57804
```

FOR the ICMP protocol, you will detect that the raw frame data (left) is an ICMP Request packet and then decode it into its components as shown on the right.

Programming Assignment #1 – ICMP (ECHO) Continued

```
uint8_t raw_packet_icmp_frame362[] = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
0x45, 0x00, 0x00, 0x54, 0x2a, 0xec, 0x00,
0x00, 0x40, 0x01, 0x89, 0x31, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0x08,
0x00, 0x7b, 0xda, 0x48, 0x59, 0x00, 0x00,
0x65, 0x0e, 0x01, 0xee, 0x00, 0x00, 0xe1,
0xcc, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22,
0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
};
```

```
PAYLOAD

OFFSET | CONTENTS
-------------------------------------------------------
0x0000 | 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0x0008 | 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17
0x0010 | 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x0018 | 0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27
0x0020 | 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
0x0028 | 0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37
```

Continued, unlike ARP, the ICMP ECHO protocol also has a payload, this will need to be decoded as well

Programming Assignment #1 – OTHER Protocols

```c
uint8_t raw_packet_icmp_frame198[] = {
0xc8, 0x89, 0xf3, 0xea, 0x93, 0x14, 0xa0,
0x36, 0xbc, 0x62, 0xed, 0x50, 0x08, 0x00,
0x45, 0xc0, 0x00, 0x50, 0xa8, 0xfa, 0x00,
0x00, 0x40, 0x01, 0xeb, 0x3d, 0xc0, 0xa8,
0x32, 0x01, 0xc0, 0xa8, 0x32, 0x63, 0x0b,
0x00, 0xbb, 0xbd, 0x00, 0x00, 0x00, 0x00,
0x45, 0x00, 0x00, 0x34, 0xea, 0x3b, 0x00,
0x00, 0x01, 0x11, 0x08, 0xf2, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0xea,
0x3a, 0x82, 0x9b, 0x00, 0x20, 0xcc, 0x4b,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00
};
```

Packet length = 94 bytes
Detected raw frame type from ethernet header: 0x800
Frame type = IPv4, now lets check for ICMP...
ICMP Type 11
Error: Expected an ECHO REQUEST or an ECHO response
ERROR: We have an ICMP packet, but it is not of type echo

You will also look at other packets and at least be able to determine that its not an ARP or an ICMP ECHO
(Note this is a traceroute packet)

Programming Assignment #1 – What You Need to Do

- For this assignment you need to start and demonstrate you can decode the three example frames I provided. These are included in the “testframes.h” file. I also have a working solution that is provided as a linux binary in the /sample-solution folder
- The second part will require you to capture a few of your own frames using wireshark and decoding them as well.

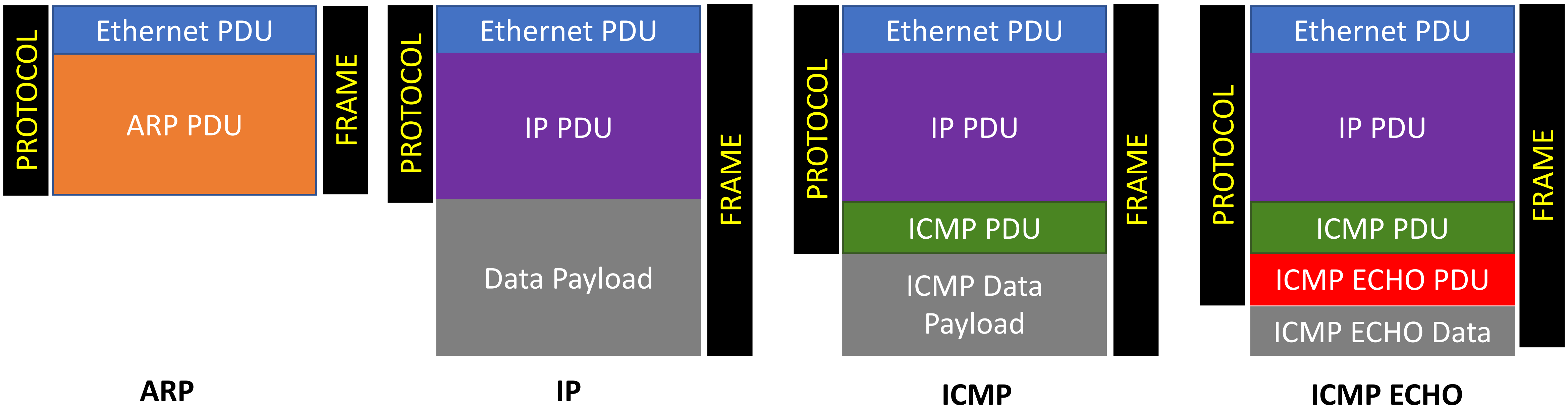
- You can hand in with blackboard a zip file containing all of your code, or even better, create your own GitHub or GitLab repo and submit a link to your solution.

Appendix 1: Intro to our Initial Protocols

Ethernet, ARP, ICMP and ICMP Echo

Basic Terminology – PDU/Protocol/Frame

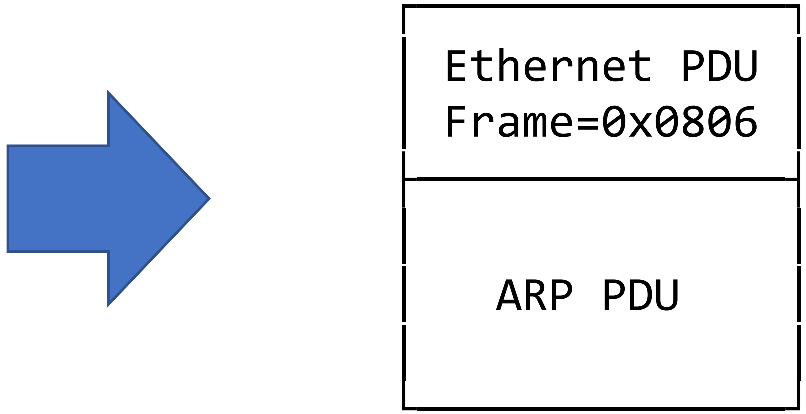
- PDU – Protocol Data Unit – For now think of it as a basic data structure that outlines the key properties of a given protocol
- Protocol – We will be studying how network protocols are layered (aka) they stack on each other. For the ones we are using in this initial assignment:



Notice that the protocol is the stacking of various PDUs. The frame is the entire unit that is transferred over the network. The frame often has protocol specific data at the end. We will see other protocol types in this class like TCP/IP, UDP/IP, HTTP, etc

Detecting ARP – Ethernet Frame-Type=0x0806

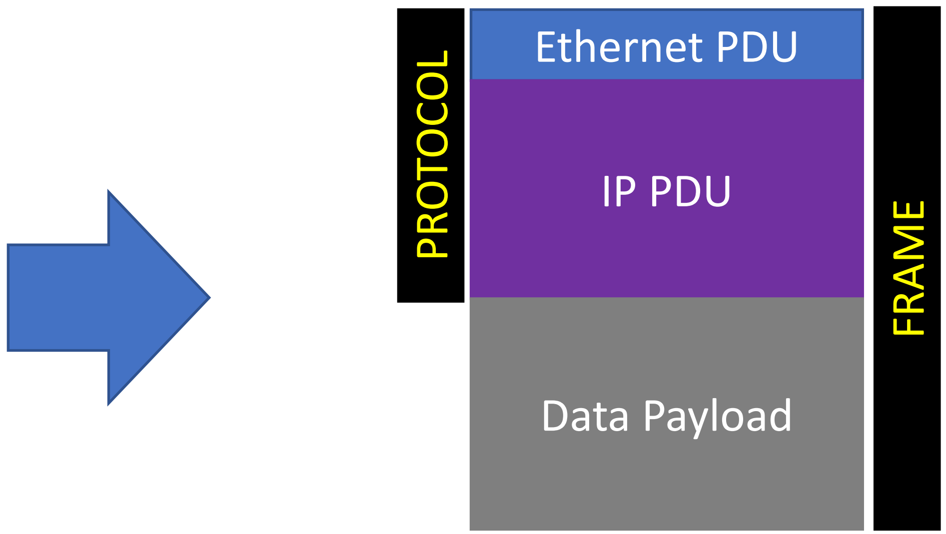
```
uint8_t raw_packet_arp_frame78[] = {
0xc8, 0x89, 0xf3, 0xea, 0x93, 0x14,
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50,
0x08, 0x06, 0x00, 0x01, 0x08, 0x00,
0x06, 0x04, 0x00, 0x01, 0xa0, 0x36,
0xbc, 0x62, 0xed, 0x50, 0xc0, 0xa8,
0x32, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xc0, 0xa8, 0x32, 0x63,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```



The ethernet PDU is always 14 bytes. It has a field that specifies that what follows is an ARP pdu

Detecting IP – Ethernet Frame-Type=0x0800

```
uint8_t raw_packet_icmp_frame362[] = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
0x45, 0x00, 0x00, 0x54, 0x2a, 0xec, 0x00,
0x00, 0x40, 0x01, 0x89, 0x31, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0x08,
0x00, 0x7b, 0xda, 0x48, 0x59, 0x00, 0x00,
0x65, 0x0e, 0x01, 0xee, 0x00, 0x00, 0xe1,
0xcc, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22,
0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
};
```



The ethernet PDU is always 14 bytes. It has a field that specifies that what follows is an IP PDU. The IP PDU is 20 bytes and describes the payload that follows

Detecting IP – Ethernet Frame-Type=0x0800
IP Protocol = 0x01

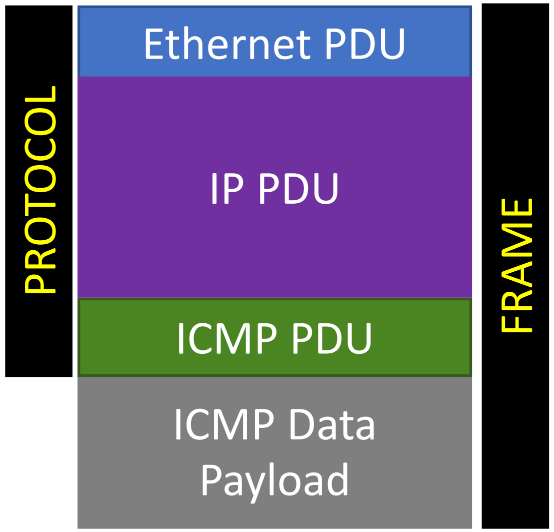
```c
uint8_t raw_packet_icmp_frame362[] = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
0x45, 0x00, 0x00, 0x54, 0x2a, 0xec, 0x00,
0x00, 0x40, 0x01, 0x89, 0x31, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0x08,
0x00, 0x7b, 0xda, 0x48, 0x59, 0x00, 0x00,
0x65, 0x0e, 0x01, 0xee, 0x00, 0x00, 0xe1,
0xcc, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22,
0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
};
```

Indicates IP PDU Next

Total Length IP PDU plus Data is 0x54 or 84 bytes

Indicates ICMP PDU Follows

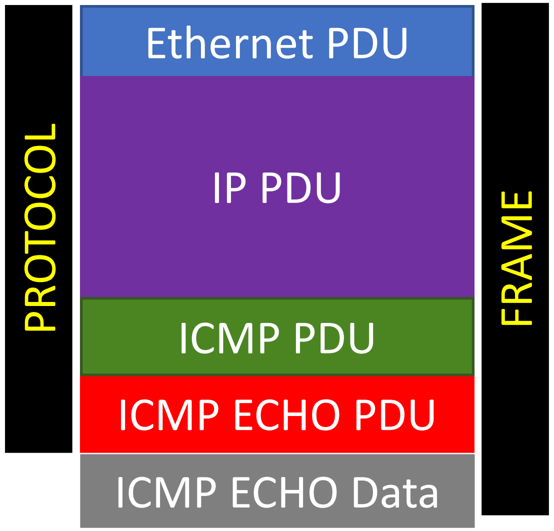
The basic ICMP PDU is only 4 bytes



Detecting IP – Ethernet Frame-Type=0x0800
IP Protocol = 0x01, ICMP Type = 0x08

```
uint8_t raw_packet_icmp_frame362[] = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
0x45, 0x00, 0x00, 0x54, 0x2a, 0xec, 0x00,
0x00, 0x40, 0x01, 0x89, 0x31, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0x08,
0x00, 0x7b, 0xda, 0x48, 0x59, 0x00, 0x00,
0x65, 0x0e, 0x01, 0xee, 0x00, 0x00, 0xe1,
0xcc, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22,
0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
};
```

Indicates that this is an ECHO (ping) Request



The basic ICMP ECHO PDU Header is 12 Bytes

We can calculate the payload to be 48 bytes. From last slide, IP length was 0x54 or 84 bytes. IP PDU = 20 bytes; ICMP PDU = 4 Bytes; ICMP ECHO PDU = 12 bytes. So 84-20-4-12 = 48 bytes. See the ICMP_Payload_Size macro in packet.h

Packet.h – From Assignment 1

This is the key include file that defines everything you need to decode raw network data. It defines the individual PDUs for Ethernet, ARP, IPv4, ICMP, and ICMP-Echo. All of these structures end with xxxx_pdu_t. This file then continues defining the packet structures by including the appropriate PDUs in correct order.

```
typedef struct ether_pdu{
   uint8_t dest_addr[ETH_ALEN];
   uint8_t src_addr[ETH_ALEN];
   ube16_t frame_type;
} ether_pdu_t;

typedef struct arp_pdu{
   ube16_t htype;
   ube16_t ptype;
   uint8_t hlen;
   uint8_t plen;
   ube16_t op;
   uint8_t sha[ETH_ALEN];
   uint8_t spa[IP4_ALEN];
   uint8_t tha[ETH_ALEN];
   uint8_t tpa[IP4_ALEN];
} arp_pdu_t;
```

```
typedef struct arp_packet{
   ether_pdu_t eth_hdr;
   arp_pdu_t   arp_hdr;
} arp_packet_t;
```

This file also defines all key constants to determine things like the frame_type in the ethernet PDU to see if the data following the PDU is an ARP or an IPv4 PDU

What is a struct in C?

A struct in C can be used for several purposes. For network programming it is often used for serialization and deserialization to raw byte buffers, which at the end of the day, is the format sent over the wire. Note that structures are a convivence as you can live without structures and just do pointer manipulation.

Consider the below.

```
#define ETH_ALEN 6

typedef struct ether_pdu{
   uint8_t dest_addr[ETH_ALEN];
   uint8_t src_addr[ETH_ALEN];
   ube16_t frame_type;
} ether_pdu_t;

sizeof(ether_pdu_t); //  This is always going to be 14


uint8_t ethernet_frame = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
}
```

```
ether_pdu_t *ether = (ether_pdu_t *)ethernet_frame;

//This means that:
//ether->dest_addr == ethernet_frame;
//ether->src_addr == ethernet_frame + 6; //starts on byte 6
//ether->frame_type == ethernet_frame + 12// starts byte 12
```

Going back and forth between structs and byte buffers – Struct to Bytes

A struct in C can be used for several purposes. For network programming it is often used for serialization and Consider the below.

```
#define ETH_ALEN 6

typedef struct ether_pdu{
  uint8_t dest_addr[ETH_ALEN];
  uint8_t src_addr[ETH_ALEN];
  ube16_t frame_type;
} ether_pdu_t;

sizeof(ether_pdu_t); //  This is always going to be 14
```

```
uint8_t ethernet_frame[] = {
0xa0, 0x36, 0xbc, 0x62,
0xed, 0x50, 0xc8, 0x89,
0xf3, 0xea, 0x93, 0x14,
0x08, 0x00
}
```

```
uint8_t dst[] = {0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50};
uint8_t src[] = {0xc8, 0x89, 0xf3, 0xea, 0x93, 0x14};

ether_pdu epdu;

memcpy(epdu.dest_addr, dst, sizeof(dst));
memcpy(epdu.src, src, sizeof(src);
epdu.frame_type = 0x0800;
```

Now you can do:

```
uint8_t *ether_frame = (uint_8 *)epdu;
```

Going back and forth between structs and byte buffers – Bytes to Struct

A struct in C can be used for several purposes. For network programming it is often used for serialization and Consider the below.

```
#define ETH_ALEN 6

typedef struct ether_pdu{
  uint8_t dest_addr[ETH_ALEN];
  uint8_t src_addr[ETH_ALEN];
  ube16_t frame_type;
} ether_pdu_t;

sizeof(ether_pdu_t); //  This is always going to be 14


uint8_t ethernet_frame[] = {
0xa0, 0x36, 0xbc, 0x62,
0xed, 0x50, 0xc8, 0x89,
0xf3, 0xea, 0x93, 0x14,
0x08, 0x00
}
```

```
ether_pdu *epdu = (ether_pdu *)ethernet_frame;

//Now you can access each field directly
//  epdu->dest_addr
//  epdu->src_addr
//  epdu->frame_type
```

Since C allows you to convert anything to anything else (for the most part) you can just overlay a pointer to a structure on a buffer and get working.

Structure padding, sometimes we need to turn this off…

From packet.h, there are a few structures that are tagged with __attribute__() directives, this tells the gcc compiler not to do something:

```
typedef struct __attribute__((packed)) icmp_echo_packet {
    ip_packet_t ip;
    icmp_echo_pdu_t icmp_echo_hdr;
    uint8_t icmp_payload[];
} icmp_echo_packet_t;
```

Lets consider a simpler example:

```
typedef struct {
    char a;
    int b;
    long c;
} icmp_echo_packet_t;
```

→

```
typedef struct {
    char a;
    char padding;
    int b;
    long c;
} icmp_echo_packet_t;
```

Most compilers will add a padding byte after the initial character. We don't want b to start on offset 3, and c to start on offset 5

Think about why?

However, we must have our network structures aligned to exact bytes so therefore we pack them

C Types – char, int, short, long, long long? We will not be using these…

When we work with network structures we are working with collection of bytes. The standard C types are great but you can get yourself into trouble. For example, is a long 32 or 64 bits? Correct answer – It depends….

Most modern compilers char=1 byte; short=2 bytes, int= 4 bytes; long and long long = 8 bytes
BUT THIS IS NOT GUARENTEED!

Since network structures are often terms defined in terms of bytes we will be using types defined in #include<stdint.h> that are always guaranteed to have the same size:

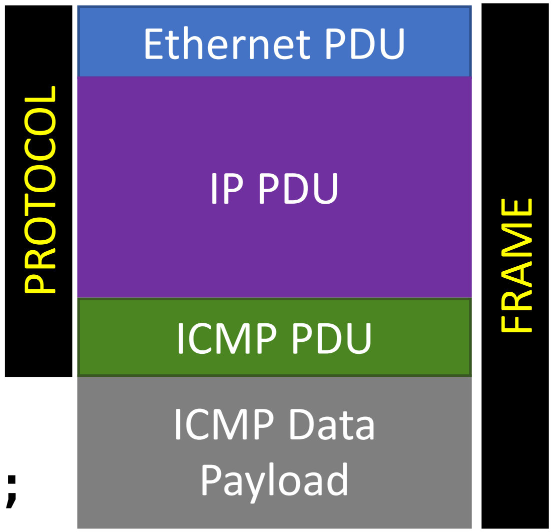
```
uint8_t
uint16_t
uint32_t
uint64_t
```

Sometimes we even work in units smaller than a byte

In this example, the IP PDU (in purple) is 20 bytes. That is the most common, but there are also other variants that add options at the end making the PDU longer. How do we tell?

```
uint8_t raw_packet_icmp_frame362[] = {
0xa0, 0x36, 0xbc, 0x62, 0xed, 0x50, 0xc8,
0x89, 0xf3, 0xea, 0x93, 0x14, 0x08, 0x00,
0x45, 0x00, 0x00, 0x54, 0x2a, 0xec, 0x00,
0x00, 0x40, 0x01, 0x89, 0x31, 0xc0, 0xa8,
0x32, 0x63, 0x90, 0x76, 0x43, 0x0a, 0x08,
0x00, 0x7b, 0xda, 0x48, 0x59, 0x00, 0x00,
0x65, 0x0e, 0x01, 0xee, 0x00, 0x00, 0xe1,
0xcc, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22,
0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29,
0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,
0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
};
```

This byte is telling us 2 different things.




```
struct ip_pdu{
  uint8_t version:4;
  uint8_t header_len:4;
  ...
}
```

The first 4 bits are the IP version, or 4 in this case

The next 4 bits are the size of the IP_PDU in words (32 bits).
So 5 * 4 words, each word is 32 bits or 4 bytes => 5 * 4 = 20

The notation u_int8_t version:4 is called a bitfield

Why do we need to know where a PDU ends?