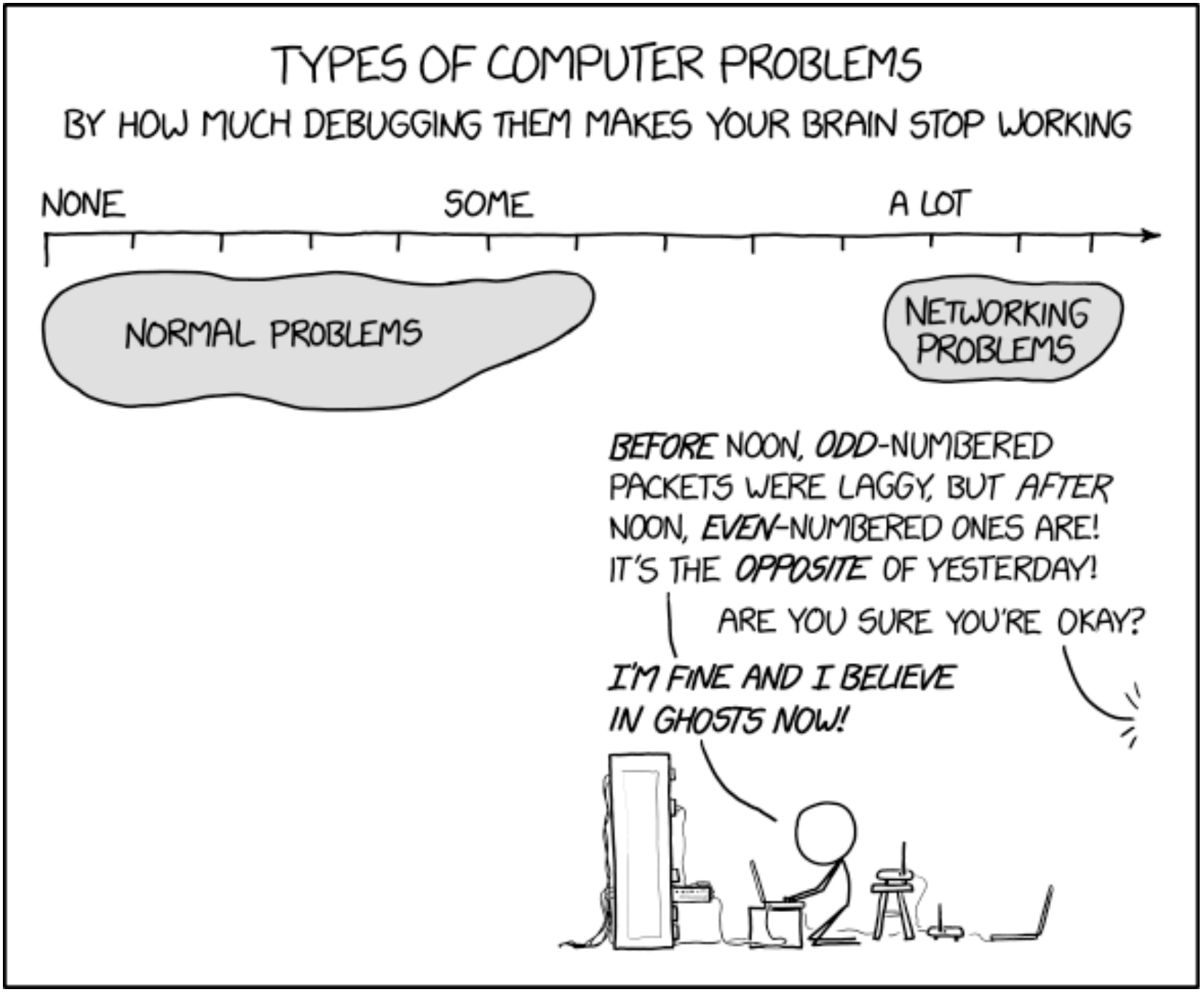
CS 55:
Security and Privacy

Packet sniffing and spoofing



https://xkcd.com/2259

Agenda

1. Network basics
2. Sending and receiving packets
3. Sniffing packets
4. Spoofing packets

The TCP/IP protocol stack uses layers to transmit data over unreliable networks

Conceptual network layers

I'll refer to this as the "protocol stack"

OSI model has 7 layers (application called layer 7)

Application	Interacts with application programs Allows applications to access network resources Interpretation of data is outside the scope of this layer Examples: HTTP, SSH, FTP, SMTP
Transport	Provides host-to-host message delivery in message packets May provide error control, flow control, application addressing (ports) Examples: TCP (connection-oriented), UDP (connectionless) TCP provides sequencing, dropped packet resend, traffic congestion routing
Network (Internet)	Moves packets (datagrams) across local area network boundaries (routing) Each computer identified by an IP address (IP v4 or v6) Also called Layer 3 or IP layer. (ICMP is here)
Link (Data Link)	Moves frames within a local area network (switching) Each computer identified by a MAC address Also called Layer 2, MAC layer, or Ethernet layer
Physical	How data is physically transmitted • Transmitter converts logical 1's and 0's to electrical/light pulses or phase/amplitude of radio frequency (RF) and sends down wire or over air • Receiver converts electrical/light or RF back to logical 1's and 0's

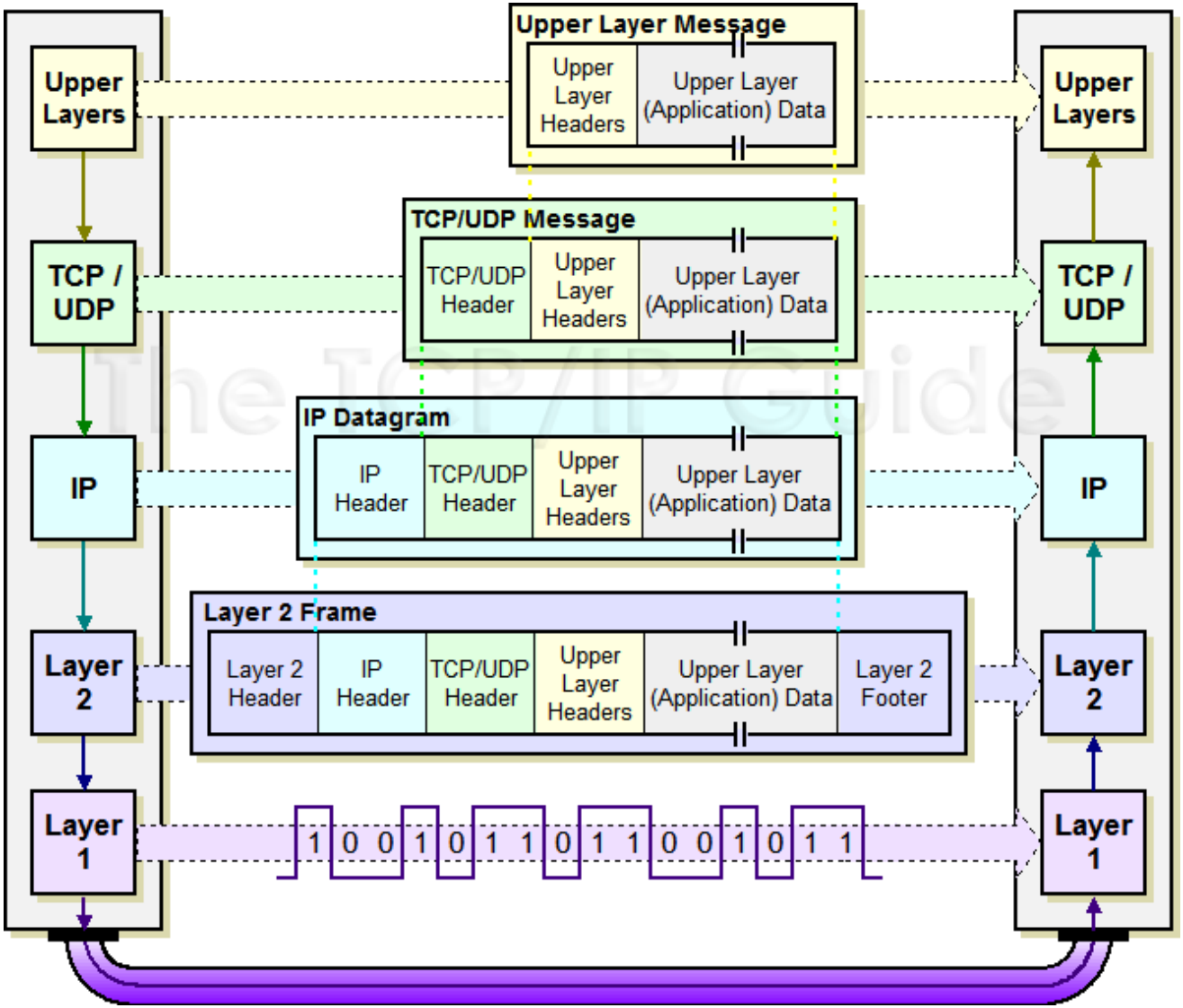
Adapted from https://www.guru99.com/tcp-ip-model.htm

Messages travel down the stack from the transmitter and up the stack at the receiver

Transmitter
Wants to send data to receiver

Data starts at top and is encapsulated by each layer down the stack

Each layer adds it own headers

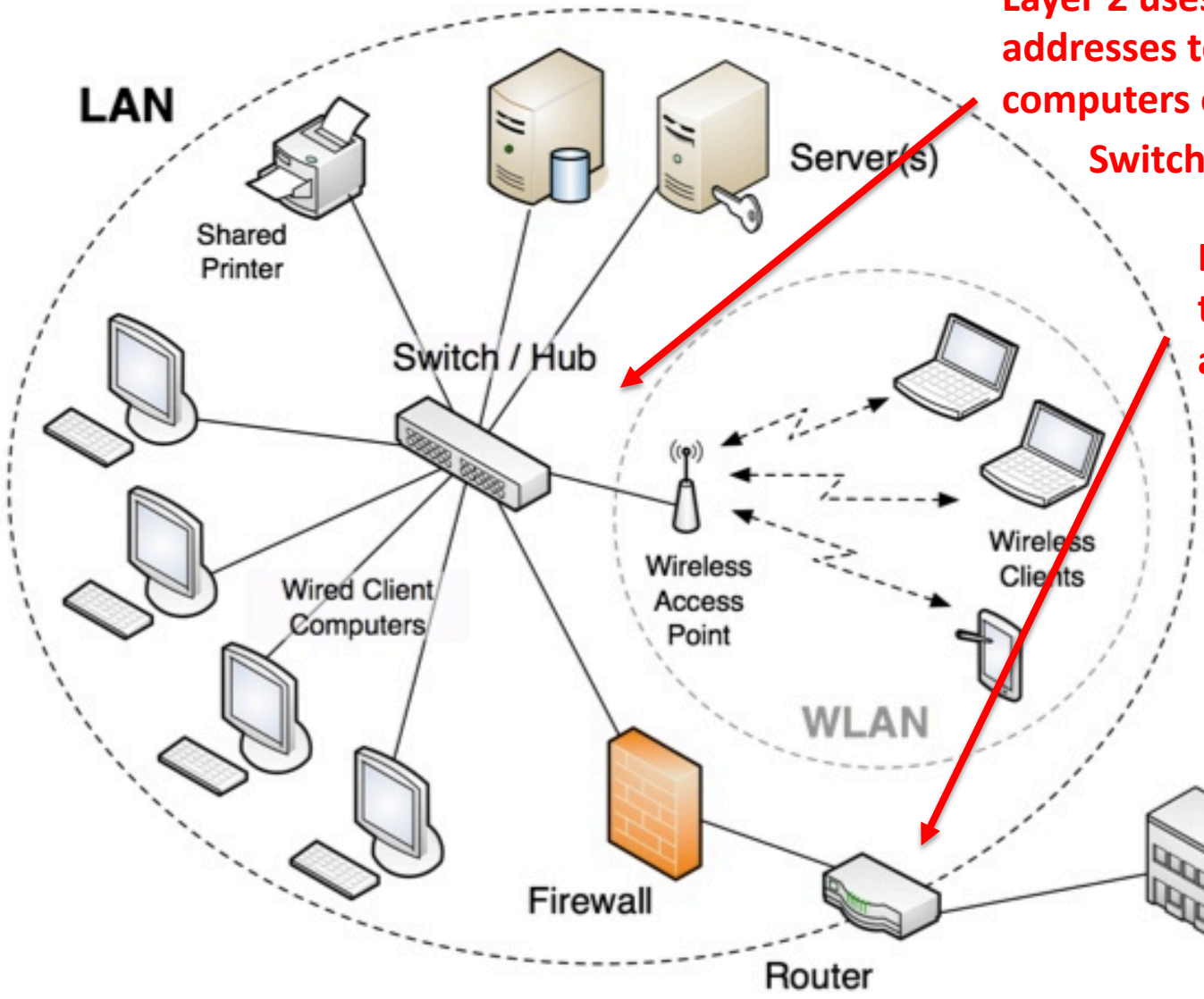


Receiver
Data starts at the physical layer and moves upward

Each layer strips off headers from prior layer

Adapted from http://www.tcpipguide.com/free/t_IPDatagramEncapsulation.htm

Small network example shows how components work together



Layer 2 uses MAC addresses to identify computers on a LAN

Switch operates at layer 2

Layer 3 uses IP addresses to identify computers across LANs

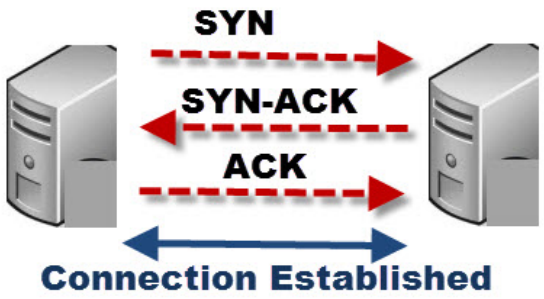
Router operates at layer 3

One device often plays the role of switch and router

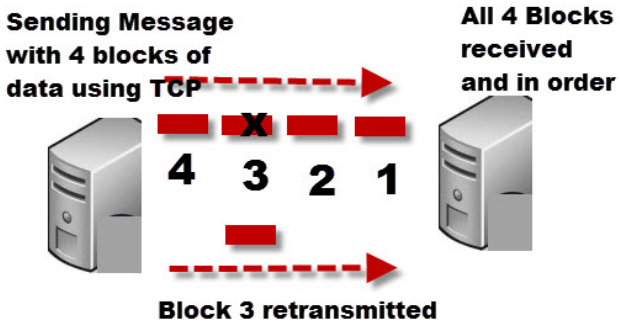
https://www.igcseict.info/theory/4/netsetup/index.html

At the Transport Layer, TCP is connection-oriented, UDP is not

Transmission Control Protocol (TCP)



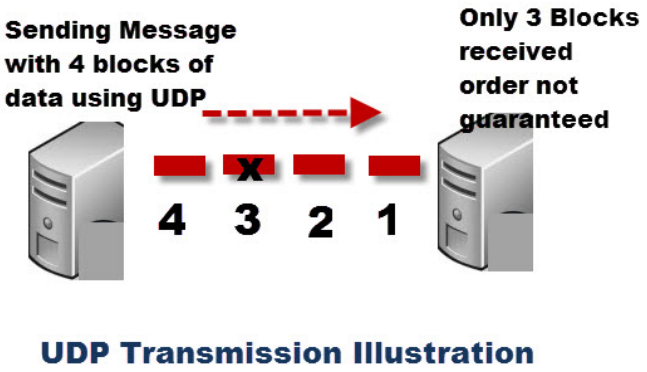
TCP establishes a connection between machines with a 3-way handshake



- Messages are broken into packets
- Each packet given sequence number
- Packets reassembled in seq order
- Missing packets are resent

User Datagram Protocol (UDP)

UDP does not establish a connection



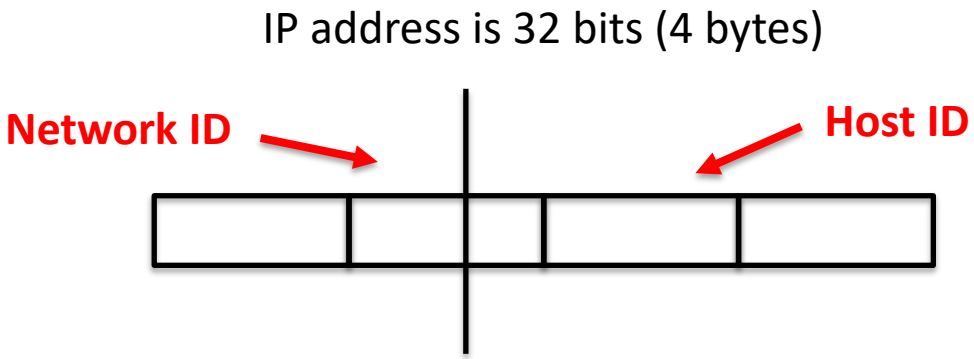
UDP Transmission Illustration

- Messages are broken into packets
- Packet received or not, no resend
- Faster than TCP, smaller headers
- Use for streaming

Adapted from http://www.steves-internet-guide.com/tcp-vs-udp/

IP addresses contain the network and device ID

Classless Interdomain Routing (CIDR, 1993+)



Split wherever we'd like

- Specify how many bits make up the network ID, the rest of the address specifies a device on that network
- 128.230.115.0/24 means left 24 bits are network ID (8 bits are used to identify a device on that network, ranging from 0 to 255)
- Split does not have to occur on 8-bit boundaries
- 128.230.0.0/9 means left most 9 bits are the network ID

Some IP addresses are reserved for private use

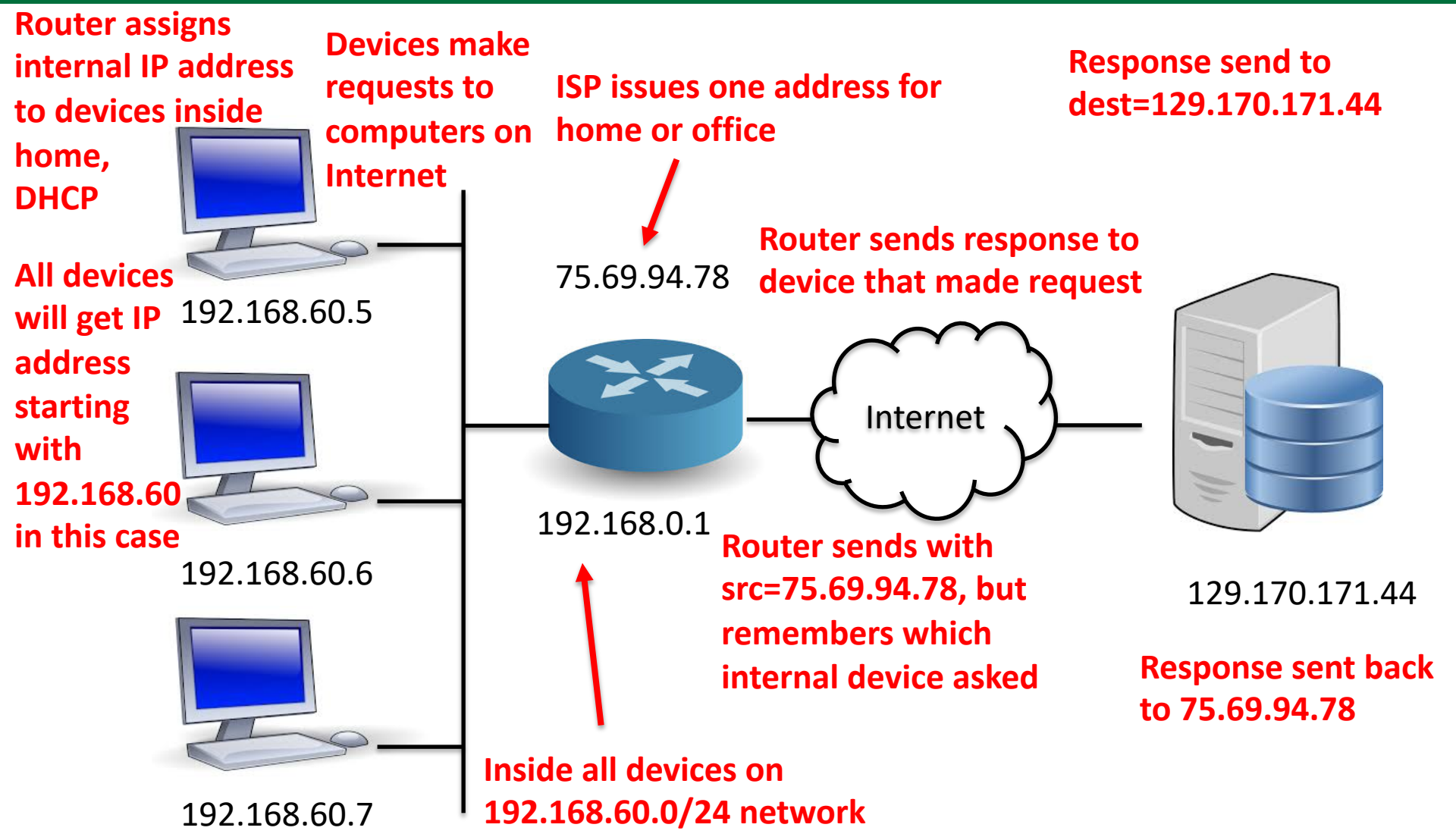
Private IP address
10.0.0.0/8
172.16.0.0/12
192.168.0.0/16

ISP issues a single IP address
Use private network for devices in home

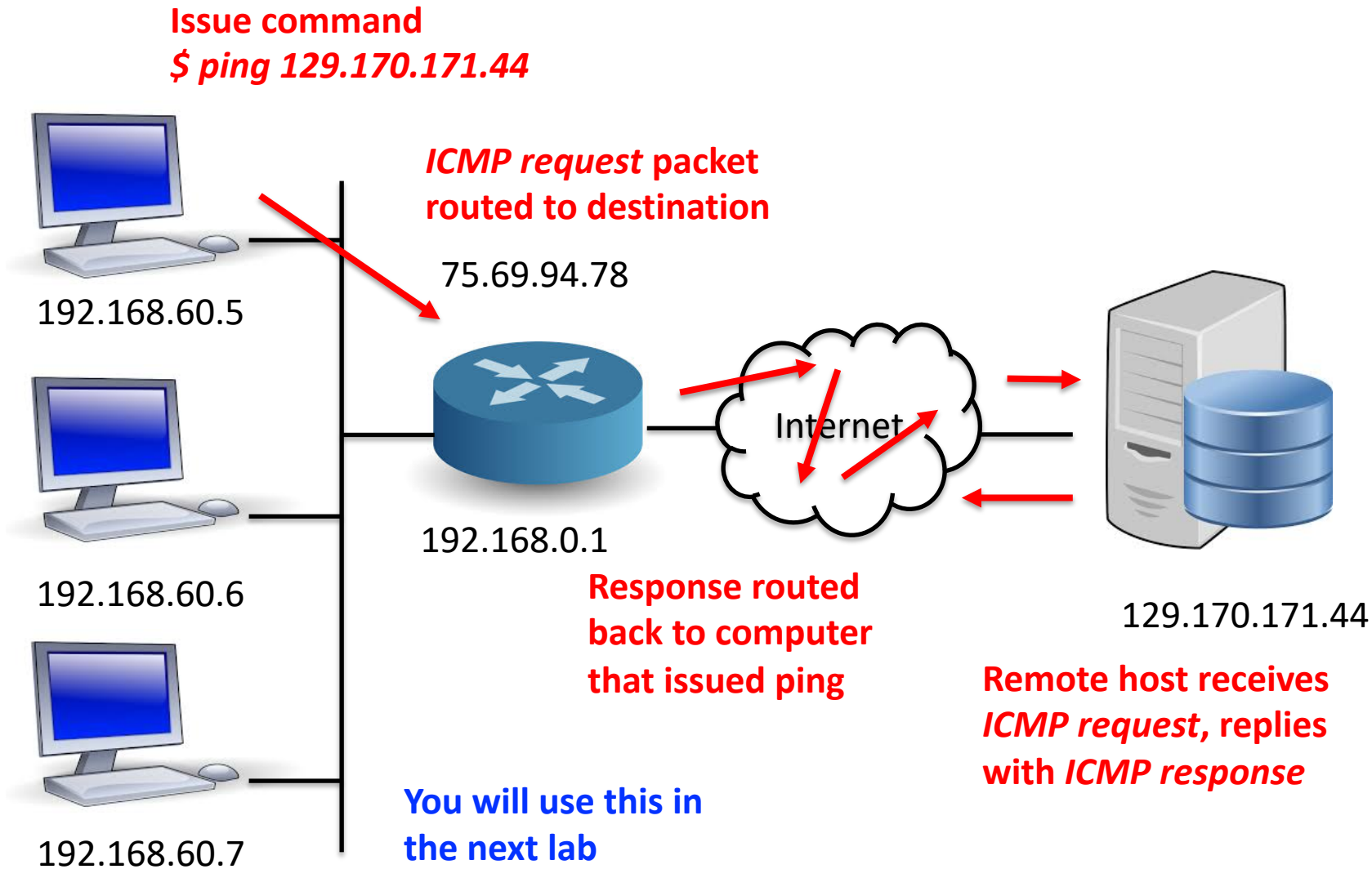
Non-routable on Internet (multiple homes may use the same address inside)

Lookback address
127.0.0.0/8
Commonly used 127.0.0.1 (localhost)

Network Address Translation (NAT) handles translation between inside and outside



The *ping* command is useful to see if a computer is reachable (and on homework)



Domain Name System (DNS) converts between IP addresses and names

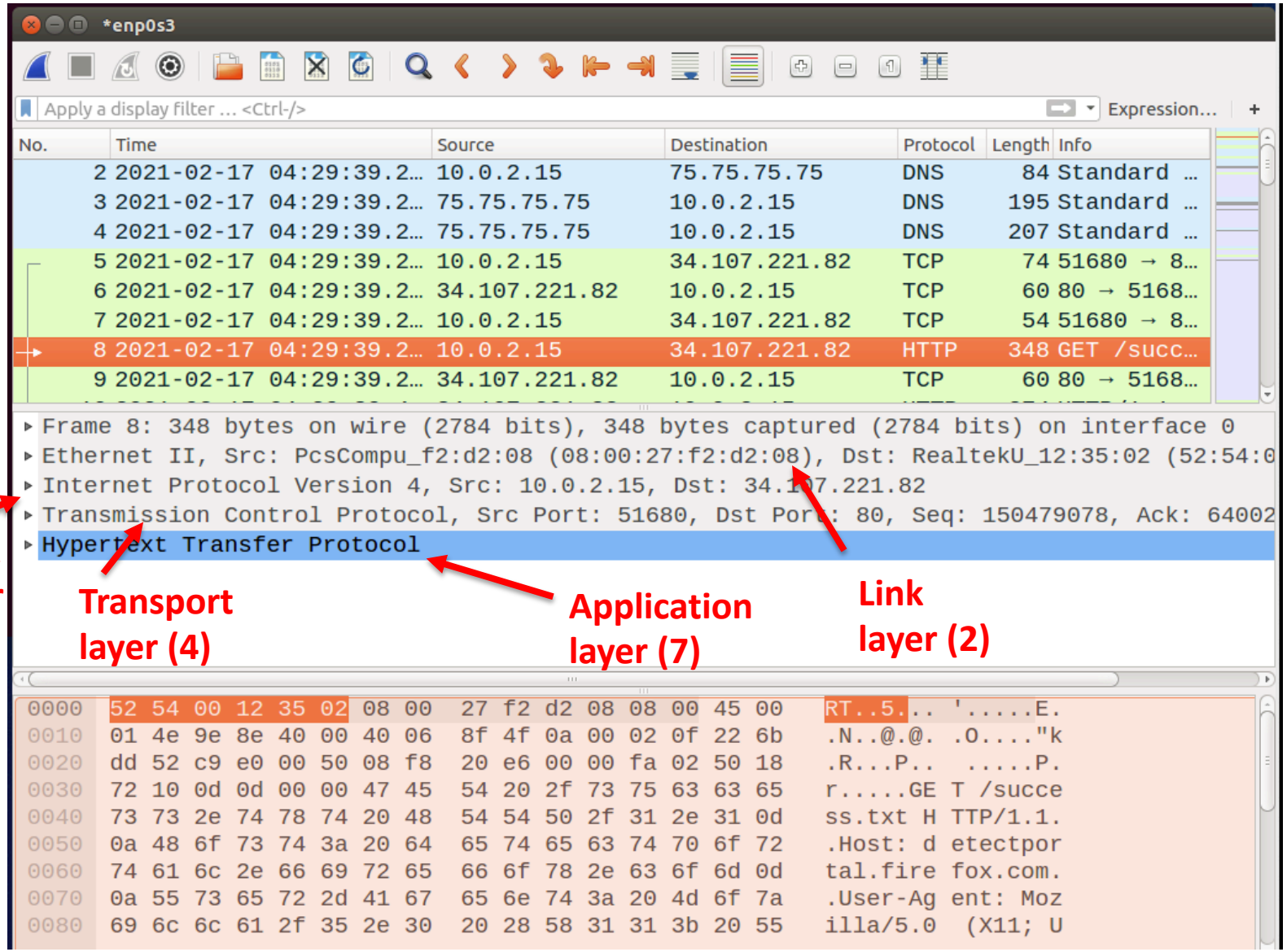
Name	IP
www.google.com	172.217.3.110
www.dartmouth.edu	129.170.171.44

Humans are not good at remembering numeric IP addresses

Give a computers a name humans can remember

DNS matches name with IP address

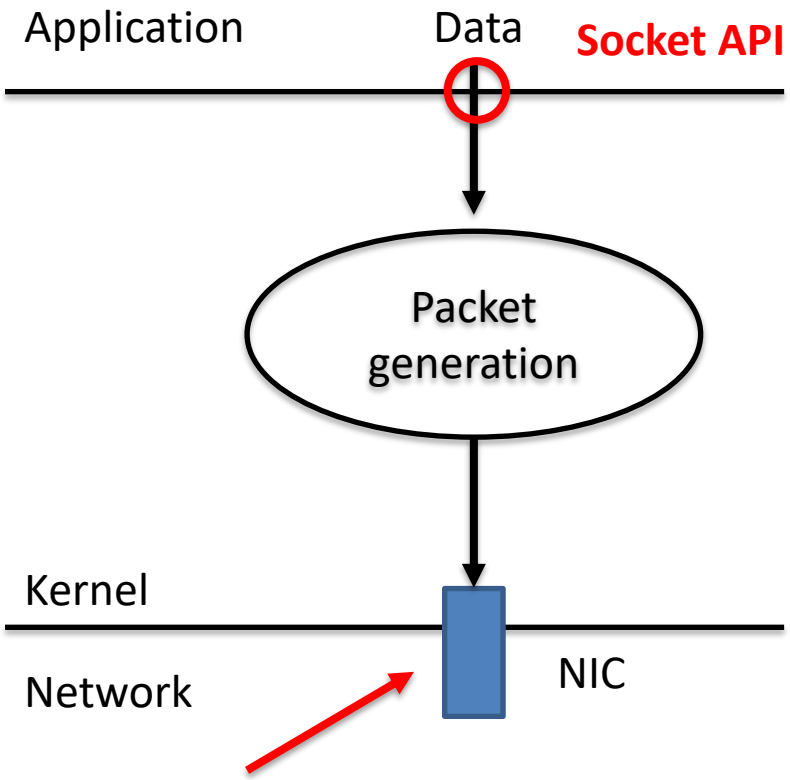
Demo: Wireshark shows the protocol stack in action



Agenda

1. Network basics
2. Sending and receiving packets
3. Sniffing packets
4. Spoofing packets

Typically, the OS handles packet construction



Application has data that it wants to send

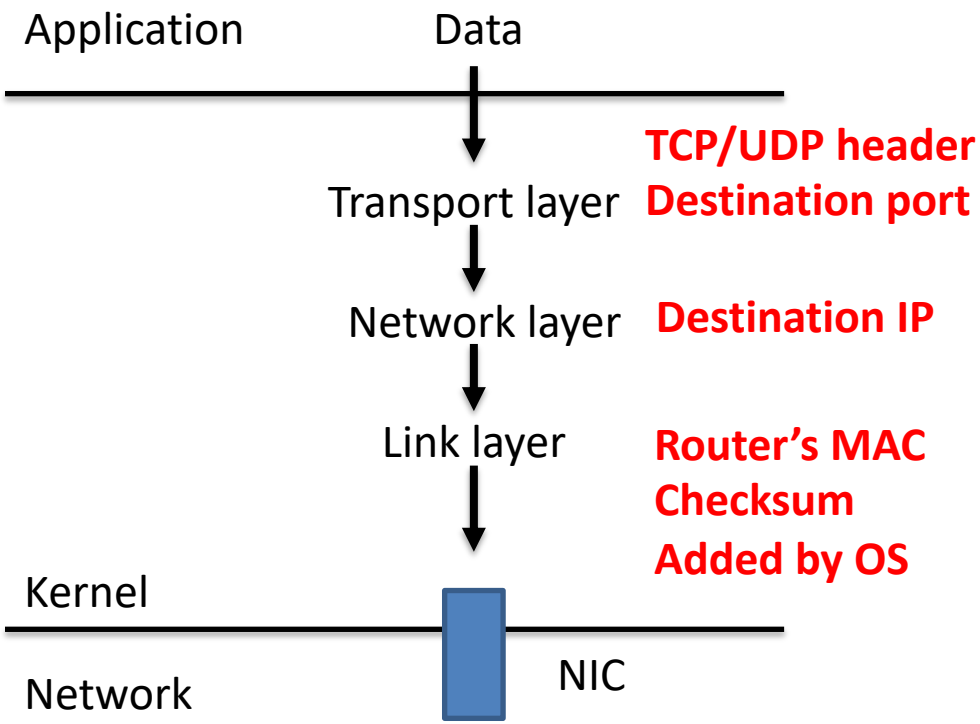
Passes data to OS which generates a packet

Packet sent out over network via NIC

How does the application hand data to OS?
OS provides socket API system calls
Recall sockets from CS10 and 50

Each NIC has unique MAC address
Sees all traffic on network
Drops packets not addresses to itself (unless promiscuous or monitor mode)

Applications must provide protocol and destination information



You must decide which transport protocol to use:

- UDP – faster, but delivery not guaranteed
- TCP – slower but delivery guaranteed

You must also provide destination information

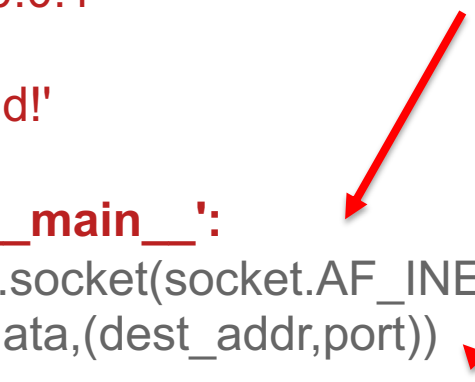
- IP address
- Port

OS adds router's MAC and checksum

Sending a packet is not difficult with Python

send_udp.py

```
import socket

dest_addr = "127.0.0.1"
port = 9090
data = b'Hello world!'

if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(data,(dest_addr,port))
```



Sending a packet is not difficult with Python

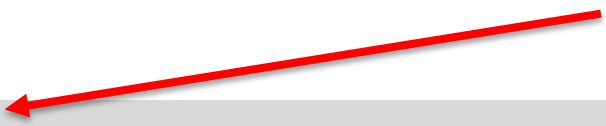
send_udp.py

```
import socket

dest_addr = "127.0.0.1"
port = 9090
data = b'Hello world!'

if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(data,(dest_addr,port))
```

Python hides a lot of complexity!

Note: we do not set a source port
Will see soon the OS picks one for us

Listen in another terminal on port 9090 using netcat



```
$ nc -luv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Hello world!^C
```

Sending a packet in C is a little more involved but gives you more control

send_udp.c

Python built on top of C libraries

```
void main() {
        struct sockaddr_in dest_info;
        char * dest_addr = "127.0.0.1";
        int port = 9090;
        char *data = "Hello World (in C!)\n";

        //Create network socket
        int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

        //Provide needed data
        memset((char *) &dest_info,0,sizeof(dest_info));
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr.s_addr = inet_addr(dest_addr);
        dest_info.sin_port = htons(port);

        //send packet
        sendto(sock, data, strlen(data), 0, (struct sockaddr *) &dest_info,
                                        sizeof(dest_info));
        close(sock);
}
```

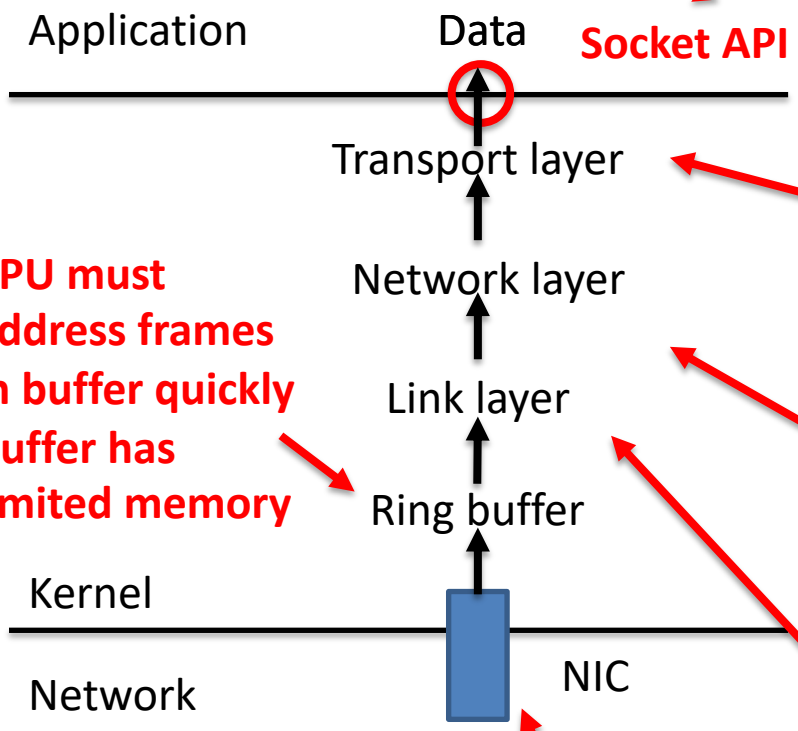
Create socket for UDP

Use IPv4 and send to dest_addr

Send

Receiving a packet is the reverse of sending a packet

Layer 3 overview



Data passed to application via socket
Socket is bound to port
Application blocks until packet arrives
Kernel wakes up application
Application can read buffer

Transport layer looks at port, fills buffer, and sends to socket listening on that port

CPU must address frames in buffer quickly
Buffer has limited memory

Packet received, looks for match with own IP address

- **Drop if no match**
- **Otherwise look at protocol and pass to appropriate Transport layer (UDP or TCP)**

Link layer (layer 2) driver takes frame from ring buffer
Strips off link-layer header

NIC looks at incoming frames
If addressed to this MAC address, DMA to kernel ring buffer
Interrupts CPU, informs CPU frame has arrived

Packets can be easily received in Python

receive_udp.py

```
import socket

ip_addr = "0.0.0.0"
port = 9090

if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))

    while (True):
        data, (ip, port) = sock.recvfrom(1024)
        print("Sender: {} and Port: {}".format(ip,port))
        print("Received message: {}".format(data))
```

Computer might have multiple NICs
Zeros means accept packets from any of them

Set up socket for UDP as before

Bind socket to listen on ip_addr and port

Block until packet arrives
recvfrom gets data then
print packet

```
$ nc –u 127.0.0.1 9090
hello world
```

Use netcat in another terminal to send UDP packets to IP address on port 9090

Type in messages to send, sends after return key pressed

OS picks a source port for the sender

receive_udp.py

```
import socket

ip_addr = "0.0.0.0"
port = 9090

if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))

    while (True):
        data, (ip, port) = sock.recvfrom(1024)
        print("Sender: {} and Port: {}".format(ip,port))
        print("Received message: {}".format(data))
```

We did not pick a source port on the sender (but did pick destination port!)
OS chooses one randomly for you
When this is run, see source port other than 9090
This way OS can sort out replies if multiple instances of application are running at the same time

```
$ nc –u 127.0.0.1 9090
hello world (in C!)
```

A receiver written in C is not too different from one written in Python

receive_udp.c

```
const int MAX_SIZE = 1500;
const int port = 9090;

void main() {
   struct sockaddr_in server;
   struct sockaddr_in client;
   int clientlen;
   char buf[MAX_SIZE];


   // Create the socket
   int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   memset((char *) &server, 0, sizeof(server));
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = htonl(INADDR_ANY);
   server.sin_port = htons(port);

   if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0) {
      printf ("Binding error!");
      return;
   }

   // Getting captured packets
   while (1) {
      bzero(buf,MAX_SIZE);
      recvfrom(sock, buf, MAX_SIZE-1, 0, (struct sockaddr *) &client, &clientlen);
      printf("%s\n",buf);

   }

   close(sock);
```

Bind to every interface (like Python with "0.0.0.0")

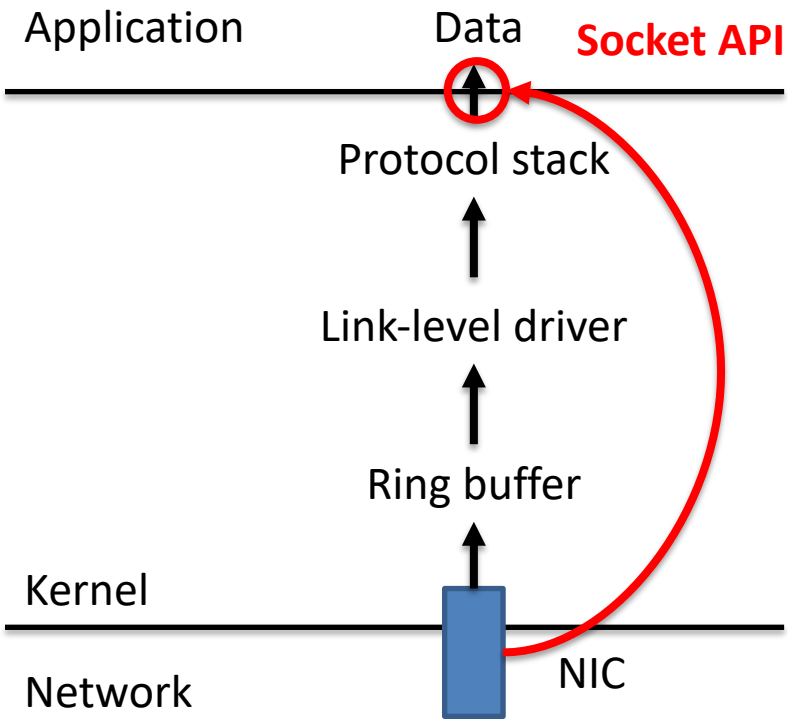
Bind to IP address (any interface here) and port

Client info filled when packets arrive

Agenda

1. Network basics
2. Sending and receiving packets
3. Sniffing packets
4. Spoofing packets

Promiscuous (monitor) mode sniffs all frames



Monitor mode only sniffs one channel

Note: some NIC's will not go into promiscuous mode (but most will)

Raw socket **delivers a copy of frame including link-layer header directly to the socket**

Promiscuous mode (monitor mode) will pass at *all* frames from NIC

NIC looks at incoming frames
If address to this MAC address, DMA to kernel ring buffer
Interrupts CPU, informs CPU frame has arrived

Sniffing all frames can take a lot of CPU time!

sniff_raw.c

```
int main() {
    int PACKET_LEN = 512;
    char buffer[PACKET_LEN];
    struct sockaddr saddr;
    struct packet_mreq mr;

    // Create the raw socket
    int sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

    // Turn on the promiscuous mode.
    mr.mr_type = PACKET_MR_PROMISC;
    setsockopt(sock, SOL_PACKET, PACKET_ADD_MEMBERSHIP, &mr, sizeof(mr));

    // Getting captured packets
    while (1) {
        int data_size=recvfrom(sock, buffer, PACKET_LEN, 0,
                    &saddr, (socklen_t*)sizeof(saddr));
        if(data_size) printf("Got one packet\n");
    }

    close(sock);
    return 0;
}
```

Sniff raw frames

Sniff all protocols (set to ETH_P_IP for only IP layer 3)

Turn on promiscuous mode to see all frames

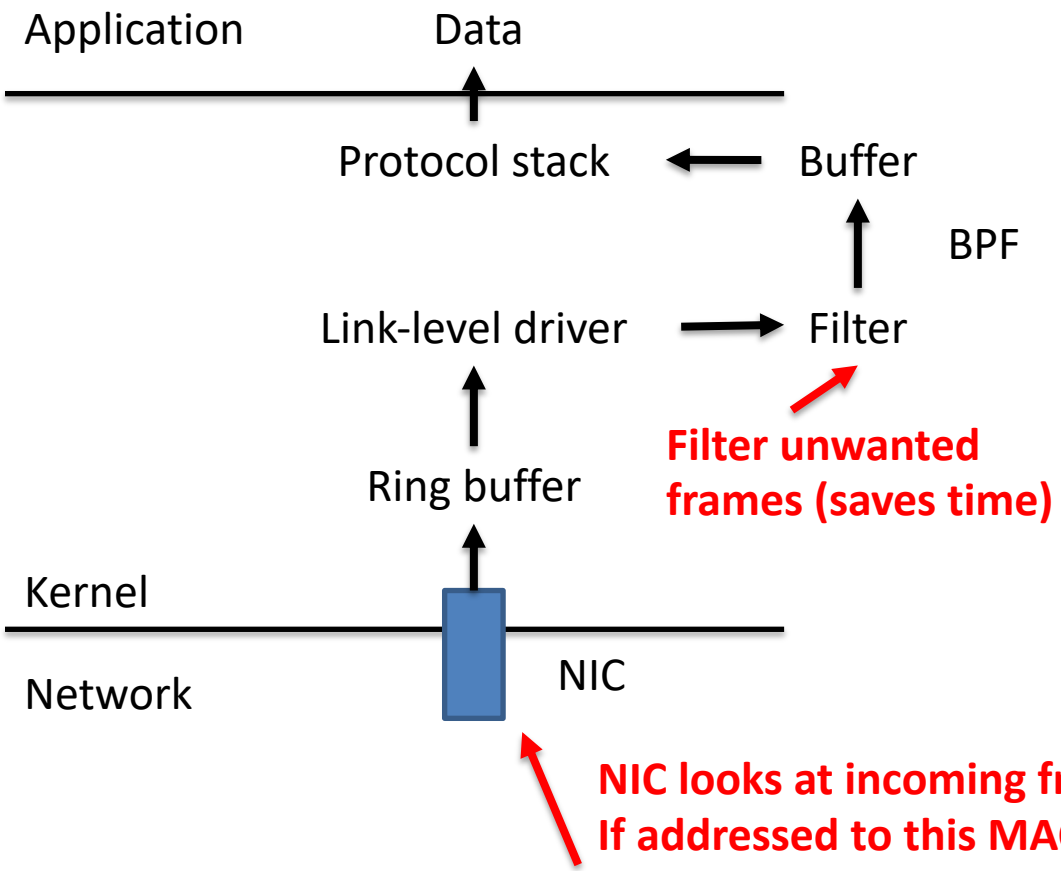
Set socket option enables promiscuous mode

Now see all layer 2 frames

Be careful, this may be very busy if you are on a high-traffic network!

Most of the time we will want to filter out unwanted frames to reduce processing

Layer 2 overview



Berkeley Packet Filter (BPF)

- Creates filter at link-level
- Only passes frames matching criteria to protocol stack
- Does not DMA to memory
- Compile filter and set with setsockopt

BPF is somewhat complicated
Add to raw socket with SO_ATTACH_FILTER
Not portable between OSes
PCAP API is much easier!

PCAP makes it easy to filter frames at a low level

PCAP (packet capture)

- Originally written for tcpdump (powerful sniffer)
- Supported by multiple platforms
  - Linux: libpcap
  - Windows: WinPcap and Npcap
- Written in C
- Other languages generally provide a wrapper around C version
- Basis used by other sniffers
  - Tcpdump (of course)
  - Wireshark
  - Scapy
  - Nmap
  - Snort

Use PCAP and compile BPF filter to filter out unwanted frames

```
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
   printf("Got a packet\n");
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "udp or icmp";
  bpf_u_int32 net;
  const int MAX_SIZE = 8192;

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", MAX_SIZE, 1, 1000, errbuf);
  if (handle == NULL) { printf("Error on open\n"); printf("errbuf %s\n",errbuf);  return (1); }

  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  if (pcap_setfilter(handle, &fp) != 0) { printf("set filter error"); return(1); }

  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);

  pcap_close(handle);   //Close the handle
  return 0;
```

Prints got packet when UDP or ICMP packet arrives
Last parameter has packet details

Parsing packets in C is tedious

Open PCAP session

- Sniff on interface *enp0s3* (use ifconfig to find)
- 3rd parameter sets promiscuous mode to true

Compile BPF filter to pass UDP or ICMP packets

Start sniffing
Set callback function to *got_packet*

Scapy makes it easy to use PCAP from Python to sniff packets with a filter

sniff_scapy.py

Scapy built on top of PCAP

Install scapy on VM

$ sudo pip3 install scapy

First install scapy on VM

```
from scapy.all import *

def process_packet(pkt):
        global count
        print('-'*40)
        print("Packet number",count)
        pkt.show()
        print('-'*40)
        count += 1

if __name__ == '__main__':
        count = 0
        pkt = sniff(iface='enp0s3', filter='icmp or udp', count=10, prn=process_packet)
```

Called whenever packet arrives

Print packet summary (much easier to process packets than with C)

Callback function when packet arrives

Only capture 10 packets

Sniff on this interface

Filter out packets other than these

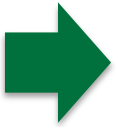
Run

$ sudo python3 sniff_scapy.py

Uses promiscuous mode, so need sudo privilege

Agenda

1. Network basics
2. Sending and receiving packets
3. Sniffing packets
4. Spoofing packets



Demo: packet spoofing is easy with Scapy

udp_spoof.py

False flag: it looks like this packet came from another computer!

```
#!/usr/bin/python3
from scapy.all import *

print("SENDING SPOOFED UDP PACKET.....")
ip = IP(src="1.2.3.4", dst="10.0.2.5") # IP Layer
udp = UDP(sport=8888, dport=9090)      # UDP Layer
data = "Hello UDP!\n"                  # Payload
pkt = ip/udp/data     # Construct the complete packet
pkt.show()
send(pkt,verbose=0)
```

Demo

- Start secondary VM with address 10.0.2.5 (confirm with ifconfig)
- Start Wireshark on secondary VM
- Run this code from primary VM (sudo python3 udp_spoof.py)
- Wireshark shows packet came from 1.2.3.4!

Create IP layer 3 with spoofed src and dest

Set UDP (layer 4) source and destination ports to destination

Stack layers to create UDP over IP layer with data

Send packet

Scapy fills in default values and calculates checksums for us! Nice!

We could do this in C also, but tedious to set packet values

Sometimes we want to sniff, then spoof a reply

sniff_spoof_icmp.py

```
from scapy.all import *

def spoof_pkt(pkt):
  if ICMP in pkt and pkt[ICMP].type == 8:
    #listen for ICMP request packets (type 8)
    print("Original Packet.....")
    print("Source IP : ", pkt[IP].src)
    print("Destination IP :", pkt[IP].dst)

    #spoof a reply, even if the request wasn't for us
    #must reverse source and destination on reply!
    ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
    icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
    data = pkt[Raw].load
    newpkt = ip/icmp/data

    print("Spoofed Packet.....")
    print("Source IP : ", newpkt[IP].src)
    print("Destination IP :", newpkt[IP].dst)

    send(newpkt,verbose=0)

pkt = sniff(filter='icmp and src host 10.0.2.15',prn=spoof_pkt) #icmp and src host 10.0.2.15
```

Demo

- Start Wireshark on secondary VM
- Run this code from primary VM
  (sudo python3 sniff_spoof_icmp.py)
- From secondary VM ping 10.0.2.6 (doesn't exist)
- Wireshark shows packet came from 10.0.2.6

Steps

- **Receive packet**
- **Extract details**
- **Spoof a new packet using extracted details**

We have created a phantom computer on the network that responds to pings from 10.0.2.15!

Scapy vs. C

Scapy	C
Pros • Easier to write • Sets reasonable default fields • Calculates values (checksums) • Focus on fields interested • Every layer is an object, can easily stack together • Increased productivity	**Pros** • Runs much faster! (50-100X!) • More control over crafting packets

Scapy vs. C

Scapy	C
Pros • Easier to write • Sets reasonable default fields • Calculates values (checksums) • Focus on fields interested • Every layer is an object, can easily stack together • Increased productivity	**Pros** • Runs much faster! (50-100X!) • More control over crafting packets
Cons • Runs slowly	**Cons** • Tricky to write, must get the byte offsets right • Tedious

Scapy and C can work together in a hybrid approach

Hybrid approach

For some uses speed is critical

Scapy is slow

Example:

Send 1000 UDP packets

- Scapy: 9.4 seconds
- C : 0.25 seconds
- C is 37X faster than Scapy in this case!

Sometimes racing to reply before the “real” computer – Scapy too slow

Idea:

- Use Scapy to create packets, and save to file
- Read packets in C (might to adjust some fields)
- Send using C

Endianness

Endianness: a term that refers to the order in which a given multi-byte data item is stored in memory

- **Little Endian**: store the most significant byte of data at the highest address
- **Big Endian**: store the most significant byte of data at the lowest address

Little endian always seems backwards to me

Store 0xDEADBEEF

Big endian

Increasing addresses →

Little endian

Endianness in network communication

Computers with different byte orders will misunderstand each other

- Solution: agree upon a common order for communication
- This is called “network order”, which is the same as Big Endian order
- But Intel computers (“hosts”) use Little Endian

- Must convert data between “host order” and “network order”

Macro	Description
`htons()`	Convert unsigned short integer from host order to network order.
`htonl()`	Convert unsigned integer from host order to network order.
`ntohs()`	Convert unsigned short integer from network order to host order.
`ntohl()`	Convert unsigned integer from network order to host order.

Prior to 1993 IP addresses were assigned in address range blocks

Classful addressing scheme (1981 -1993)

Class A: 0.0.0.0 – 127.255.255.255

Class B: 128.0.0.0 – 191.255.255.2555

Class C: 192.0.0.0 – 223.255.255.255

Class D: 224.0.0.0 – 239.255.255.255

Class E: 240.0.0.0 – 255.255.255.255

Used to help routing, 32 bits total

Left bit always 0, if see you a zero in the left bit, you know it belongs to a Class A network

Network ID is 2^7 = 127 possible address for first octet

If you had Class A network (first 8 bits set) you have 2^{24} = 16.7M possible addresses that can be given to devices

Issued to very large organizations

Prior to 1993 IP addresses were assigned in address range blocks

Classful addressing scheme (1981 -1993)

Class A: 0.0.0.0 – 127.255.255.255

Class B: 128.0.0.0 – 191.255.255.2555

Class C: 192.0.0.0 – 223.255.255.255

Class D: 224.0.0.0 – 239.255.255.255

Class E: 240.0.0.0 – 255.255.255.255

Used to help routing
32 bits total

Network ID is left 16 bits

- **Left bit = 1**
- **Left second bit = 0**

If you see left bits are 10, you know this is a Class B network

If you had Class B network (first 16 bits set) you have 2^{16} = 65.5K possible addresses that can be given to devices

Issued to large organizations

Prior to 1993 IP addresses were assigned in address range blocks

Classful addressing scheme (1981 -1993)

Class A: 0.0.0.0 – 127.255.255.255

Class B: 128.0.0.0 – 191.255.255.2555

Class C: 192.0.0.0 – 223.255.255.255

Class D: 224.0.0.0 – 239.255.255.255

Class E: 240.0.0.0 – 255.255.255.255

Used to help routing
32 bits total

Network ID is left 24 bits

- **Left bit = 1**
- **Left second bit = 1**
- **Third left bit = 0**

If you see left bits are 110, you know this is a Class C network

If you had Class C network (first 24 bits set) you have 2^8 = 256 possible addresses that can be given to devices

Issued to medium sized organizations

Prior to 1993 IP addresses were assigned in address range blocks

Classful addressing scheme (1981 -1993)

Class A: 0.0.0.0 – 127.255.255.255

Class B: 128.0.0.0 – 191.255.255.2555

Class C: 192.0.0.0 – 223.255.255.255

Class D: 224.0.0.0 – 239.255.255.255

Class E: 240.0.0.0 – 255.255.255.255

Used to help routing
32 bits total

Class D for multicast (special purpose)

Class E reserved

Problem:
We were using up IP addresses too quickly (lots of unused addresses if you own a Class A network)

Happened to us, we had a class C and had to give up some addresses