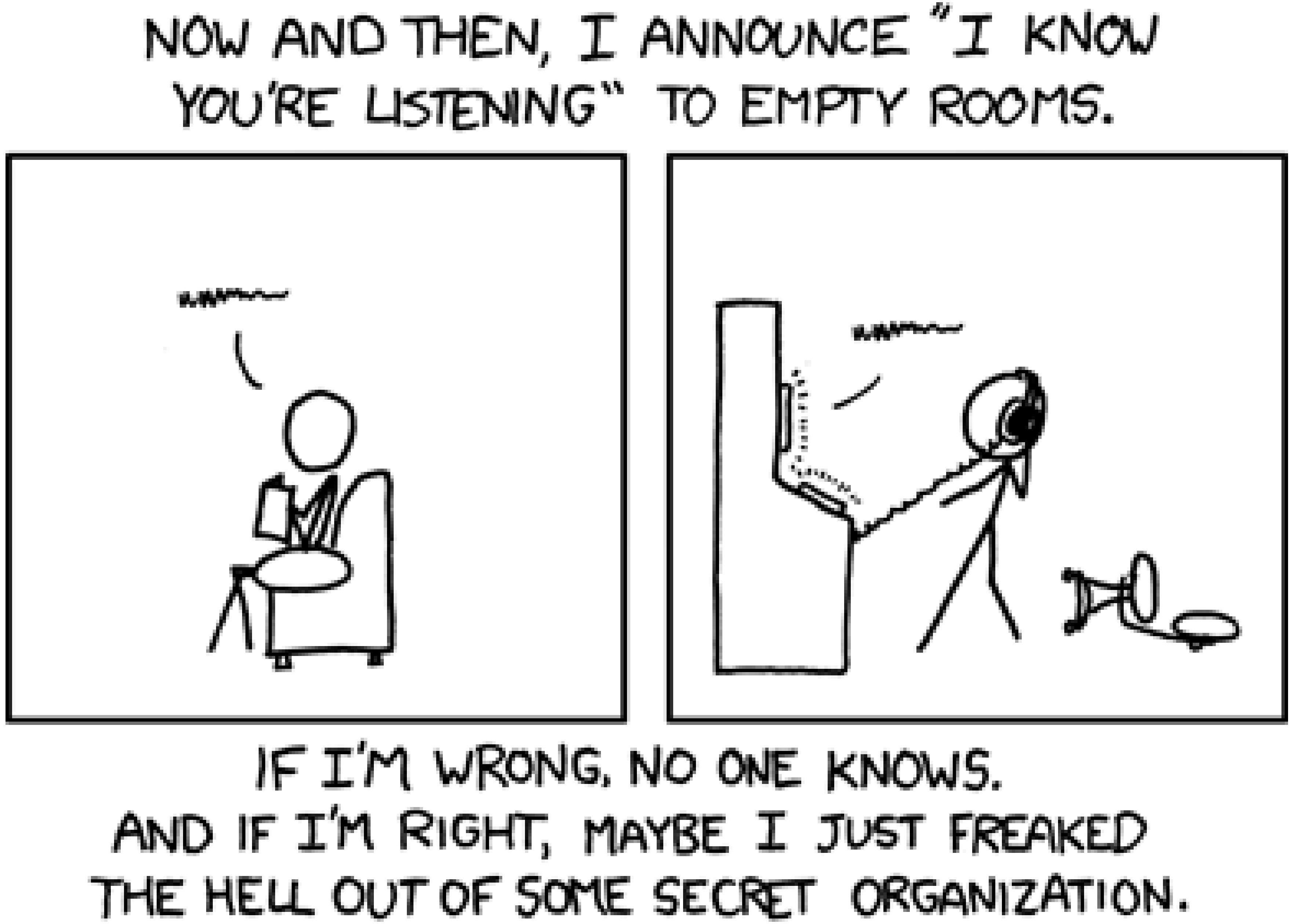


Source: https://xkcd.com/525/

CS 60:
Computer Networks

Packet sniffing and spoofing

Review from last class

1. All hosts on the Internet have an IP address
2. How does a host get an IP address?
   - Statically (manually) assigned by network administrator
   - DHCP server provides IP address from a pool of addresses
3. Network Interface Cards (NICs) have a unique Media Access Controller (MAC) address burned into ROM
4. Switches direct traffic on a LAN based on MAC address
5. How does a computer find the MAC of another computer?
   - ARP
6. What's left?
   - Routing between LANs
7. Before we get to routing, we'll take a closer look at higher layers then we will focus on creating and sniffing network traffic

Agenda

1. Review: network layers
2. Sockets
3. Scapy
4. Exercises

Review: network layers

Network model

Critical information

7) Application

Application data to be sent across network

Data

4) Transport

3) Network (IP)

2) Link (MAC)

1) Physical

Review: network layers

Network model

Critical information

7) Application

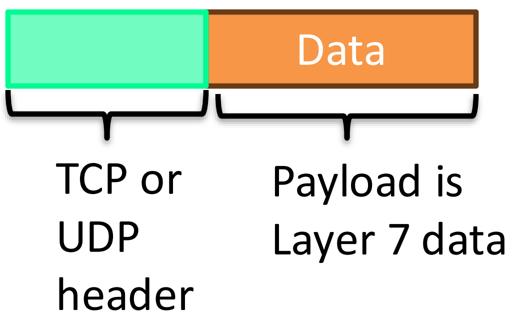
Application data to be sent across network

4) Transport

TCP or UDP

TCP is connection oriented, missing packets *are* resent

UDP is not connection oriented, missing packets are *not* resent



3) Network (IP)

2) Link (MAC)

1) Physical

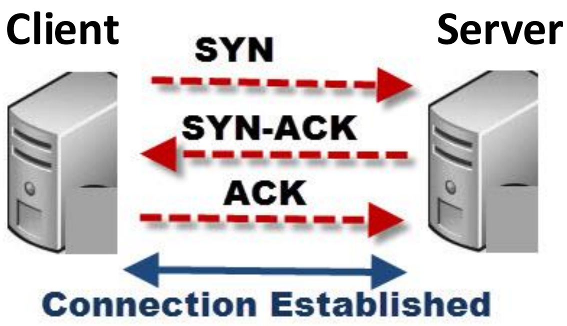
Demo: UDP vs TCP

Write message on pieces of paper
Send paper to a student volunteer

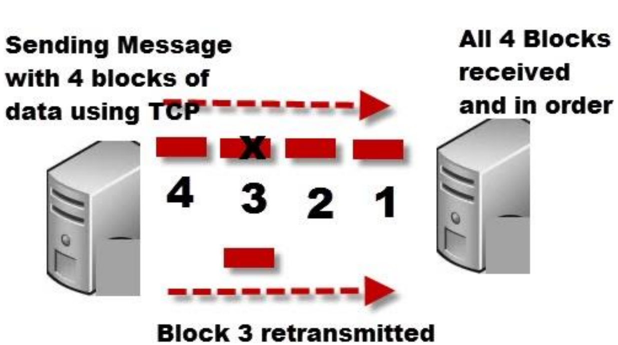
- UDP may drop packets, and they may also arrive out of order
- TCP will ACK packets to make sure missing packets are resent, and packets can be put in proper order

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

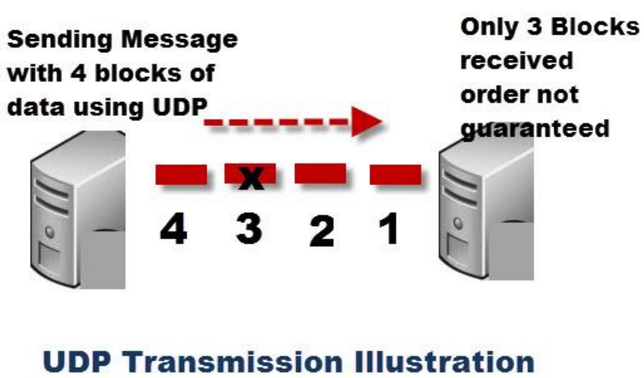


We will look at the transport layer in detail soon

- 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



- 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/

Layer 4 ports identify an application running on a host

Ports

Your computer is running multiple applications at the same time (email, web browsing, others)

Without a means to differentiate, all received data would go to the same place (e.g., web browser gets email traffic and vice versa)

Ports identify applications by a number that ranges from 0 to 65,535 = 2^{16}-1

Servers run applications that listen for connections on well-known ports

Clients connect to servers on these ports

Well known ports	
Port	**Service**
20 and 21	FTP
22	SSH
23	Telnet
53	DNS
80	HTTP
443	Encrypted HTTP
587 (old 25)	Email (SMTP)

Ports
0 - 1023 are well-known ports
Commonly reserved for system apps
1024-49,151 user or registered ports
49,152-65,535 ephemeral ports
See list at /etc/services

Review: network layers

Network model

Critical information

7) Application

Application message to be sent across network

4) Transport

Data

Payload is
Layer 7 data

3) Network (IP)

2) Link (MAC)

1) Physical

Review: network layers

Network model

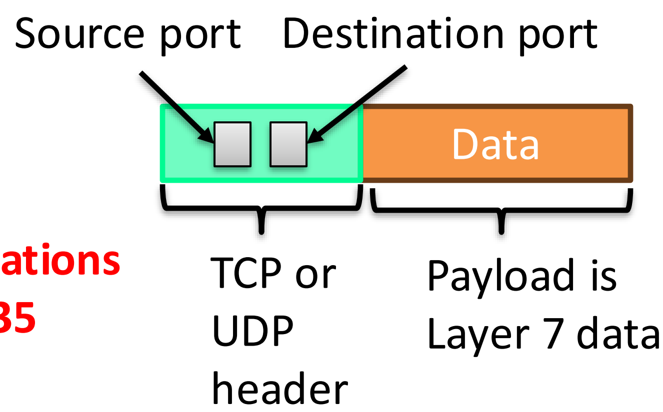
Critical information

7) Application

Application message to be sent across network

4) Transport

TCP or UDP
src and dst port



Layer 4 uses ports to identify applications
- **A port is just an integer 0 – 65,535**

Server side:
- **Ports identify an application**
- **Different apps listen on different ports (e.g., web on port 80, telnet on port 23)**

Client side:
- **Ports identify an instance of an application (might have multiple copies of an app running)**
- **Port numbers are randomly chosen by OS**

3) Network (IP)

2) Link (MAC)

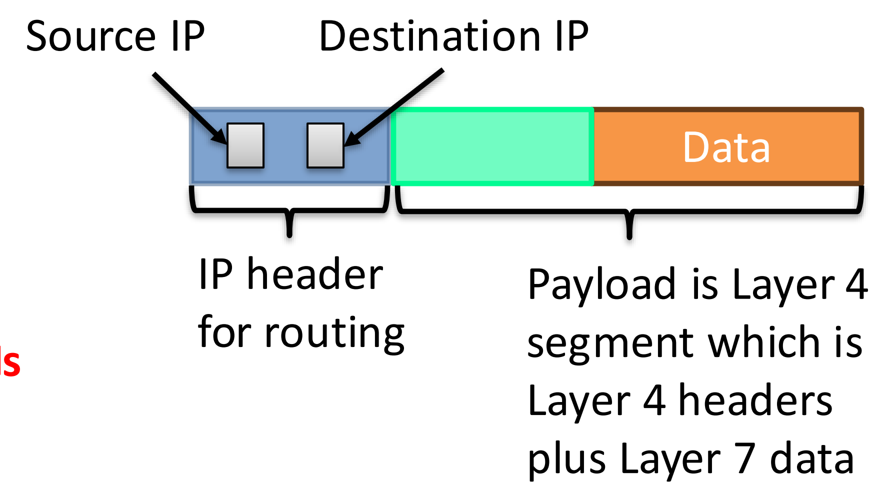
1) Physical

Review: network layers

Network model

Critical information

Network model	Critical information
7) Application	Application message to be sent across network
4) Transport	TCP or UDP src and dst port
3) Network (IP)	src and dst IP address **Routing!**
2) Link (MAC)	**Packets routed across LANs using IP addresses**
1) Physical	



Review: network layers

Network model

Critical information

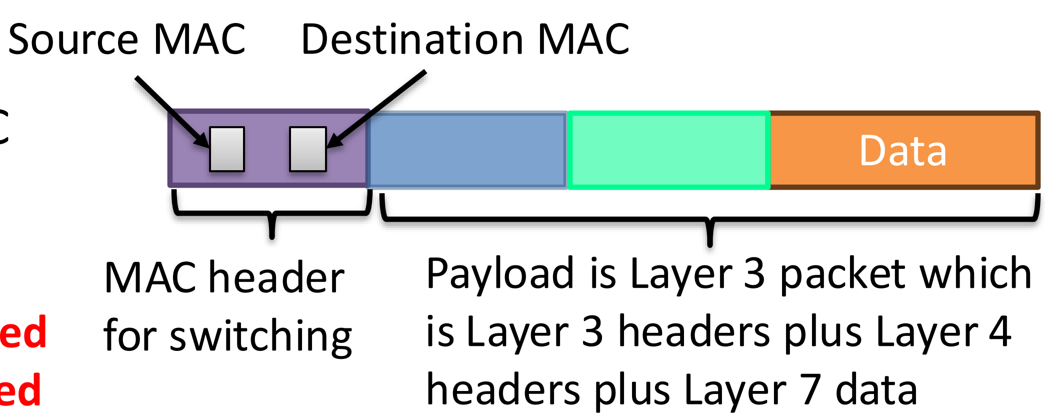
7) Application — Application message to be sent across network

4) Transport — TCP or UDP
src and dst port

3) Network (IP) — src and dst
IP address
Routing!

2) Link (MAC) — src and dst MAC
address
Switching!

1) Physical — **Frames transmitted *within* a LAN based on MAC address**

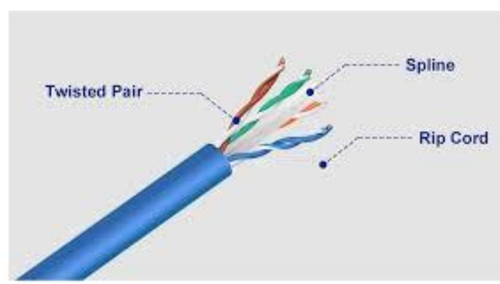


Review: network layers

Network model

Critical information

Network model	Critical information
7) Application	Application message to be sent across network
4) Transport	TCP or UDP src and dst port
3) Network (IP)	src and dst IP address Routing!
2) Link (MAC)	src and dst MAC address Switching!
1) Physical	**Sends frames to another host over cable or RF**

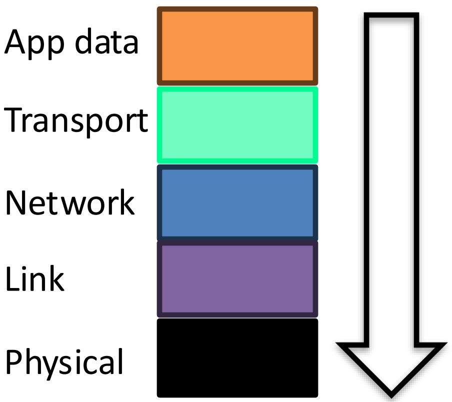


Sender’s data goes from Layer 7 to Layer 1

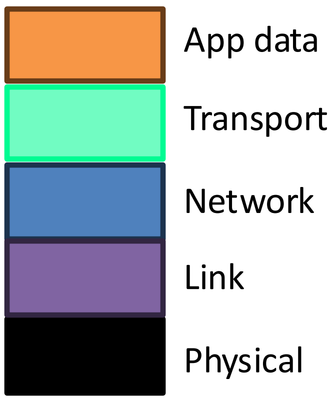
Client

Sending from Client to Server

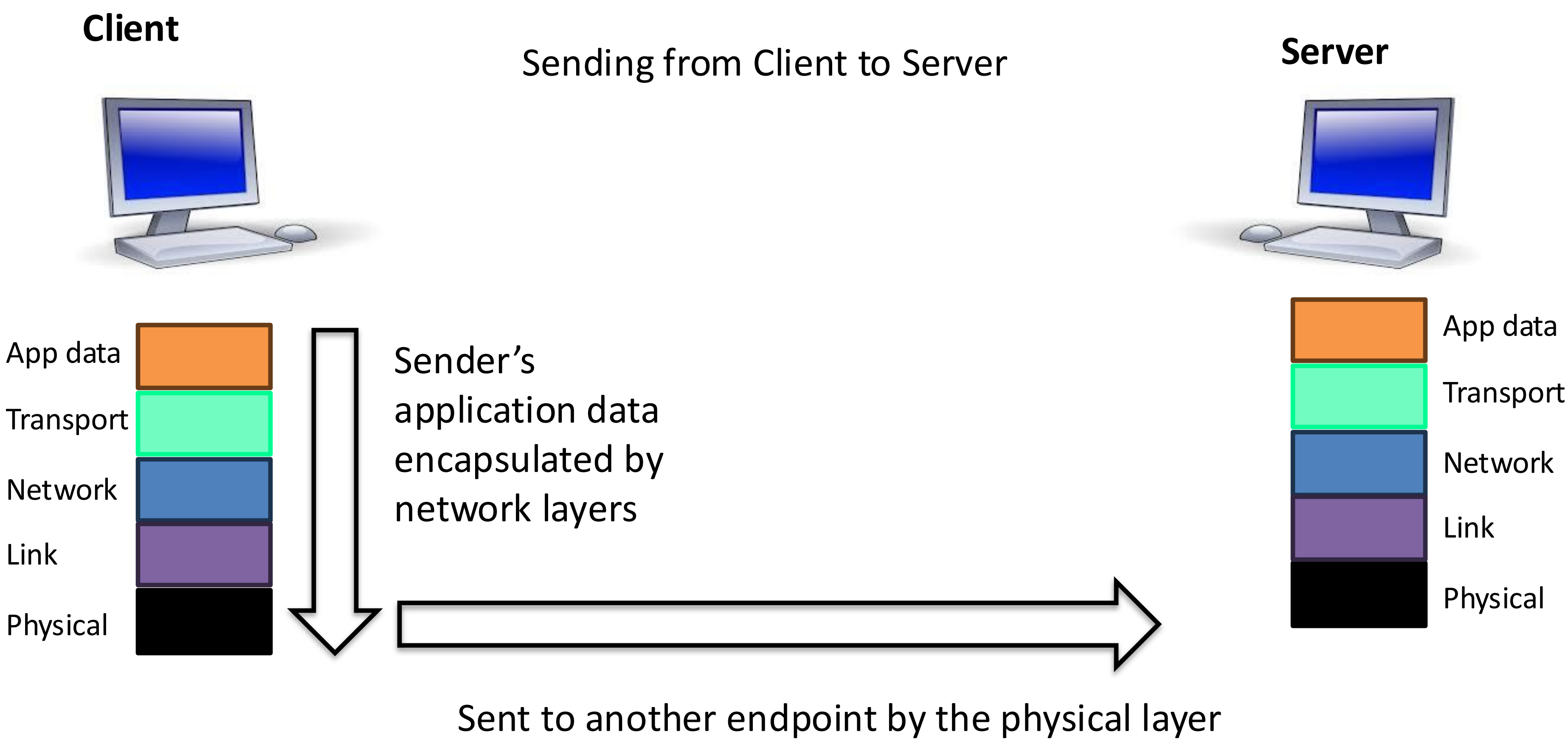
Server



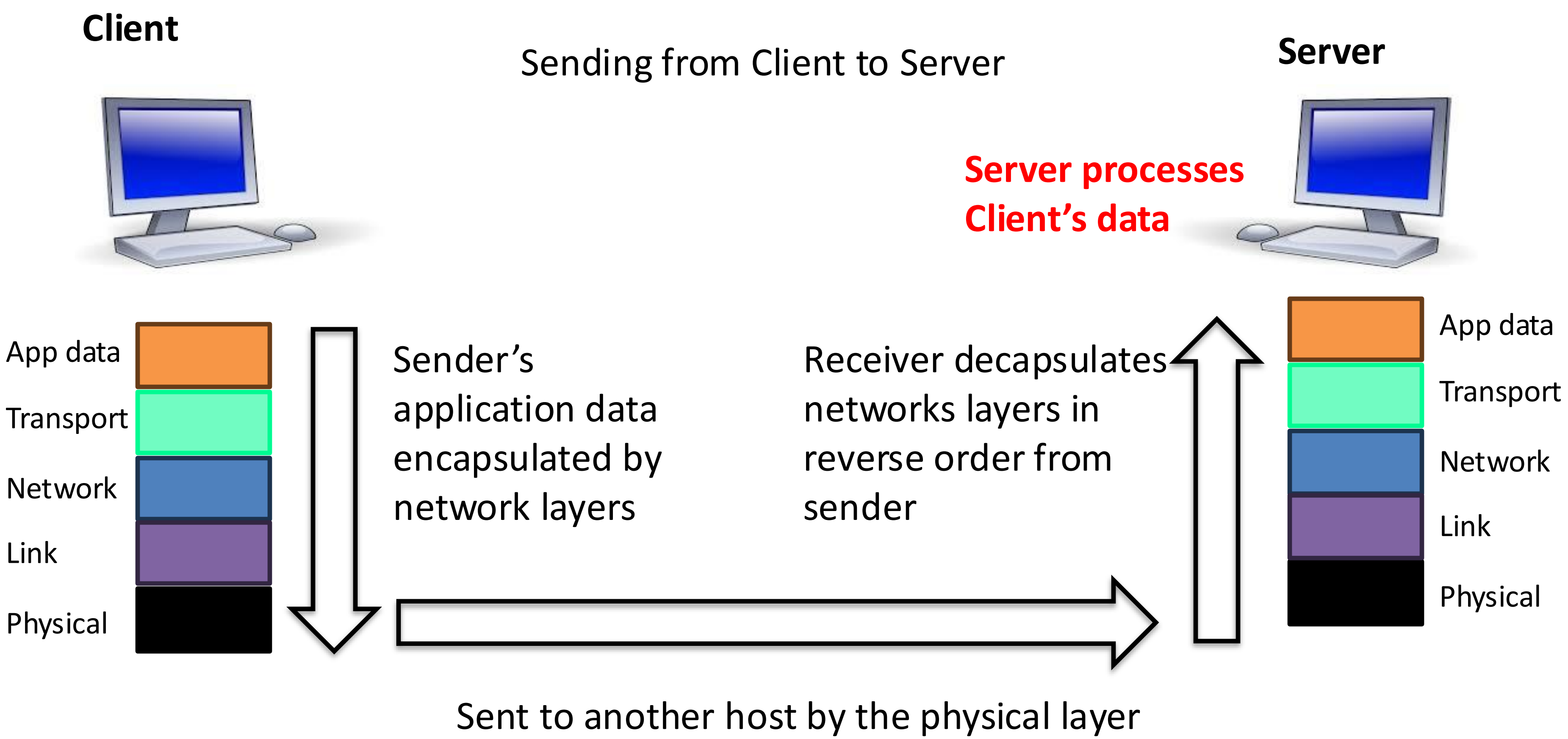
Sender’s application data encapsulated by network layers



Data sent across the network on Layer 1

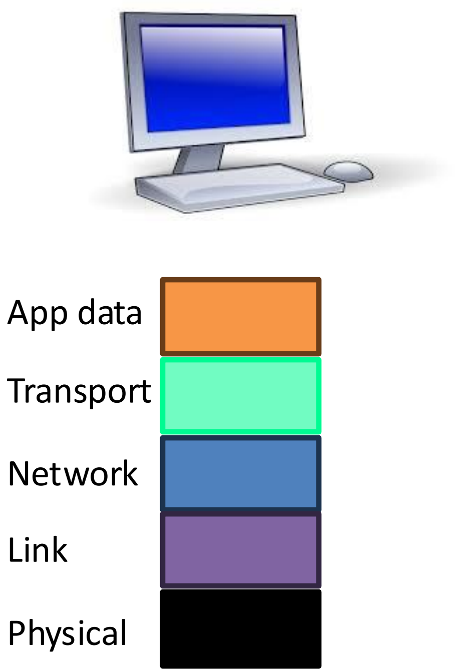


Receiver strips off headers from Layer 2 up to Layer 7

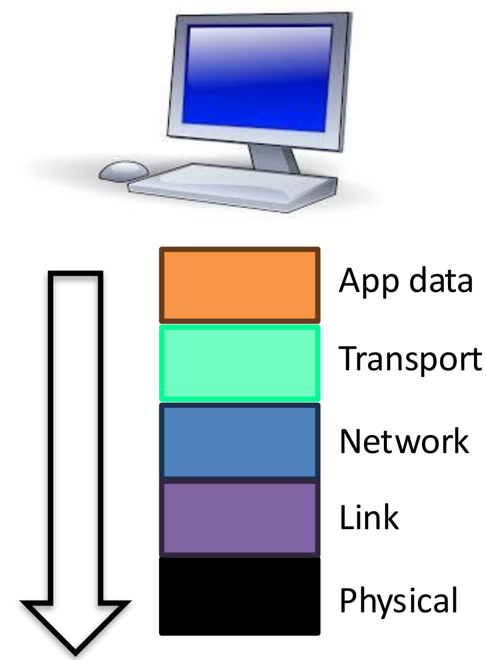


Reply from Server reverses the process back to the Client

Client

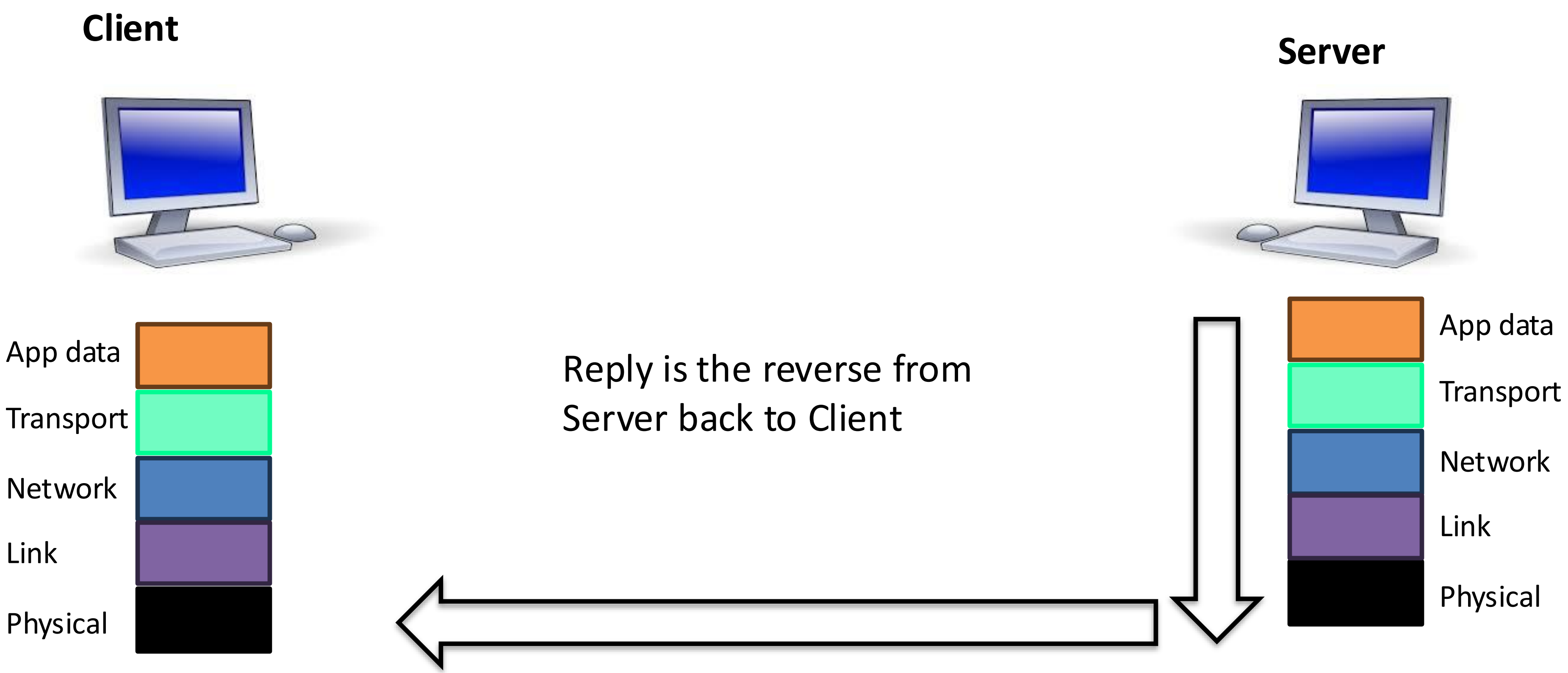


Server

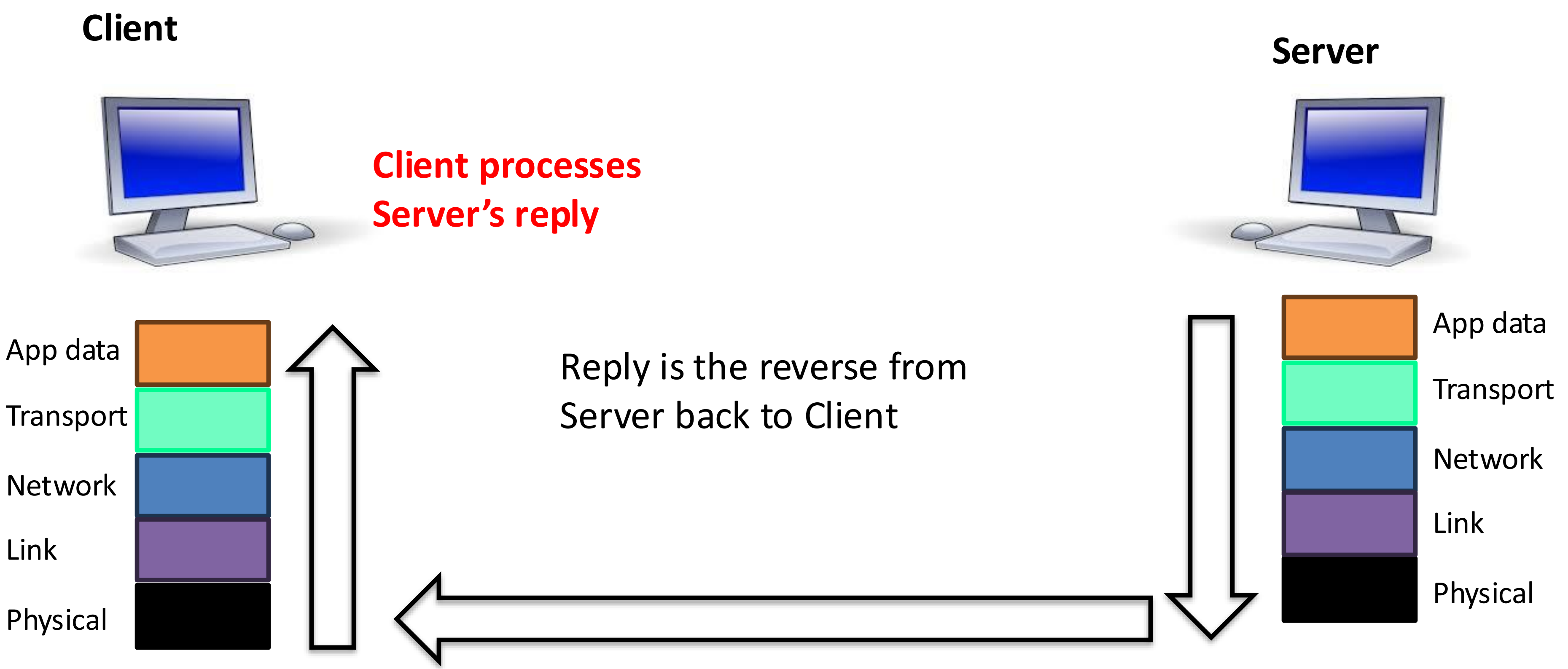


Reply is the reverse from Server back to Client

Reply from Server reverses the process back to the Client

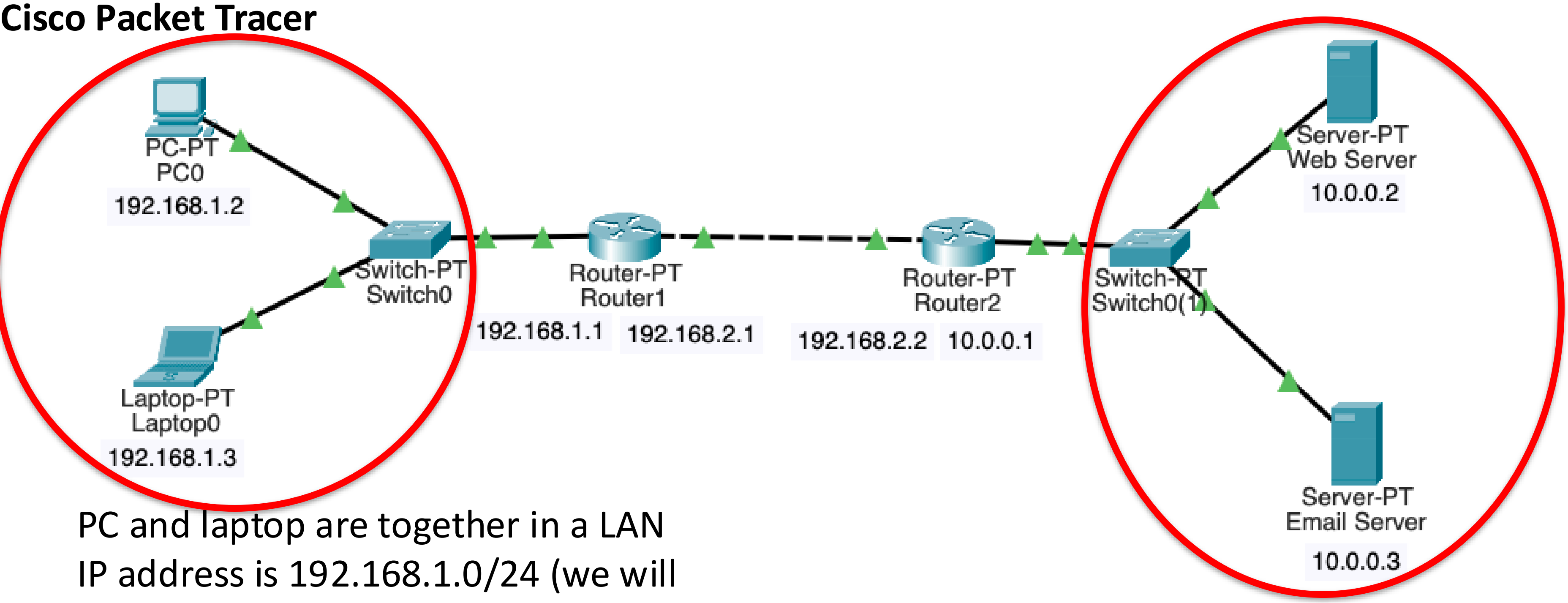


Reply from Server reverses the process back to the Client



Example networks, PCs in one network, servers in another network

Cisco Packet Tracer

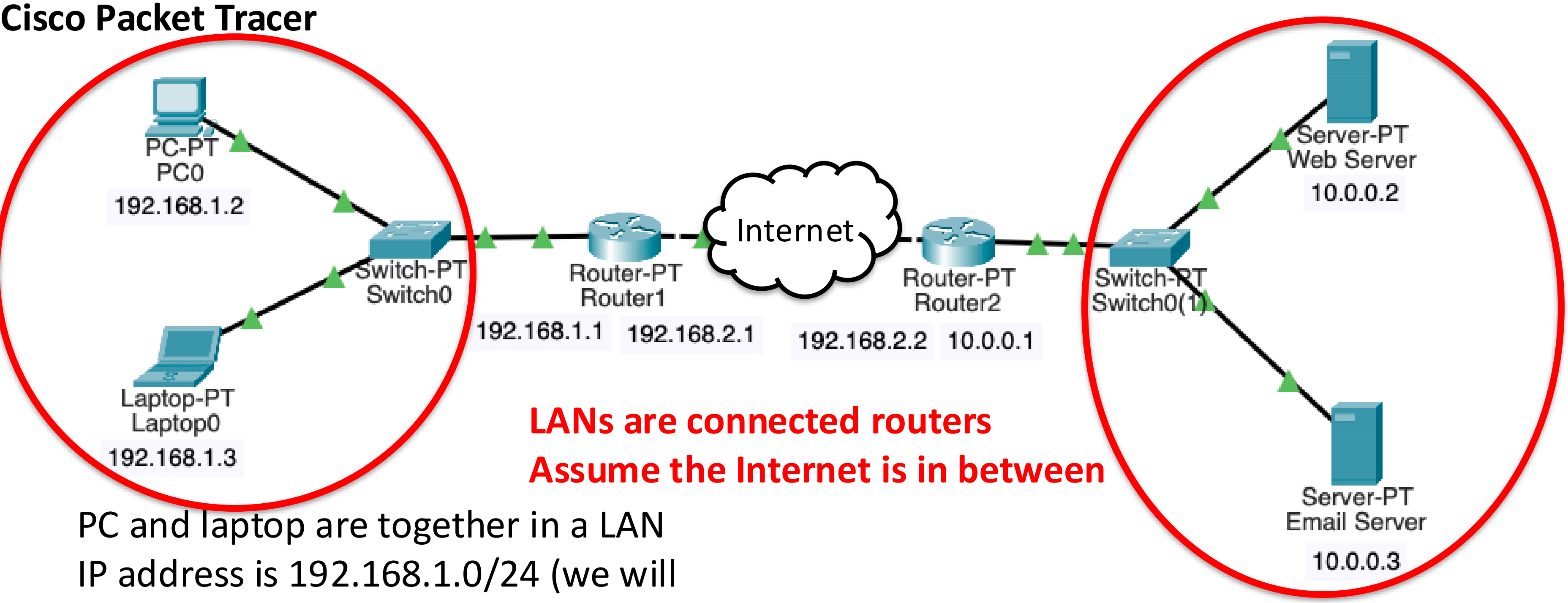


PC and laptop are together in a LAN
IP address is 192.168.1.0/24 (we will cover this notation soon)
This LAN is connected by a Switch

Web server and Email server are together in another LAN
IP address is 10.0.0.0/24
This LAN is connected by a different Switch

Example networks, PCs in one network, servers in another network

Cisco Packet Tracer



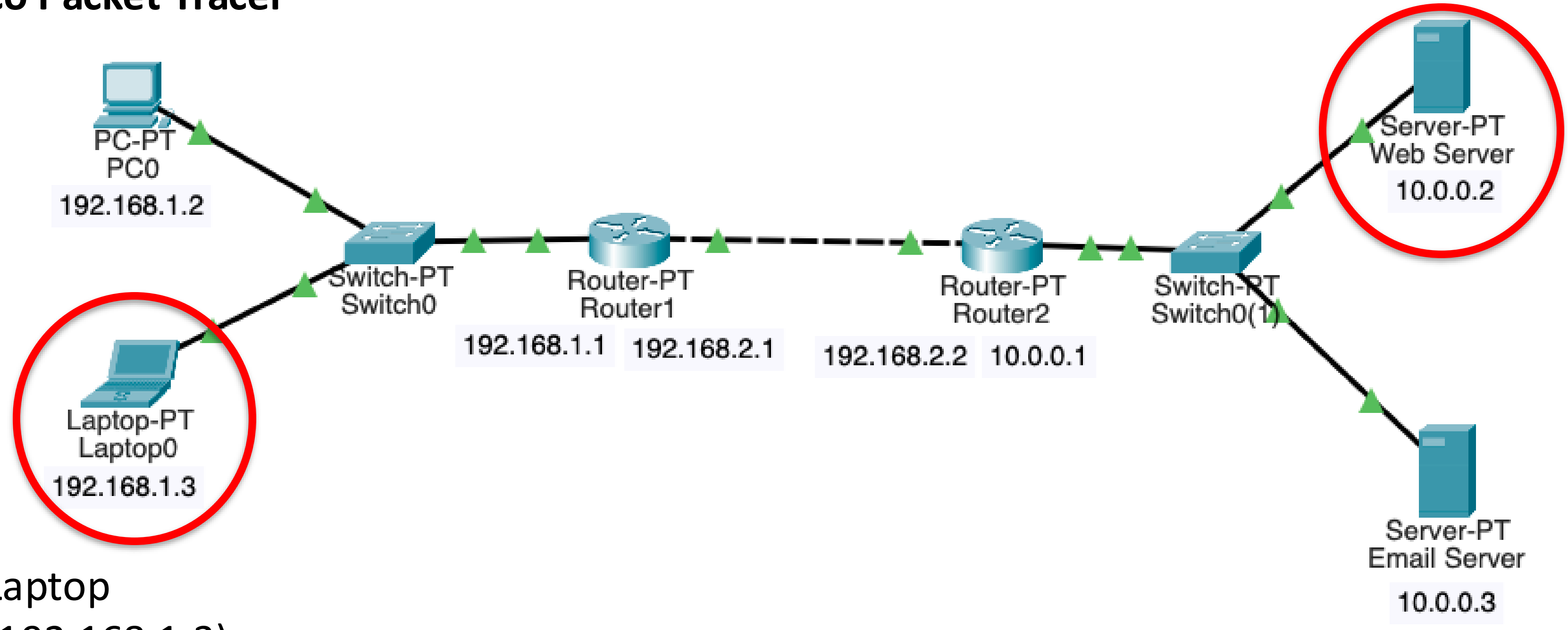
LANs are connected routers
Assume the Internet is in between

PC and laptop are together in a LAN
IP address is 192.168.1.0/24 (we will cover this notation soon)
This LAN is connected by a Switch

Web server and Email server are together in another LAN
IP address is 10.0.0.0/24
This LAN is connected by a different Switch

Internet largely uses a client/server model where clients make requests to servers

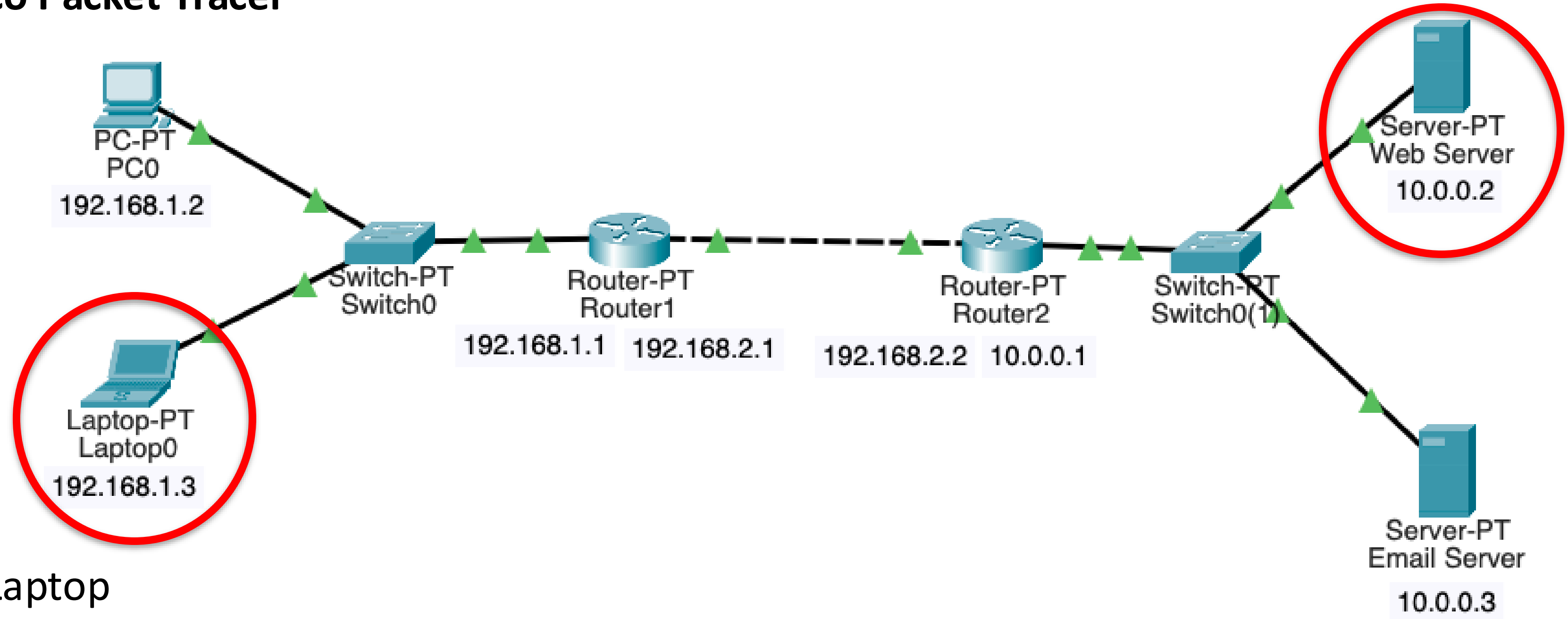
Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer

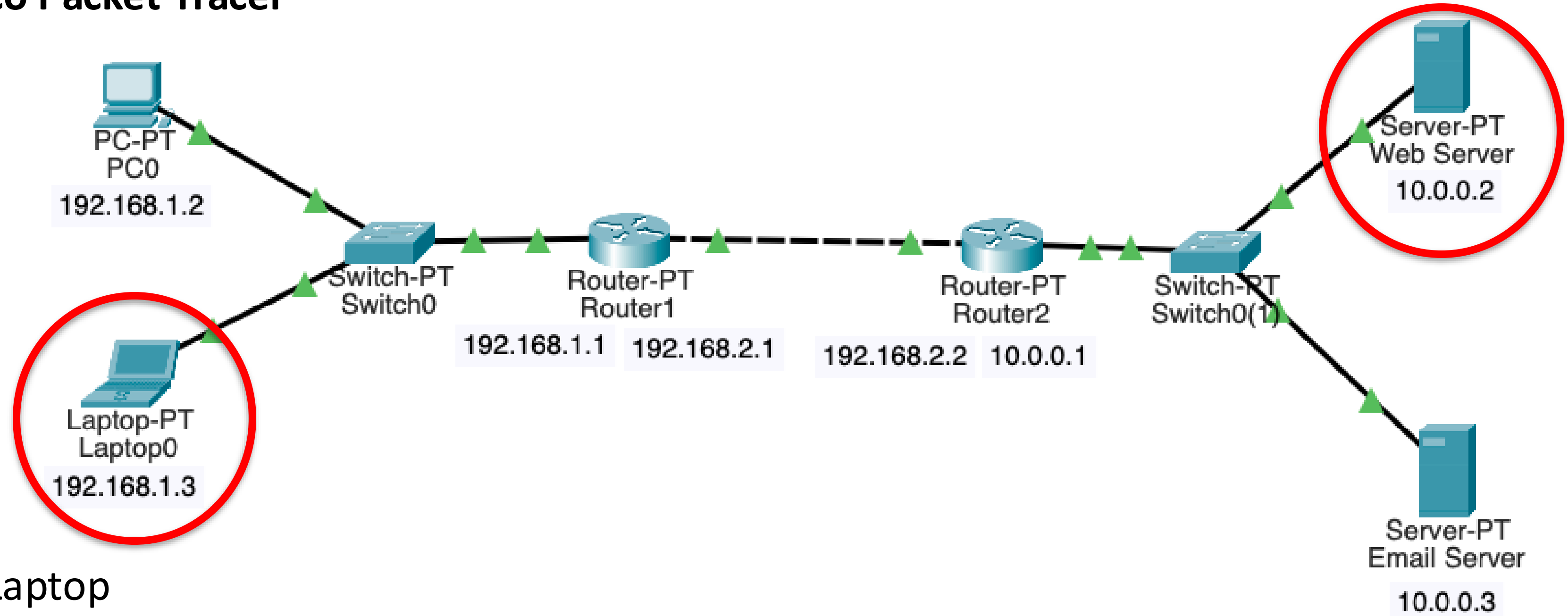


Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

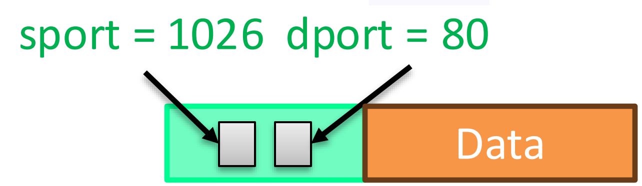
Layer 7 (App): Laptop formats message to Web Server identifying the web page the Laptop wants to see

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

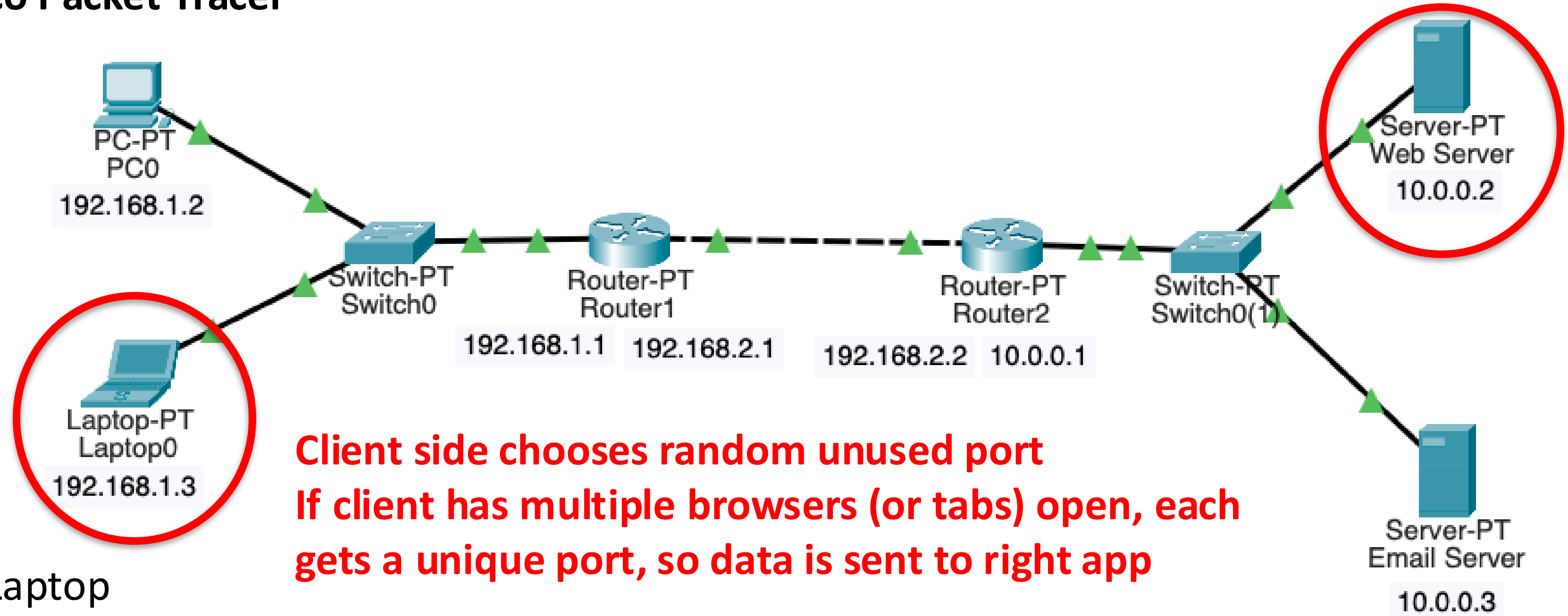


Layer 4 (Transport): Laptop sets

- **dst port = 80 (well-known HTTP)**
- **src port = 1026 (random unused)**

Internet largely uses a client/server model where clients make requests to servers

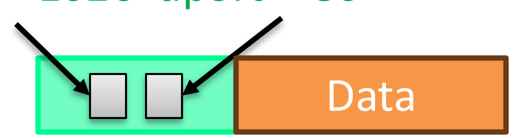
Cisco Packet Tracer



Client side chooses random unused port
If client has multiple browsers (or tabs) open, each gets a unique port, so data is sent to right app

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

sport = 1026 dport = 80

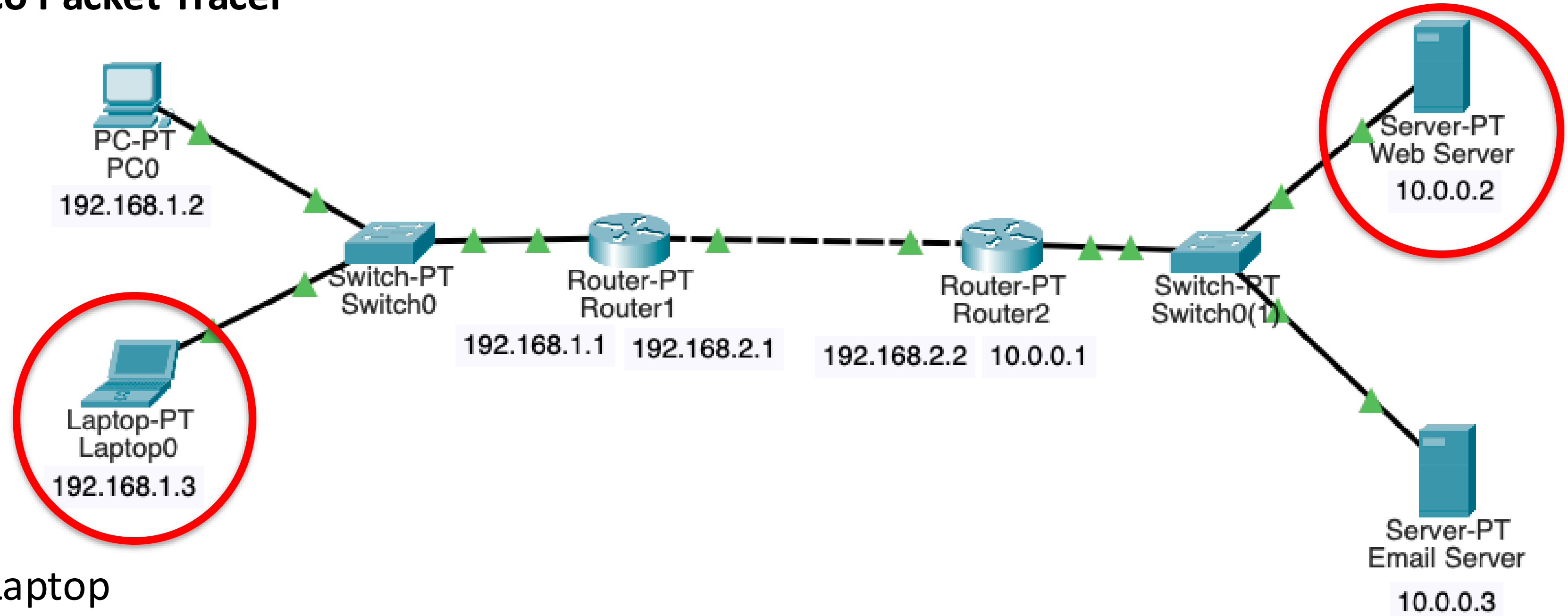


Layer 4 (Transport): Laptop sets

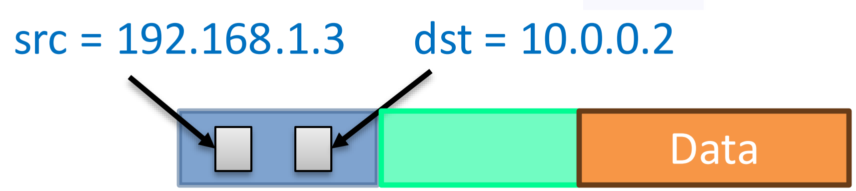
- **dst port = 80 (well-known HTTP)**
- **src port = 1026 (random unused)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

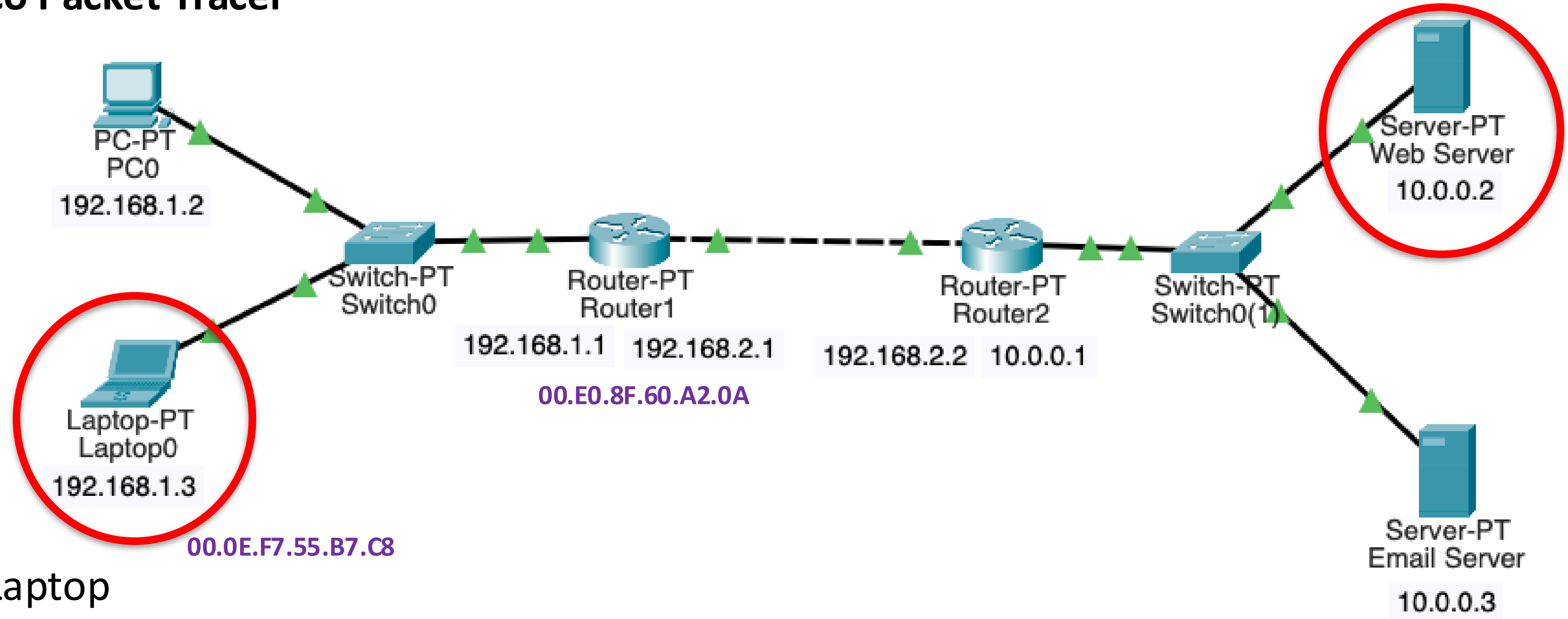


Layer 3 (Network): Laptop sets IP addresses

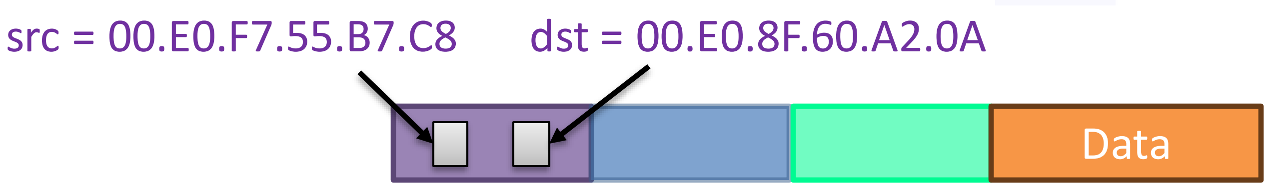
- **dst= 10.0.0.2 (Web server)**
- **src = 192.168.1.3 (Laptop)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

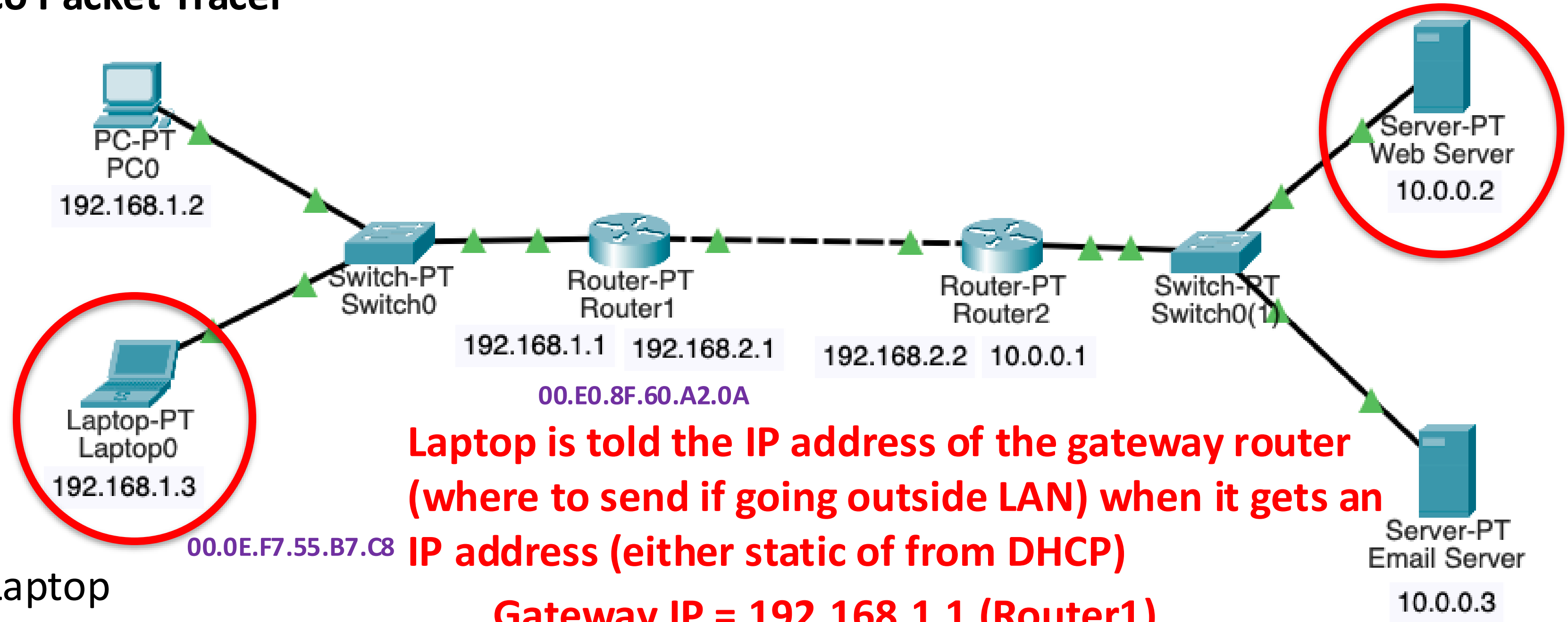


Layer 2: Laptop sets MAC addresses

- **dst= 00.E0.8F.60.A2.0A (Router 1)**
- **src = 00.E0.F7.55.B7.C8 (Laptop)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



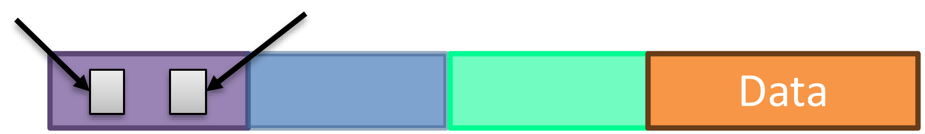
Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Laptop is told the IP address of the gateway router (where to send if going outside LAN) when it gets an IP address (either static of from DHCP)

Gateway IP = 192.168.1.1 (Router1)

Laptop gets gateway router's MAC from ARP

src = 00.E0.F7.55.B7.C8 dst = 00.E0.8F.60.A2.0A

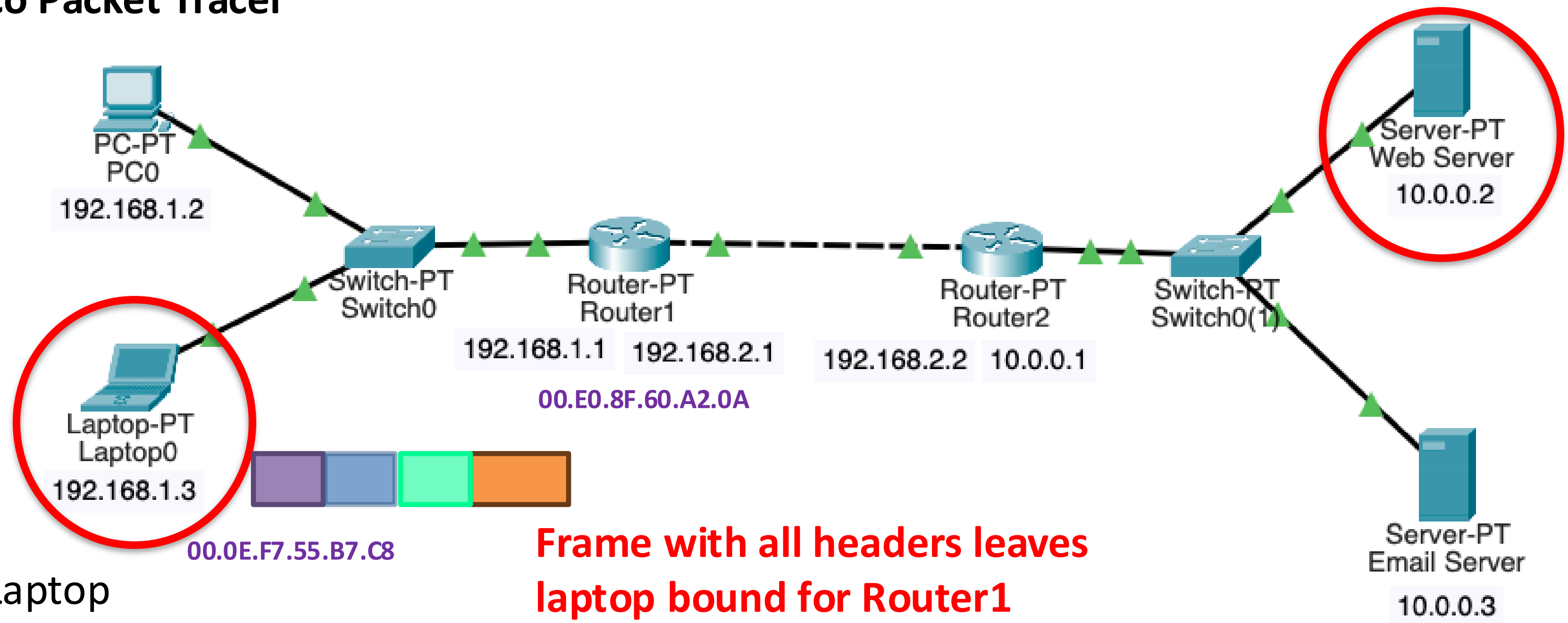


Layer 2: Laptop sets MAC addresses

- **dst= 00.E0.8F.60.A2.0A (Router 1)**
- **src = 00.E0.F7.55.B7.C8 (Laptop)**

Internet largely uses a client/server model where clients make requests to servers

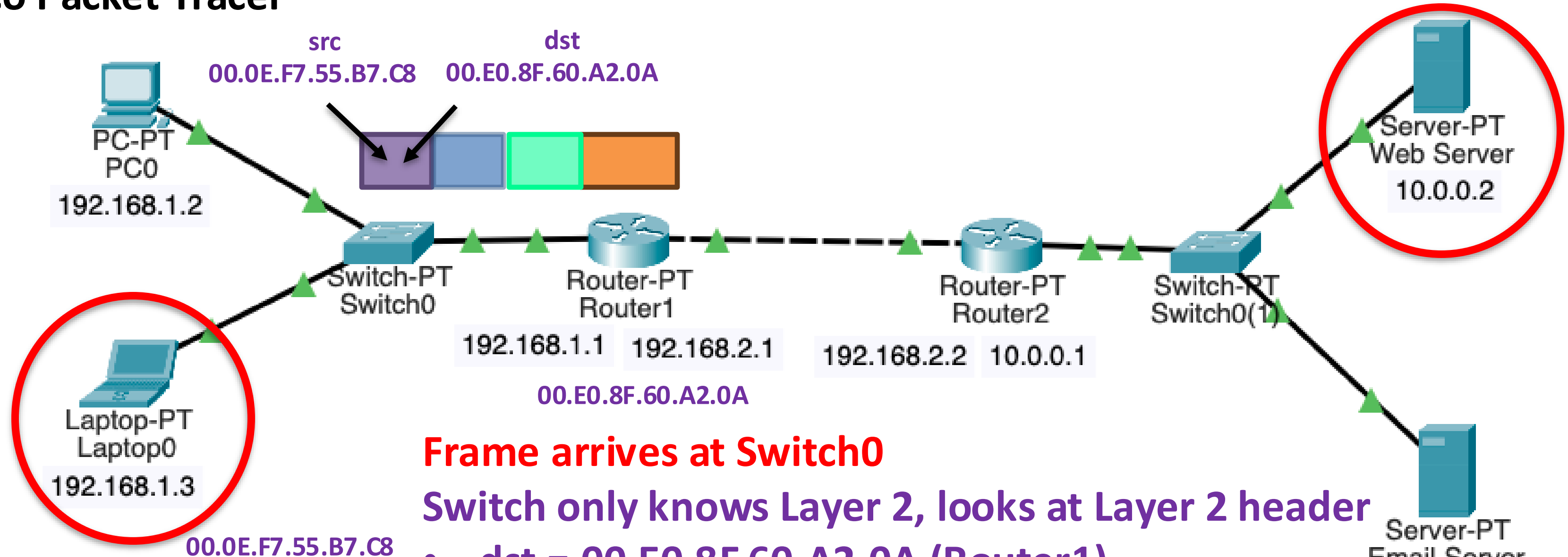
Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Frame arrives at Switch0

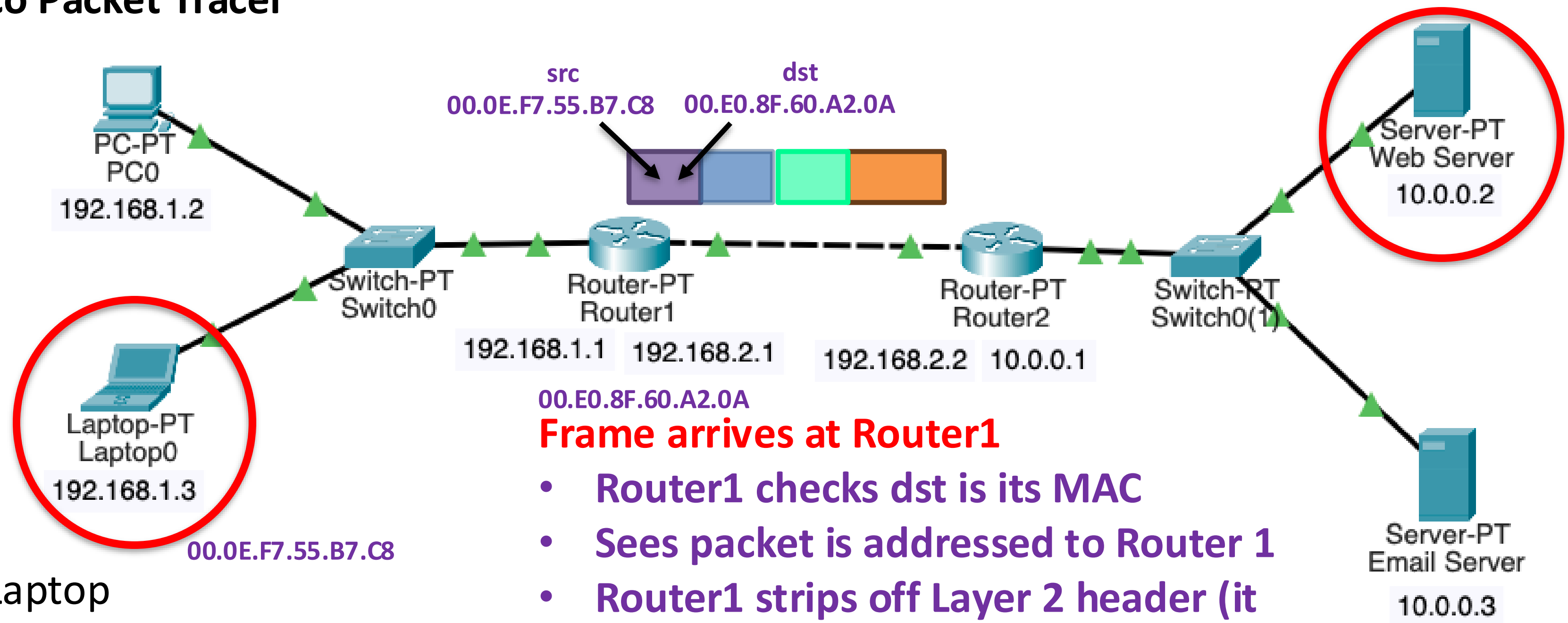
Switch only knows Layer 2, looks at Layer 2 header

- **dst = 00.E0.8F.60.A2.0A (Router1)**
- **src = 00.0E.F7.55.B7.C8 (Laptop)**

Switch knows which physical port dst is plugged into, forwards packet to Router1

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



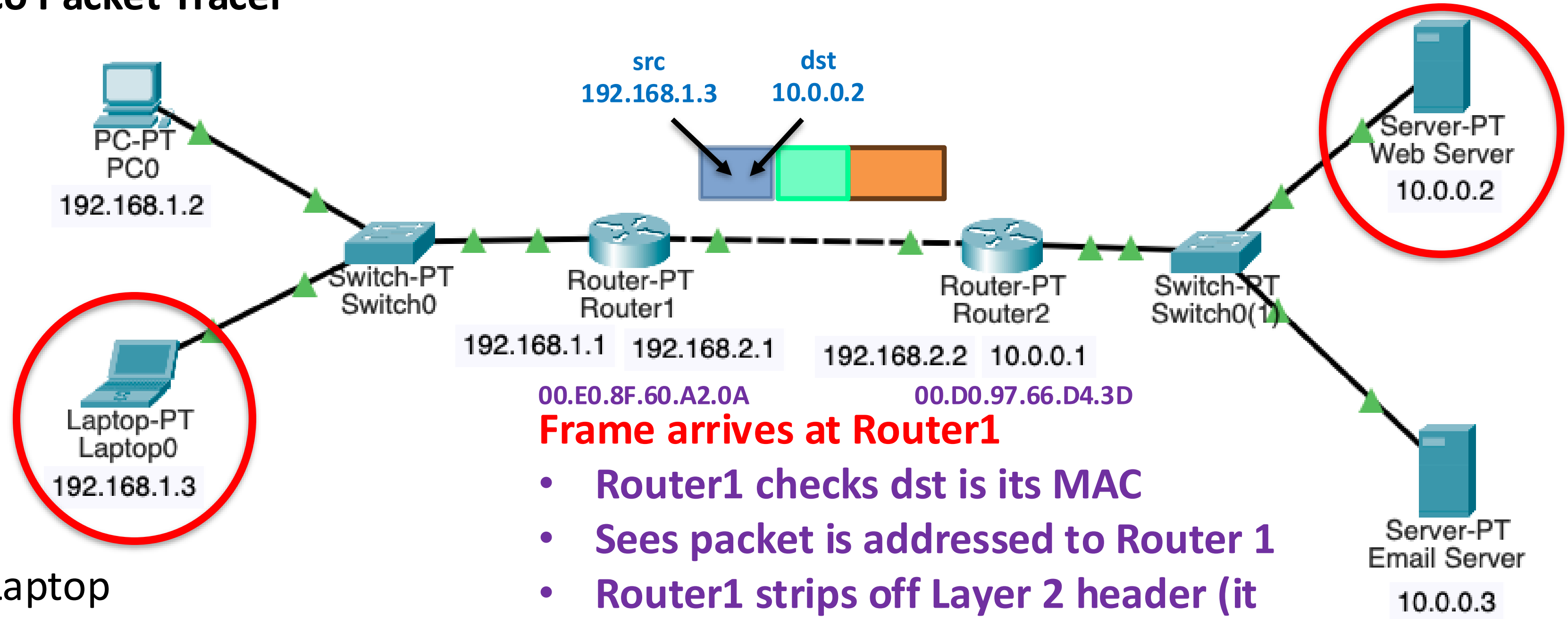
Frame arrives at Router1

- **Router1 checks dst is its MAC**
- **Sees packet is addressed to Router 1**
- **Router1 strips off Layer 2 header (it doesn't know Layer 2, only Layer 3)**

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Frame arrives at Router1

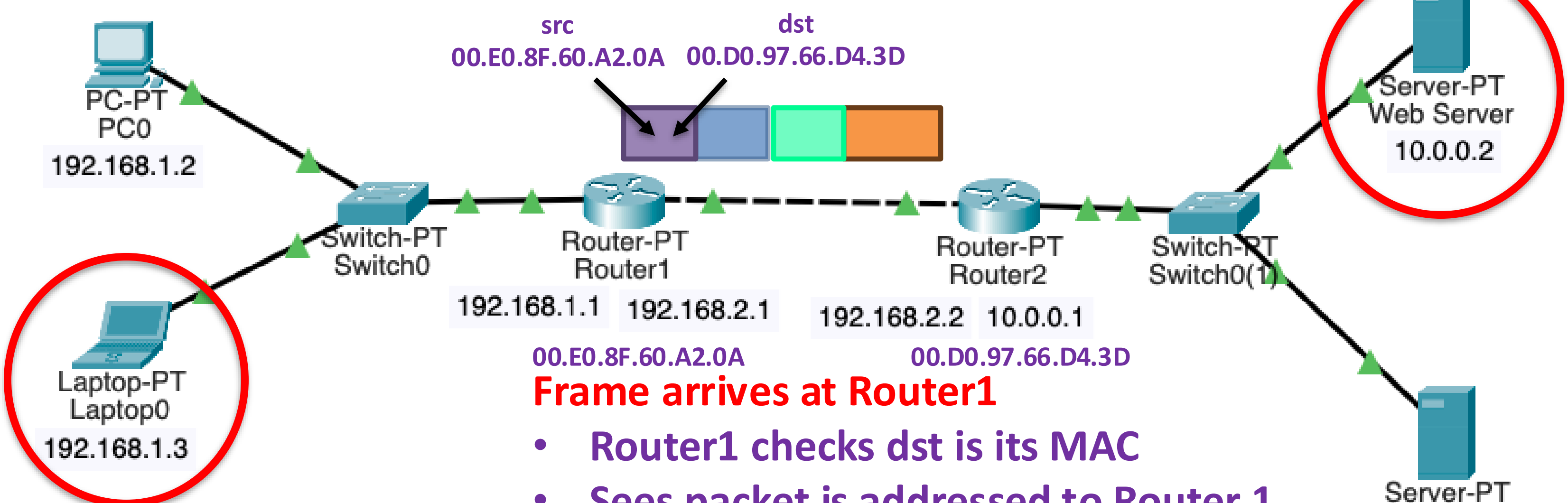
- **Router1 checks dst is its MAC**
- **Sees packet is addressed to Router 1**
- **Router1 strips off Layer 2 header (it doesn't know Layer 2, only Layer 3)**

Router 1 looks at Layer 3 header

- **Realizes next stop is Router2 if it wants to send the packet to 10.0.0.2 (we will see how soon)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Frame arrives at Router1

- **Router1 checks dst is its MAC**
- **Sees packet is addressed to Router 1**
- **Router1 strips off Layer 2 header (it doesn't know Layer 2, only Layer 3)**

Router 1 looks at Layer 3 header

- **Realizes next stop is Router2 if it wants to send the packet to 10.0.0.2 (we will see how soon)**

Router1 "re-writes" Layer 2 header for Router2

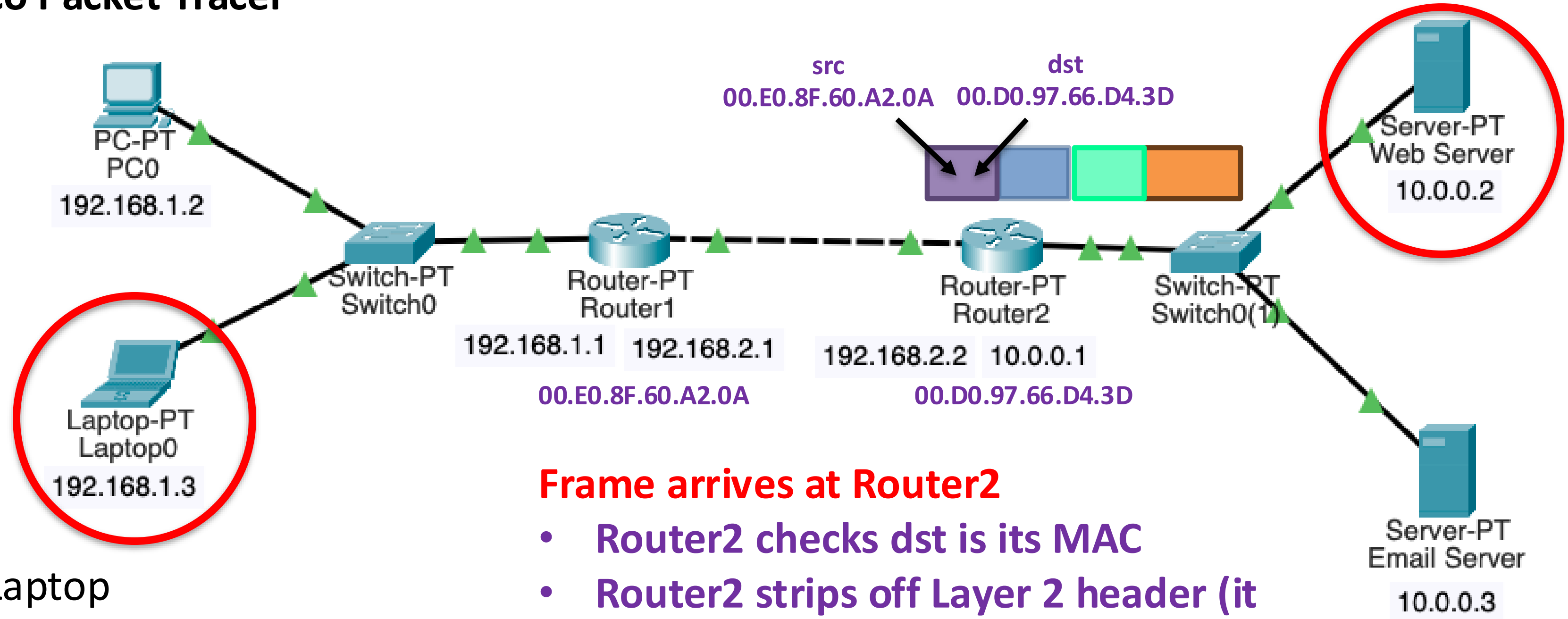
- **dst = 00.D0.97.66.D4.3D (Router2)**
- **src = 00.E0.8F.60.A2.0A (Router1)**

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

How did Router1 get the MAC of Router2? ARP!

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



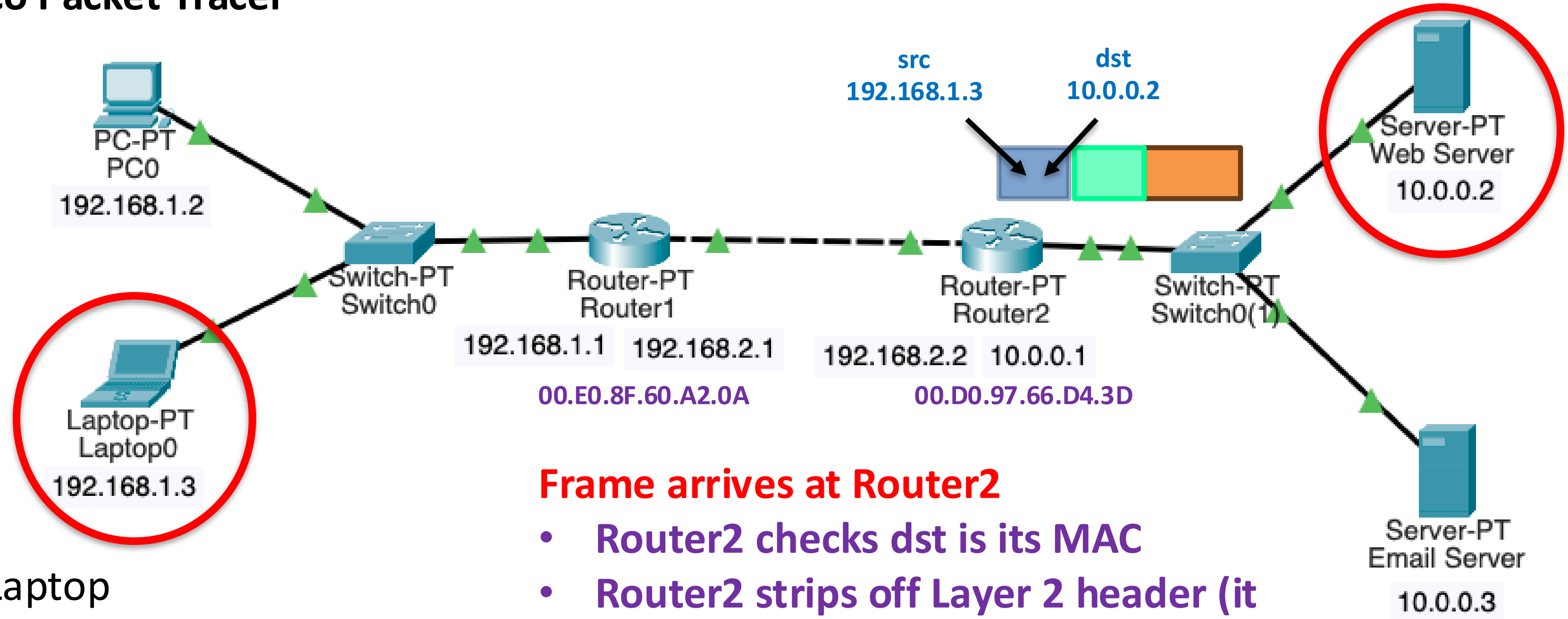
Frame arrives at Router2

- **Router2 checks dst is its MAC**
- **Router2 strips off Layer 2 header (it doesn't know Layer 2, only Layer 3)**

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Frame arrives at Router2

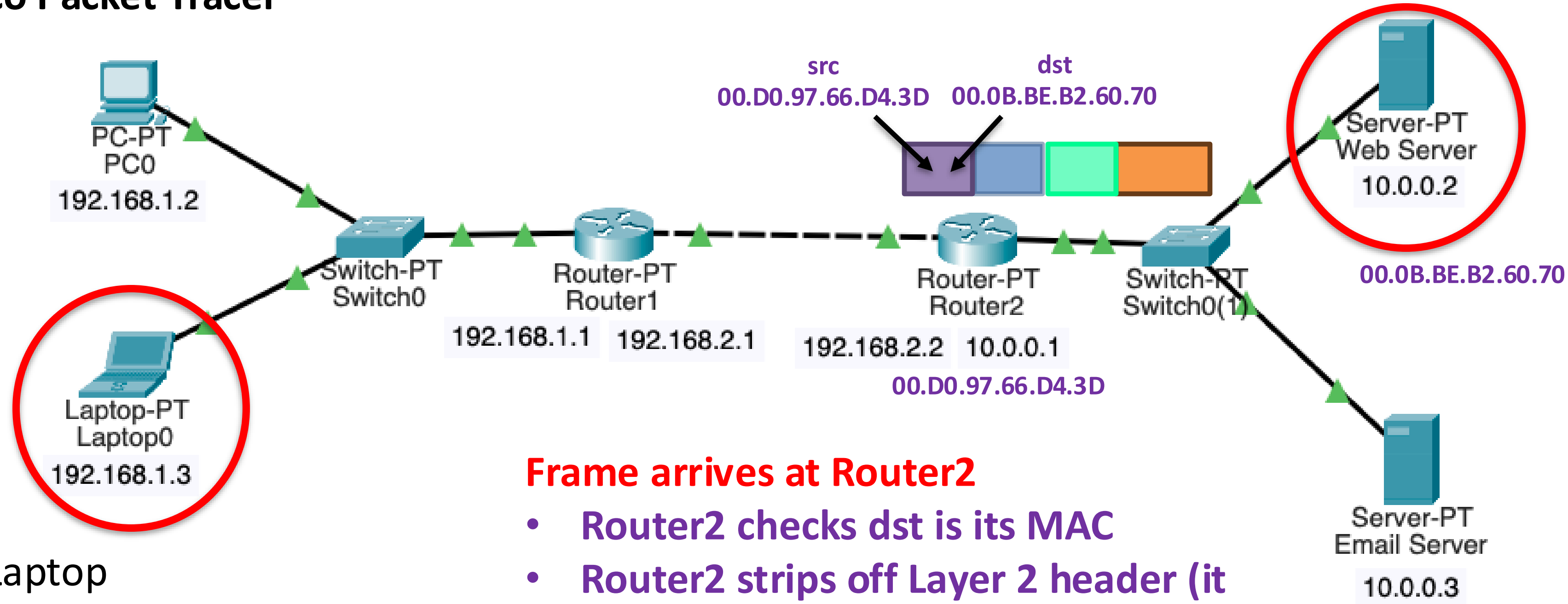
- **Router2 checks dst is its MAC**
- **Router2 strips off Layer 2 header (it doesn't know Layer 2, only Layer 3)**

Router 2 looks at Layer 3 header

- **dst = 10.0.0.2 (Web Server)**
- **src = 192.168.1.3 (Laptop)**
- **Realizes next stop is Web Server (we will see how soon)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Frame arrives at Router2

- **Router2 checks dst is its MAC**
- **Router2 strips off Layer 2 header (it doesn't know Layer 2, only Layer 3)**

Router 2 looks at Layer 3 header

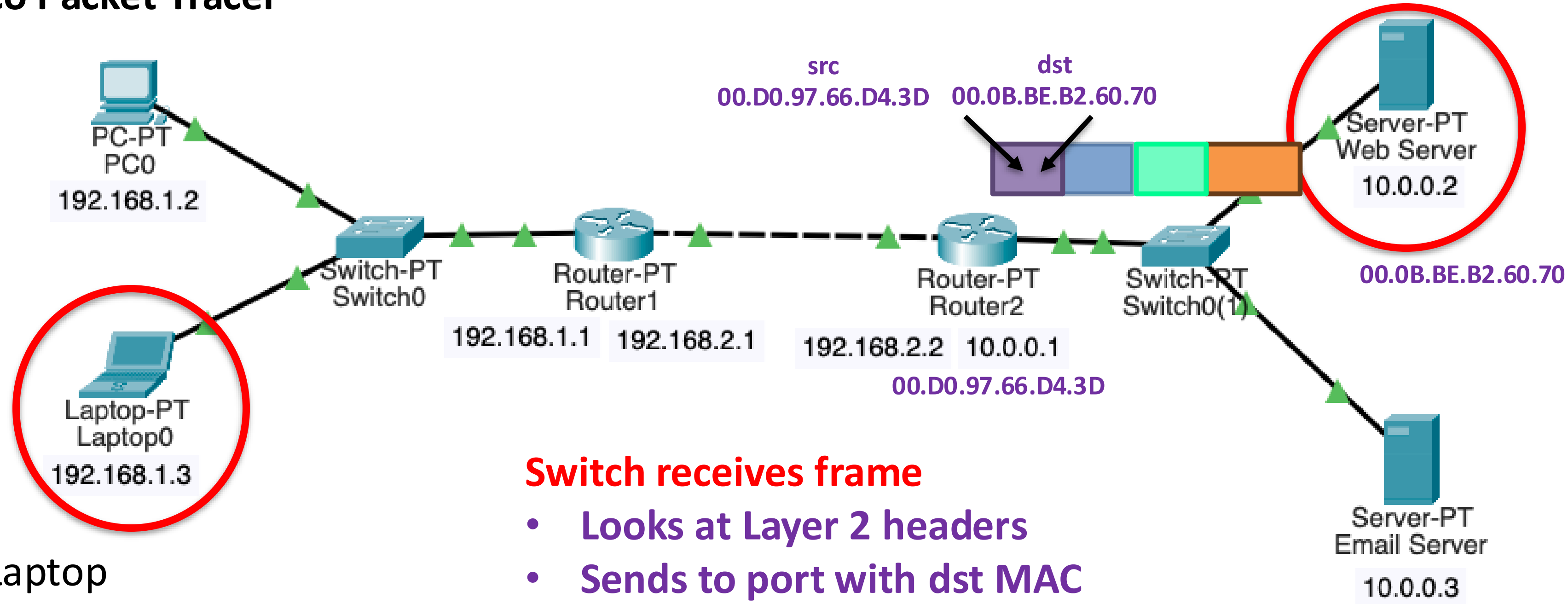
- **dst = 10.0.0.2 (Web Server)**
- **src = 192.168.1.3 (Laptop)**
- **Realizes next stop is Router2 (we will see how soon)**

Router2 "re-writes" Layer 2 header for Web Server

- **dst = 00.0B.BE.B2.60.70 (Web server)**
- **src = 00.D0.97.66.D4.3D (Router 2)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



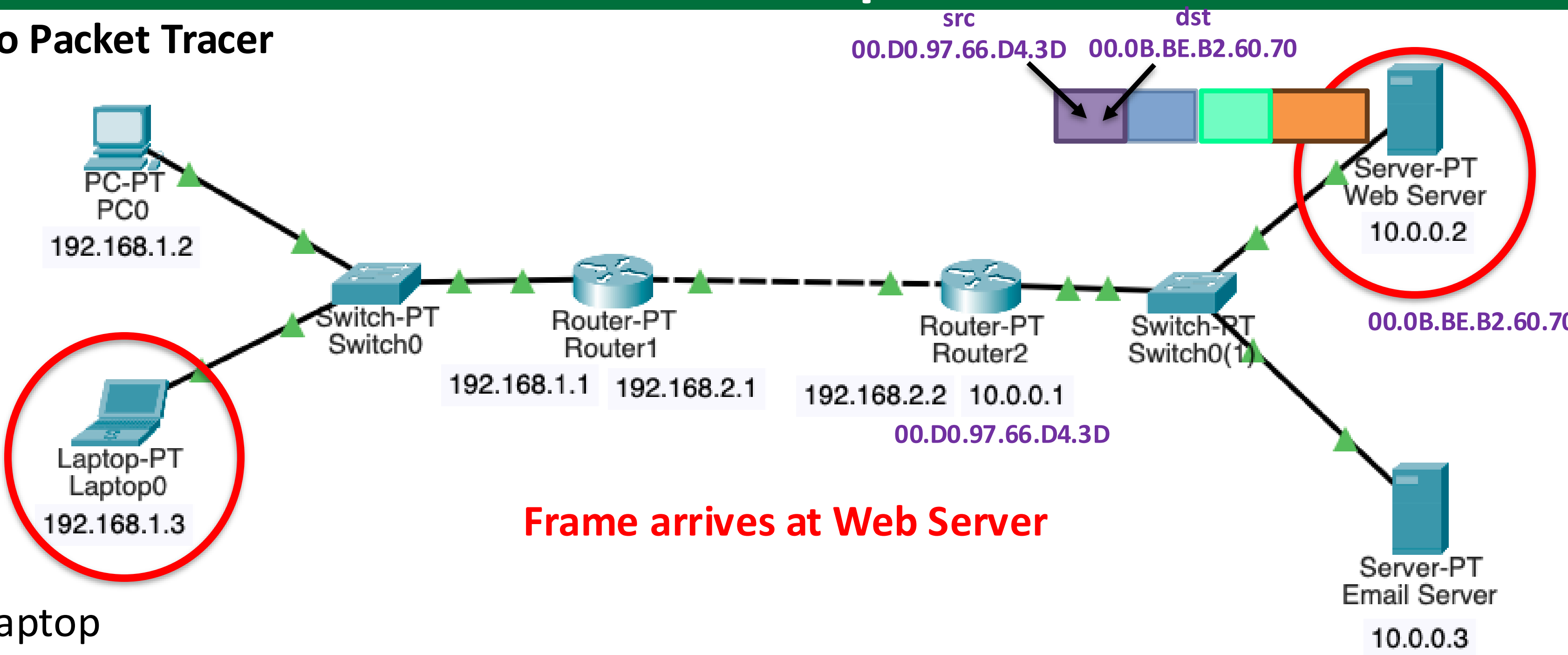
Switch receives frame

- **Looks at Layer 2 headers**
- **Sends to port with dst MAC (Web Server)**

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

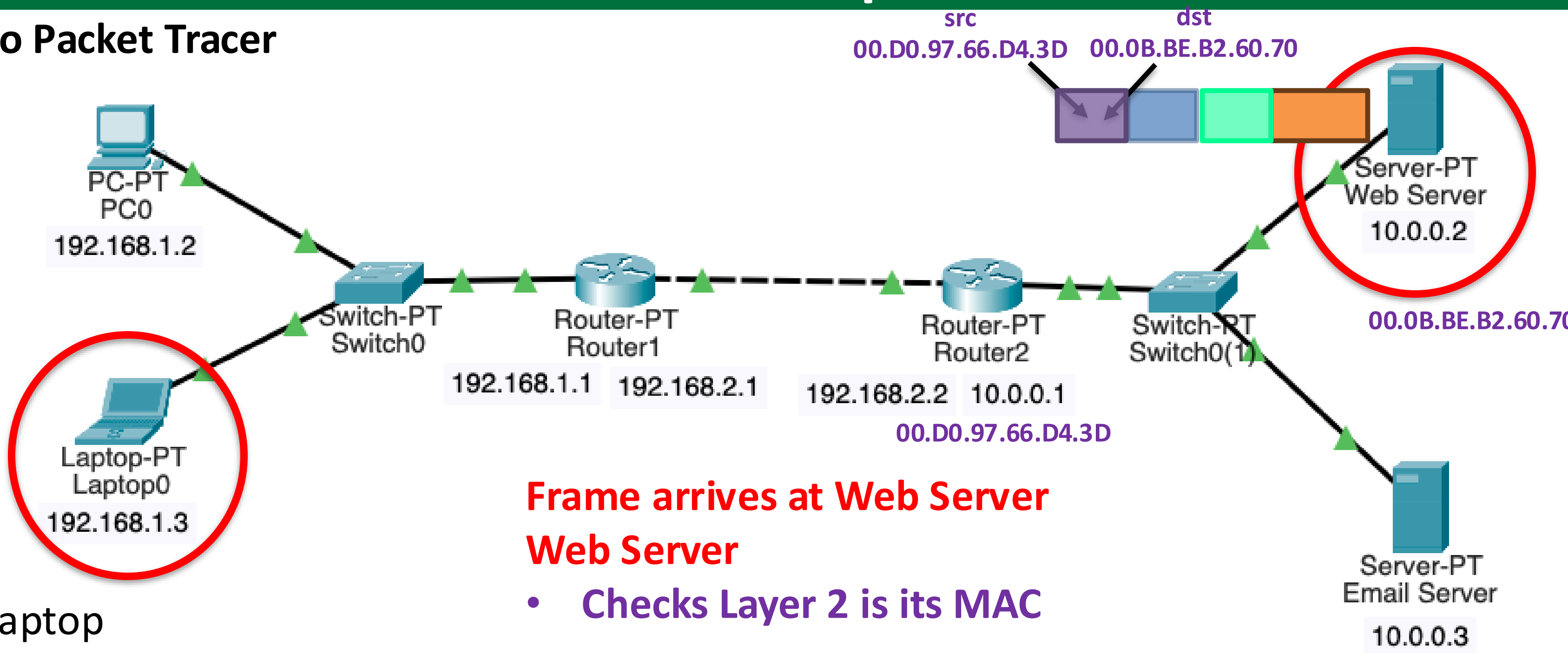
Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

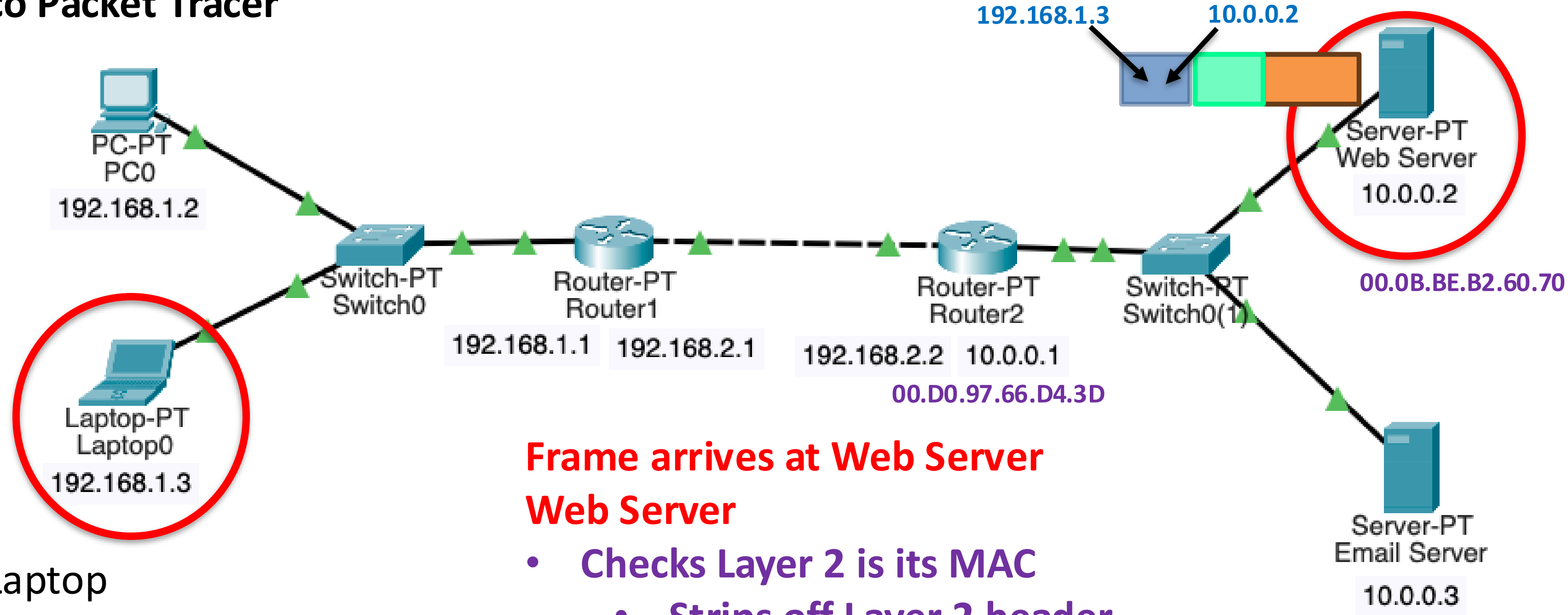
Frame arrives at Web Server

Web Server

- **Checks Layer 2 is its MAC**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Frame arrives at Web Server

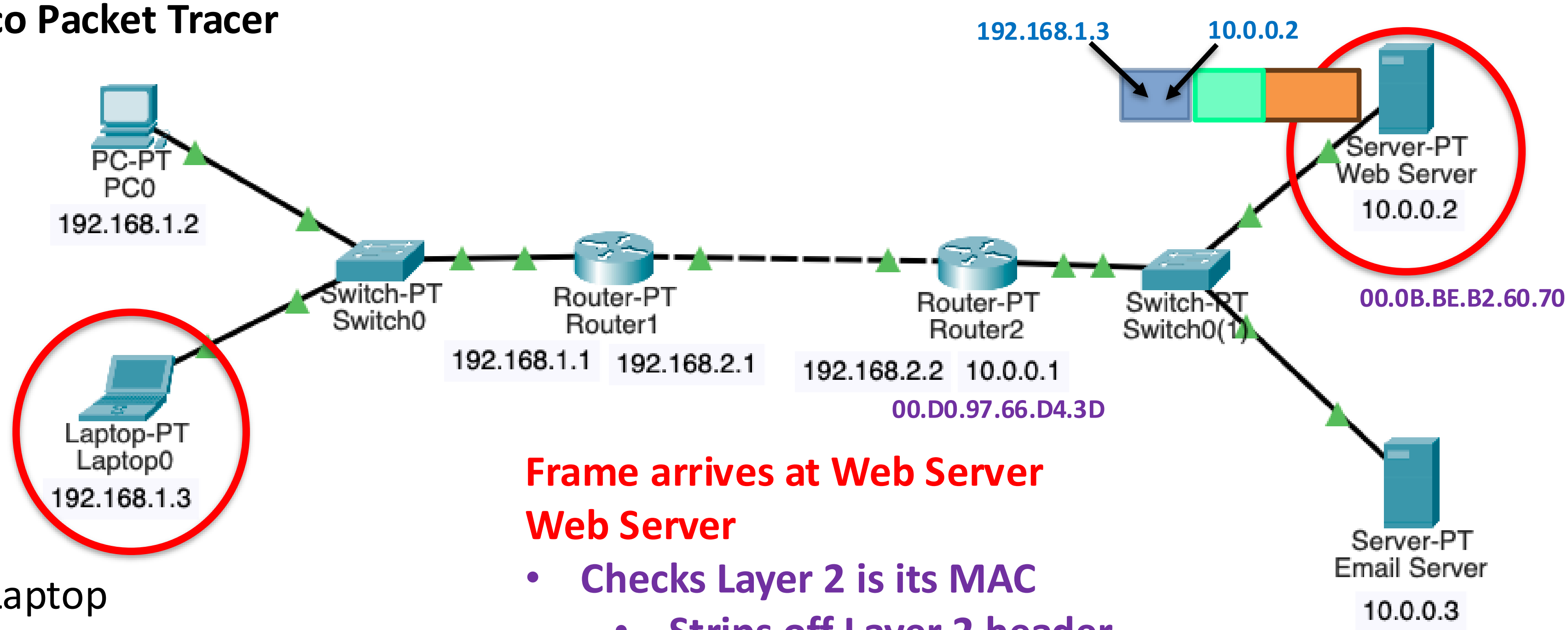
Web Server

- **Checks Layer 2 is its MAC**
    - **Strips off Layer 2 header**

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



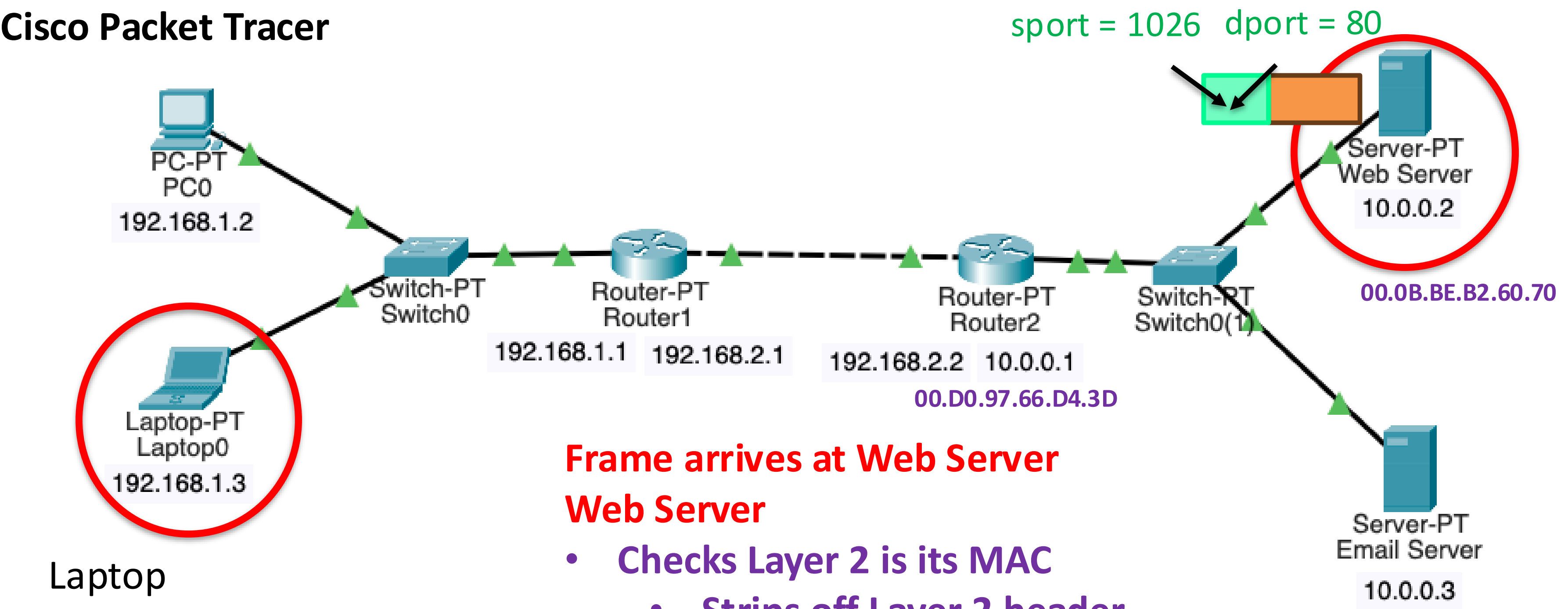
Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Frame arrives at Web Server

Web Server

- **Checks Layer 2 is its MAC**
  - **Strips off Layer 2 header**
- **Checks Layer 3 IP address**
  - **Stores src IP (Laptop at 192.168.1.3)**
  - **Strips off Layer 3 header**

Internet largely uses a client/server model where clients make requests to servers



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

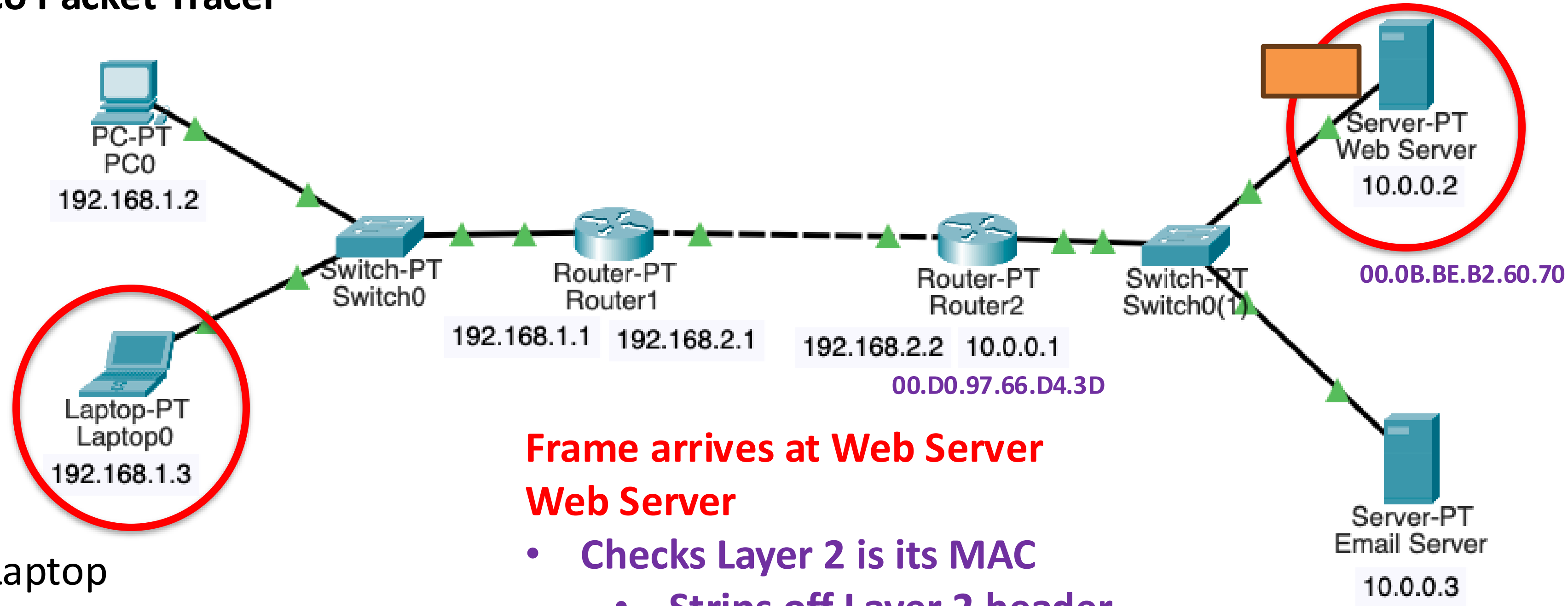
Frame arrives at Web Server

Web Server

- **Checks Layer 2 is its MAC**
  - **Strips off Layer 2 header**
- **Checks Layer 3 IP address**
  - **Stores src IP (Laptop at 192.168.1.3)**
  - **Strips off Layer 3 header**
- **Checks Layer 4**
  - **Notes dst port (80=web)**
  - **Stores sport (1026=Laptop) for return**
  - **Strips off Layer 4 header**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

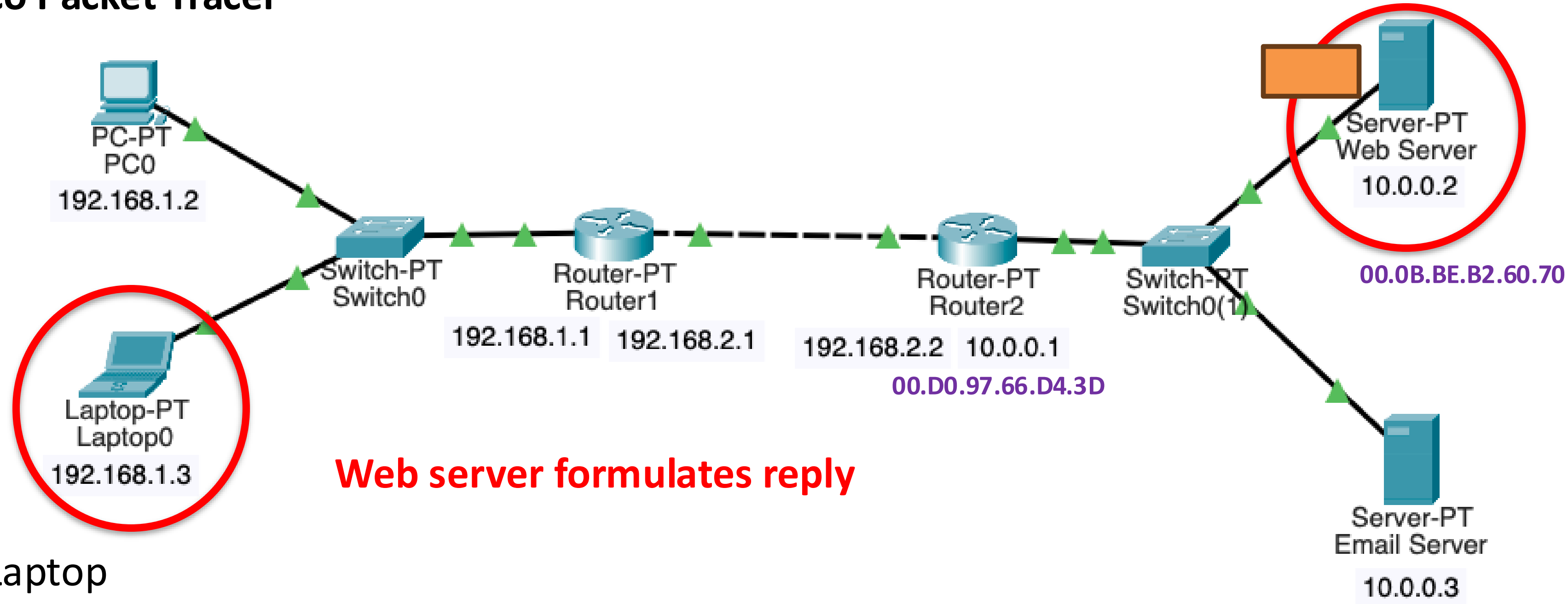
Frame arrives at Web Server

Web Server

- **Checks Layer 2 is its MAC**
  - **Strips off Layer 2 header**
- **Checks Layer 3 IP address**
  - **Stores src IP (Laptop at 192.168.1.3)**
  - **Strips off Layer 3 header**
- **Checks Layer 4**
  - **Notes dst port (80=web)**
  - **Stores sport (1026=Laptop) for return**
  - **Strips off Layer 4 header**
- **Sends data to web app**

Internet largely uses a client/server model where clients make requests to servers

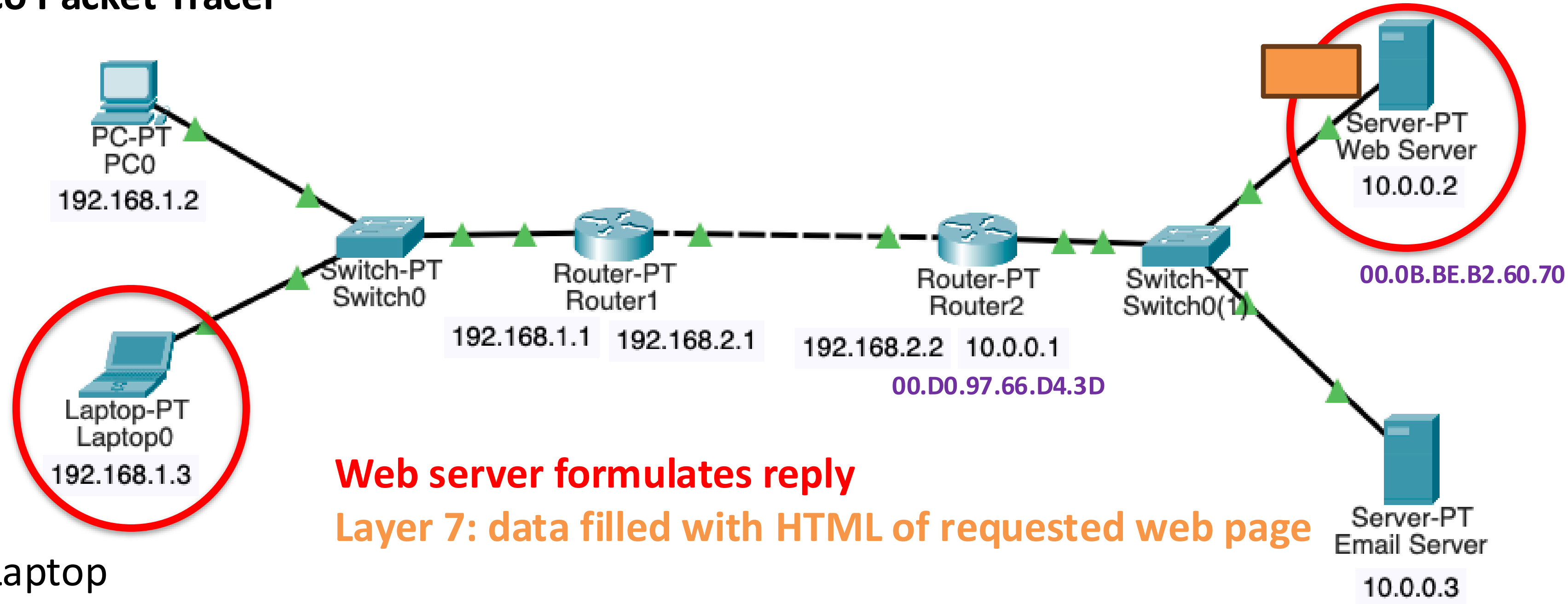
Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

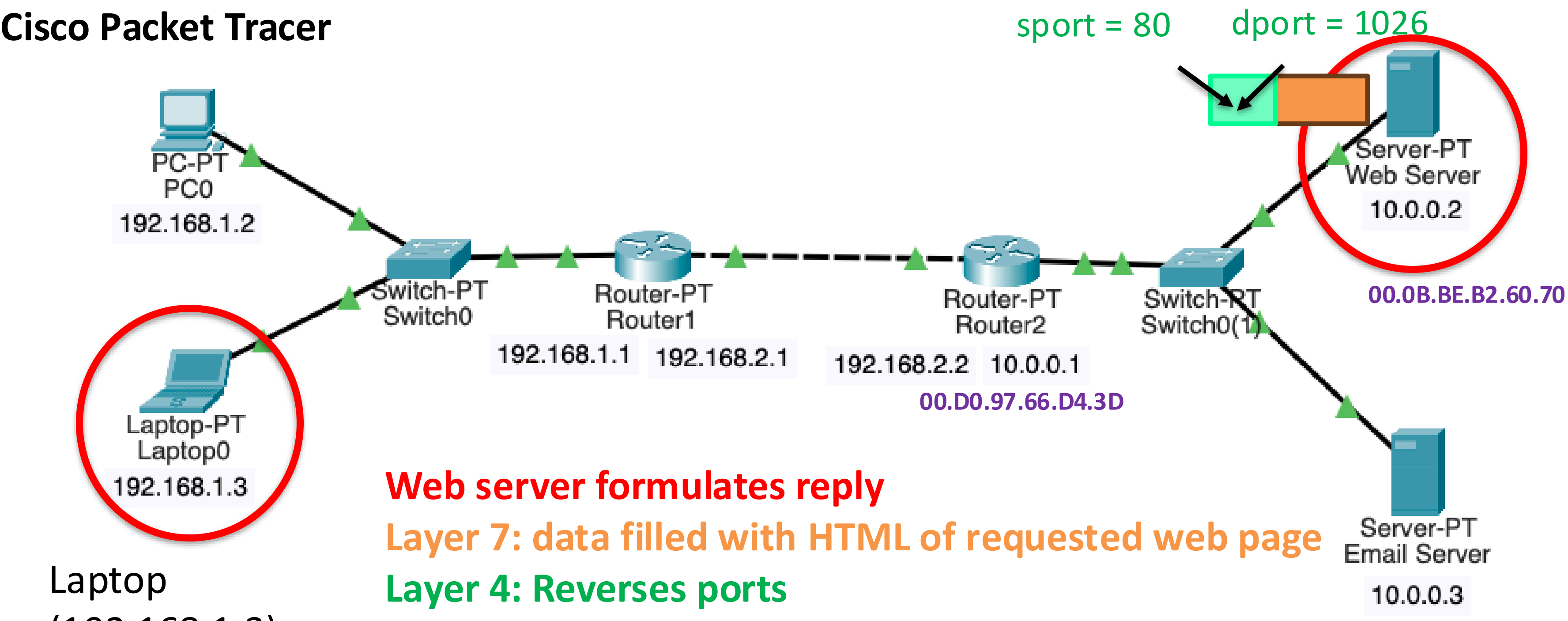
Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Web server formulates reply

Layer 7: data filled with HTML of requested web page

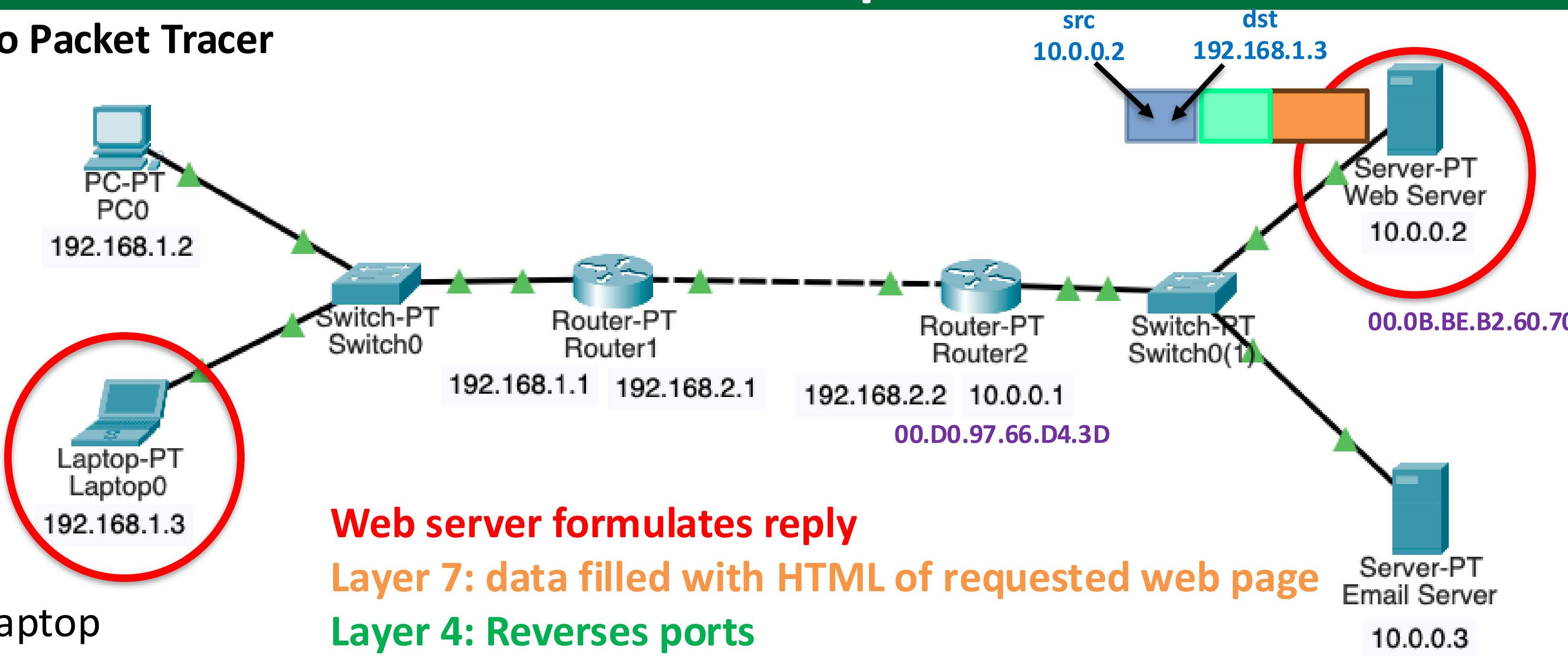
Layer 4: Reverses ports

- **sport = 80 (original dport)**
- **dport = 1026 (original sport)**

Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Web server formulates reply

Layer 7: data filled with HTML of requested web page

Layer 4: Reverses ports

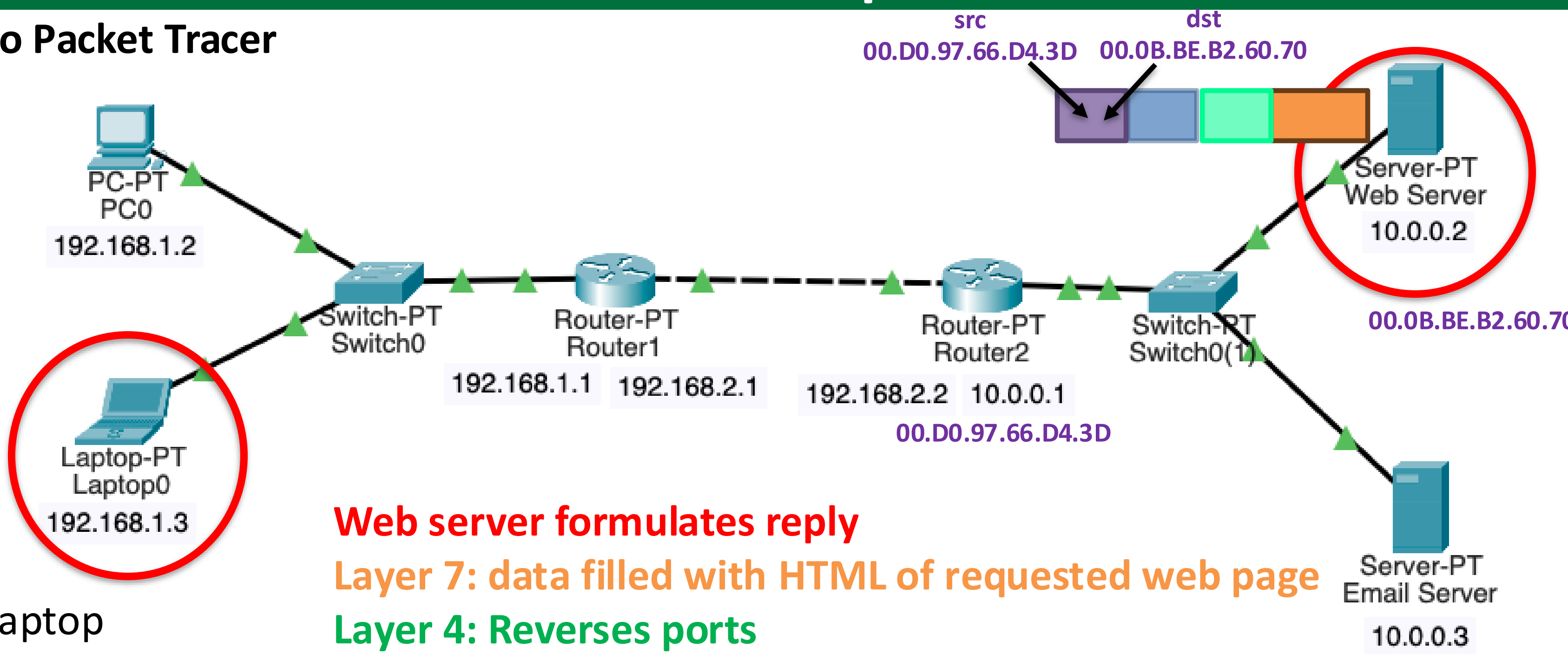
- **sport = 80 (original dport)**
- **dport = 1026 (original sport)**

Layer 3: Reverses IP addresses

- **src = 10.0.0.2 (Web Server)**
- **dst = 192.168.1.3 (Laptop)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Web server formulates reply

Layer 7: data filled with HTML of requested web page

Layer 4: Reverses ports

- **sport = 80 (original dport)**
- **dport = 1026 (original sport)**

Layer 3: Reverses IP addresses

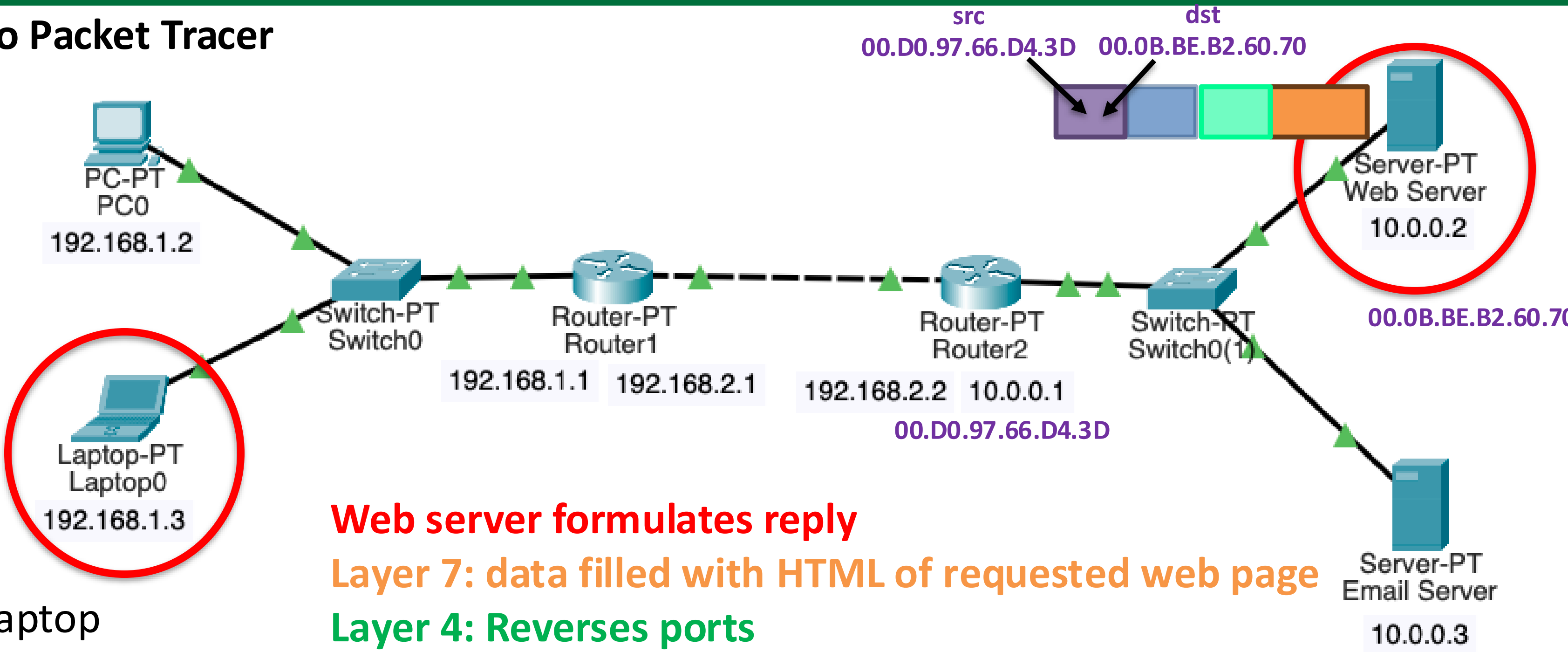
- **src = 10.0.0.2 (Web Server)**
- **dst = 192.168.1.3 (Laptop)**

Layer 2: Reverses MAC addresses

- **src = 00.0B.BE.B2.60.70 (Web Server)**
- **dst = 00.D0.97.66.D4.3D (Router2)**

Internet largely uses a client/server model where clients make requests to servers

Cisco Packet Tracer



Laptop (192.168.1.3) makes a request for a web page from the Web Server (10.0.0.2)

Web server formulates reply

Layer 7: data filled with HTML of requested web page

Layer 4: Reverses ports

- sport = 80 (original dport)
- dport = 1026 (original sport)

Layer 3: Reverses IP addresses

- src = 10.0.0.2 (Web Server)
- dst = 192.168.1.3 (Laptop)

Layer 2: Reverses MAC addresses

- src = 00.0B.BE.B2.60.70 (Web Server)
- dst = 00.D0.97.66.D4.3D (Router2)

Reply routed back to Laptop

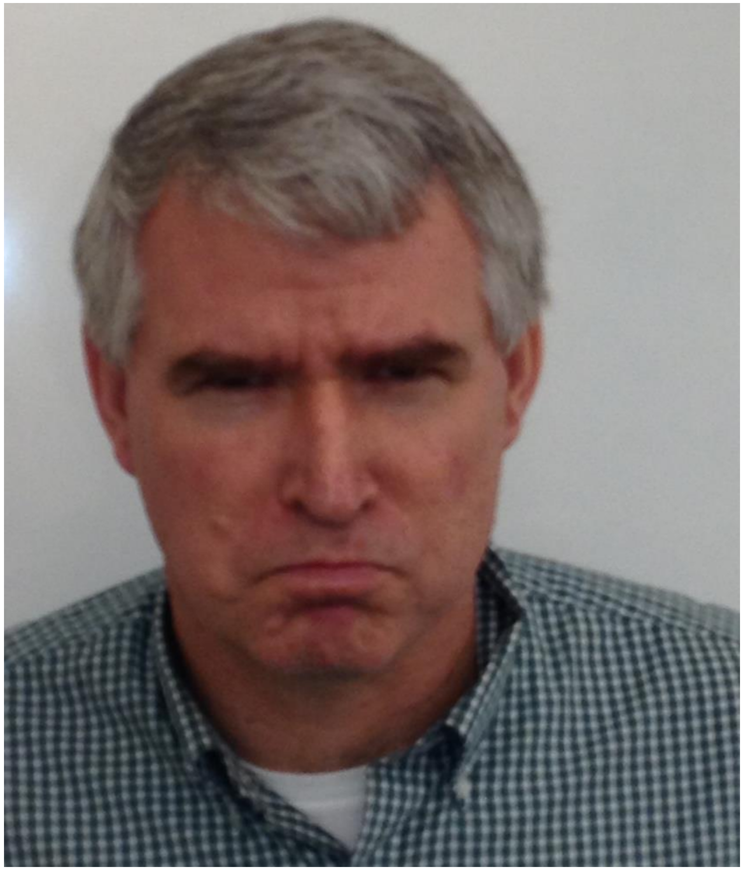
Can the Server get the Laptop's MAC?

No! Router1 stripped it out and overwrote it

I made one simplification and didn’t address NAT

The local network probably uses Network Address Translation (NAT)

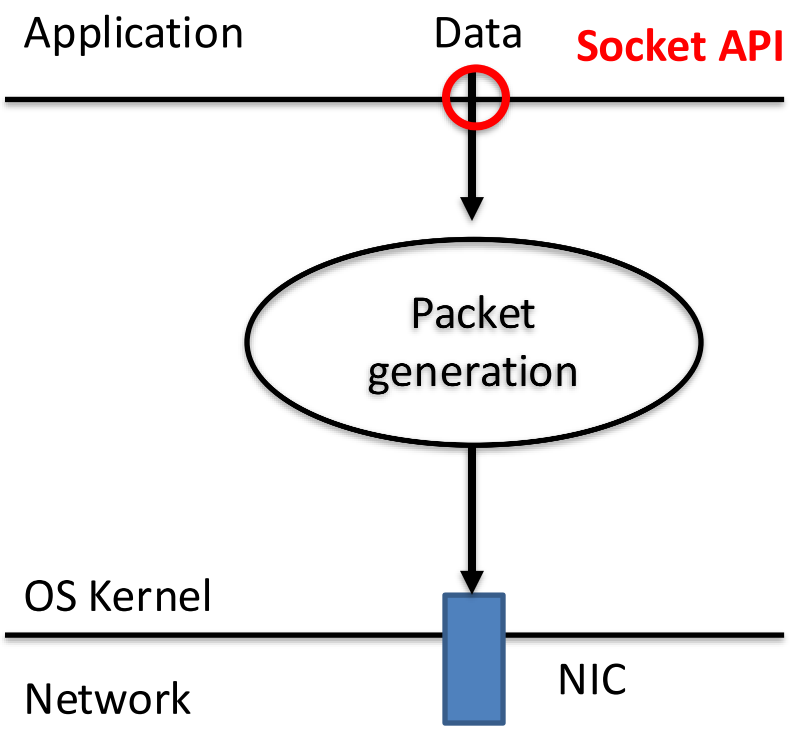
- The client’s 192.168.1.0/24 address wasn’t what was used for routing on the Internet (it is non-routable)
- We will cover NAT soon and why we use it (short answer: we ran out of IP addresses)



Agenda

1. Review: network layers
2. Sockets
3. Scapy
4. Exercises

Socket: an endpoint for two-way comms between programs running on a network



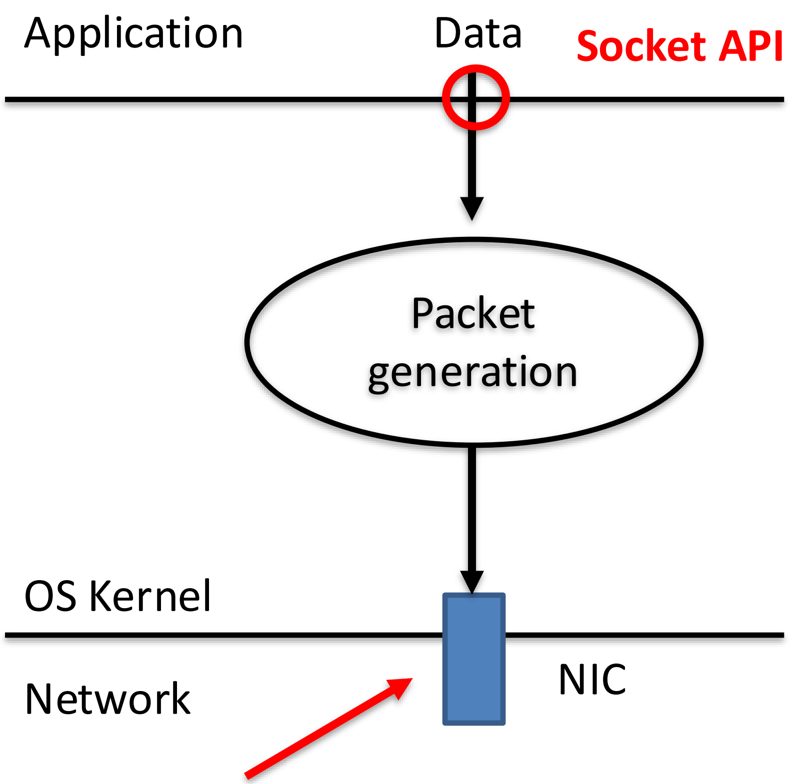
Sockets are defined by:

- IP address
- Port
- Protocol (TCP or UDP)

Sockets are an interface for an application to send or receive data from the network

Adapted from Du: Computer and Internet Security

Typically, the OS handles packet construction



Application has data that it wants to send

Application passes data to OS via a socket

OS generates a packet with Transport, Network, and Link Layer headers

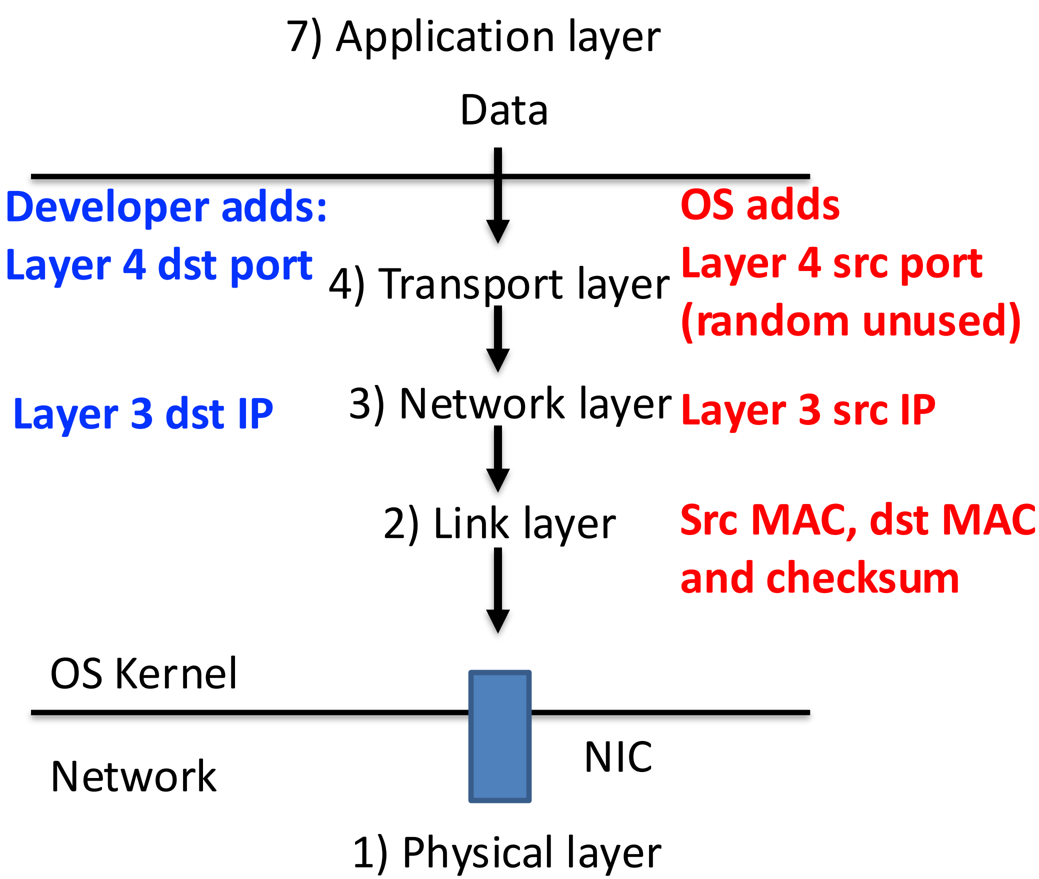
Packet sent out over network via NIC

How does the application hand data to OS?
OS provides socket API system calls
Recall sockets from CS10 and CS50

Each NIC has unique MAC address
Drops packets not addresses to itself
(unless promiscuous or monitor mode)

Adapted from Du: Computer and Internet Security

Applications must provide protocol and destination information



Application developer must decide which transport protocol to use:

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

Developer must also provide *destination* information

- IP address (Layer 3)
- Port (Layer 4)

OS adds *source* port and IP, MAC and checksum

Adapted from Du: Computer and Internet Security

Sending a packet using a socket is not difficult with Python

send_udp.py

We will use a socket (as with CS10 and CS50)

```
import socket

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

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

Socket will use IPv4

This Socket will use UDP protocol
Set to socket.SOCK_STREAM for TCP

Application's data (in bytes) to send

Send 'Hello world'

To destination IP address
(localhost = 127.0.0.1)

Using port 9090

OS handles adding Transport, Network, and Link Layer headers to applications data
OS sends headers and data over Physical Layer 1 via NIC

Sending a packet using a socket is not difficult with Python

send_udp.py

Python hides a lot of complexity!

```
import socket

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


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

Note: we *do not* set a source port on the client side
OS picks a random unused port for us
We *do* set a port on the server side
Client needs to know what port the server is listening on

Run program

Terminal 1

```
$ python3 send_udp.py
```

Listen for UDP in a terminal on port 9090 using netcat (nc)

Terminal 2

```
$ nc -luv 9090
Hello world!
```

netcat receives UDP message when sent

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

send_udp.c

```c
int 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);
}
```

Python built on top of C libraries

Create socket for UDP

Use IPv4

Set destination info

Send

Adapted from Du: Computer and Internet Security

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

send_udp.c

```c
int 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,

    clo
}
```

Python built on top of C libraries

Create socket for UDP

Use IPv4

Set destination info

Send

Terminal 1

Compile and run

```
$ gcc gcc send_udp.c -o send
$ ./send
```

Listen for UDP in a terminal

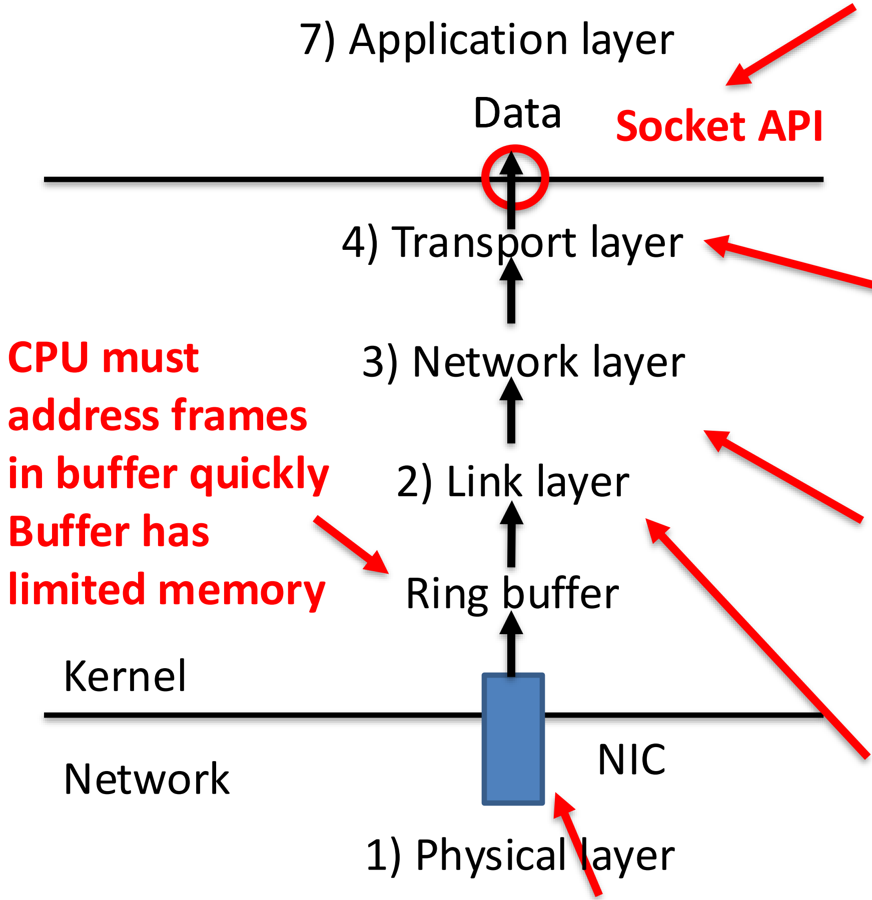
Terminal 2

```
$ nc -luv 9090
Hello World (in C)!
```

netcat receives UDP message when sent

Receiving a packet via a Socket 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 for data

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

Packet received, looks for match with own IP address
Pass to Layer 4 if IP matches

Link layer (layer 2) driver takes frame from ring buffer
Strips off link-layer header
Pass to layer 3 if MAC matches

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

Adapted from Du: Computer and Internet Security

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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


	while (True):
		msg, (ip, port) = sock.recvfrom(MAX_SIZE)
		print(f"{ip}:{port} {msg.decode('utf-8')}")
```

Set up socket for UDP as before

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f"{ip}:{port} {msg.decode('utf-8')}")
```

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

Set up socket for UDP as before

Bind socket to listen on ip_addr and port
Note: we pick a port on the server (receiver) side

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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

      while (True):
            msg, (ip, port) = sock.recvfrom(MAX_SIZE)
            print(f"{ip}:{port} {msg.decode('utf-8')}")
```

Loop forever

Block until packet arrives recvfrom gets data here print client ip, port, and msg

Use recvfrom for UDP
Use recv for TCP

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f'{ip}:{port} {msg.decode('utf-8')}')
```

Terminal 1 (receiver)

$ python3 receive_udp.py

Terminal 2 (sender)

$ nc –u 127.0.0.1 9090

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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


	while (True):
		msg, (ip, port) = sock.recvfrom(MAX_SIZE)
		print(f'{ip}:{port} {msg.decode('utf-8')}')
```

Terminal 1 (receiver)

$ python3 receive_udp.py

Sender types msg

Terminal 2 (sender)

$ nc –u 127.0.0.1 9090
hello

UDP receiver in Python using a Socket

receive_udp.py

```
import socket

MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port

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

    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f'{ip}:{port} {msg.decode('utf-8')}')
```

Receiver gets msg, ip addr, and port

Sender types msg

Terminal 1 (receiver)

```
$ python3 receive_udp.py
127.0.0.1:64369 hello
```

Terminal 2 (sender)

```
$ nc –u 127.0.0.1 9090
hello
```

UDP receiver in Python using a Socket

receive_udp.py

We did not pick a source port on the sender (Terminal 2)
OS chooses source port randomly
When this is run, see *source* port other than 9090 (it was 64369 here)
On *receiver*, OS can sort out replies if multiple instances of application are running at the same time based on port

```
import socket

MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f'{ip}:{port} {msg.decode('utf-8')}')
```

Receiver gets msg, ip addr, and port

Sender types msg

Terminal 1 (receiver)
$ python3 receive_udp.py
127.0.0.1:64369 hello

Terminal 2 (sender)
$ nc –u 127.0.0.1 9090
hello

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


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


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f'{ip}:{port} {msg.decode('utf-8')}')
```

Run with two senders on the same computer (localhost)
Each sender got different source port
Receiver can tell them apart

Terminal 1 (receiver)

```
$ python3 receive_udp.py
127.0.0.1:53072 hello
127.0.0.1:52815 sup?
```

Terminal 2 (sender)

```
$ nc –u 127.0.0.1 9090
hello
```

Terminal 3 (sender)

```
$ nc –u 127.0.0.1 9090
sup?
```

UDP receiver in C using Sockets is not much different from Python

receive_udp.c

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

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

   // 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(EXIT_FAILURE);
   }

   // Getting captured packets
   while (1) {
      bzero(buf,MAX_SIZE);
      recvfrom(sock, buf, MAX_SIZE-1, 0, (struct sockaddr *) &client, &clientlen);
      //get ip address as string
      inet_ntop(AF_INET, &(client.sin_addr), ip, INET_ADDRSTRLEN);
      printf("%s:%i %s\n",ip, client.sin_port,buf);
   }
   close(sock);
   return(EXIT_SUCCESS);
}
```

Create socket named sock:

- **AF_INET = IPv4**
- **SOCK_DGRAM = UDP**

Create server and fill memory with zeros

Set server:

- **AF_INET = IPv4**
- **INADDR_ANY = use any interface like Python 0.0.0.0**
- **Port = 9090**

Bind socket to server

Loop forever

Msg buffer and client info filled when packets arrive

Adapted from Du: Computer and Internet Security

UDP receiver in C using Sockets is not much different from Python

receive_udp.c

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

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

   // 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(EXIT_FAILURE);
   }

   // Getting captured packets
   while (1) {
      bzero(buf,MAX_SIZE);
      recvfrom(sock, buf, MAX_SIZE-1, 0, (struct sockaddr *) &client, &clientlen);
      //get ip address
      inet_ntop(AF_INET, &(client.sin_addr), ip, INET_ADDRSTRLEN);
      printf("%s:%i %s\n",ip, client.sin_port,buf);
   }
   close(sock);
   return(EXIT_SUCCESS);
}
```

Terminal 1 (receiver)

```
$ gcc receive_udp.c -o receive
./receive
```

UDP receiver in C using Sockets is not much different from Python

receive_udp.c

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

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

   // 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(EXIT_FAILURE);
   }

   // Getting captured packets
   while (1) {
      bzero(buf,MAX_SIZE);
      recvfrom(sock, buf, MAX_SIZE-1, 0, (struct sockaddr *) &client, &clientlen);
      //get ip address
      inet_ntop(AF_INET, &(client.sin_addr), ip, INET_ADDRSTRLEN);
      printf("%s:%i %s\n",ip, client.sin_port,buf);
   }
   close(sock);
   return(EXIT_SUCCESS);
}
```

Terminal 1 (receiver)

```
$ gcc receive_udp.c -o receive
./receive
```

Terminal 2 (sender)

```
$ nc –u 127.0.0.1 9090
hello
```

UDP receiver in C using Sockets is not much different from Python

receive_udp.c

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

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

  // 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(EXIT_FAILURE);
  }

  // Getting captured packets
  while (1) {
    bzero(buf,MAX_SIZE);
    recvfrom(sock, buf, MAX_SIZE-1, 0, (struct sockaddr *) &client, &clientlen);
    //get ip address
    inet_ntop(AF_INET, &(client.sin_addr), ip, INET_ADDRSTRLEN);
    printf("%s:%i %s\n",ip, client.sin_port,buf);
  }
  close(sock);
  return(EXIT_SUCCESS);
}
```

Terminal 1 (receiver)

```
$ gcc receive_udp.c -o receive
./receive
127.0.0.1:14809 hello
```

Terminal 2 (sender)

```
$ nc –u 127.0.0.1 9090
hello
```

Notice that sender got random, OS-assigned port

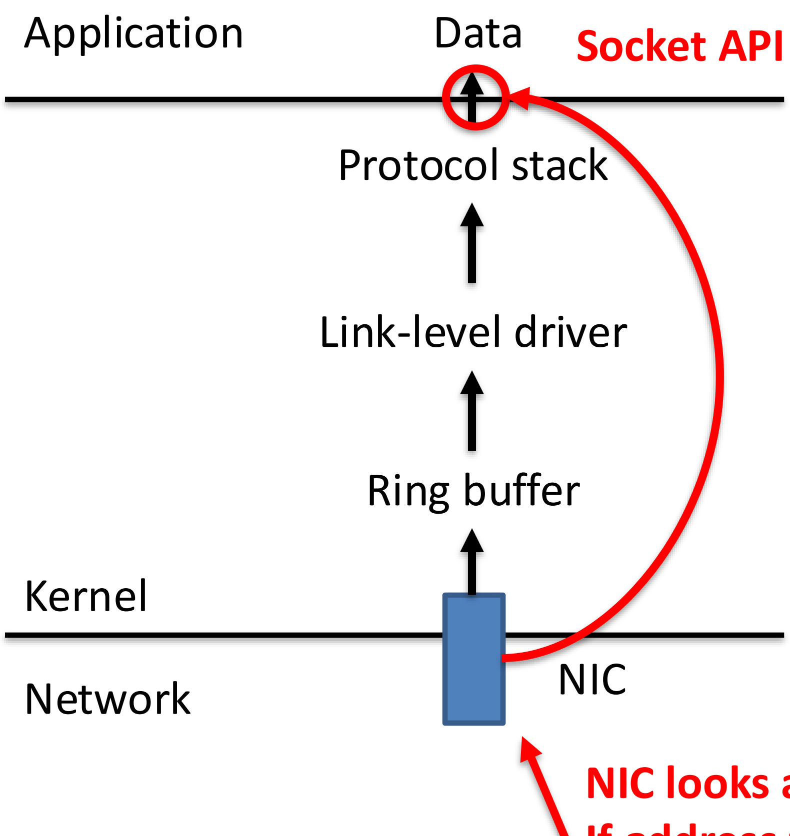
Can a receiver written in C receive packets from a sender written in Python?

Yes!

We also do not care if one computer is a Mac and the other is Windows

Promiscuous (monitor) mode sniffs all frames

Monitor mode only sniffs one channel



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

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

New Mac do not go into promiscuous mode! ☹

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

Adapted from Du: Computer and Internet Security

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

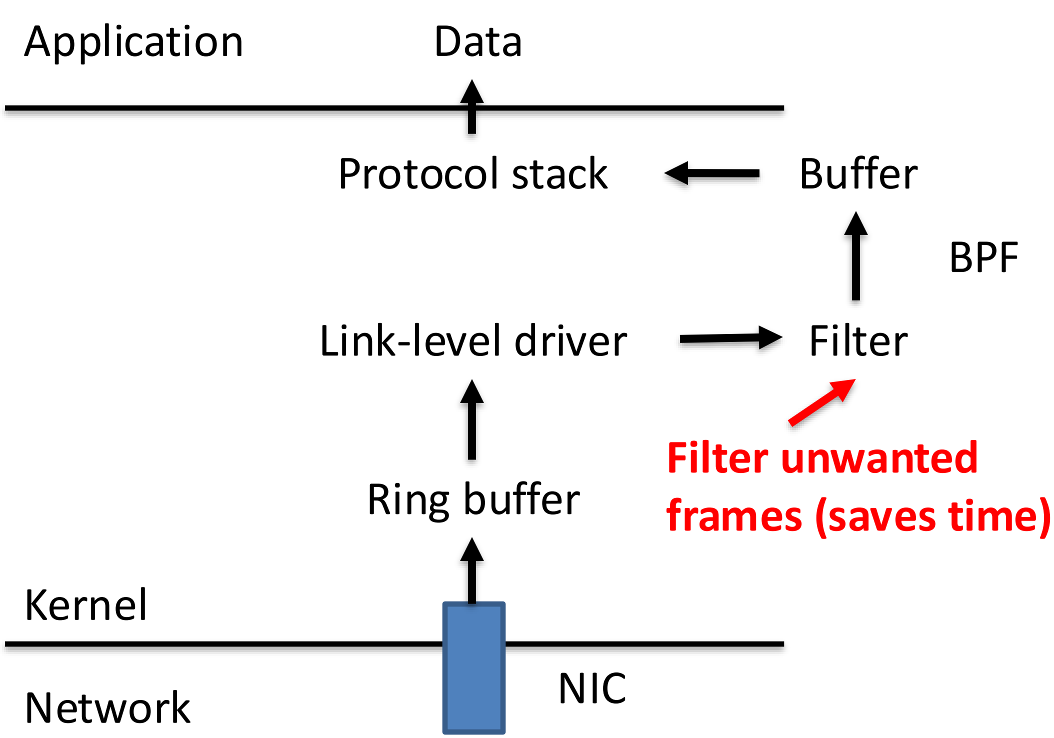
Set socket option enables promiscuous mode

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



NIC looks at incoming frames
If addressed to this MAC
DMA to kernel ring buffer
Wake up CPU

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!

Adapted from Du: Computer and Internet Security

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("en0", 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

Open PCAP session

- **Sniff on interface *en0* (use ifconfig to find)**
- **3rd parameter sets promiscuous mode to true**

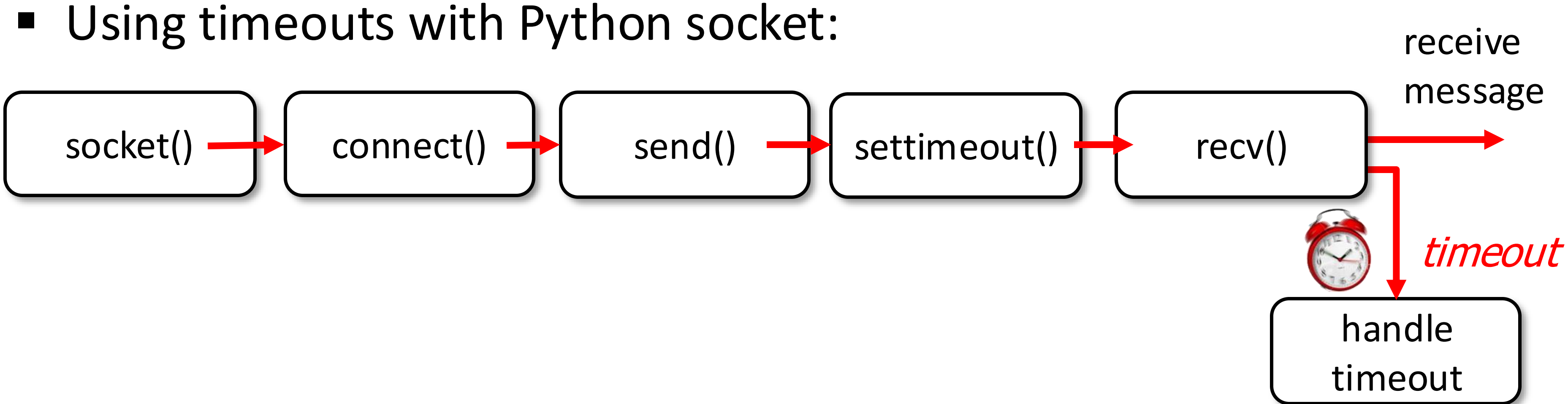
Parsing packets in C is tedious
Why?
Layer 2, 3, and 4 headers not constant size (optional components)

Compile BPF filter to pass UDP or ICMP packets

Start sniffing
Set callback function to *got_packet*

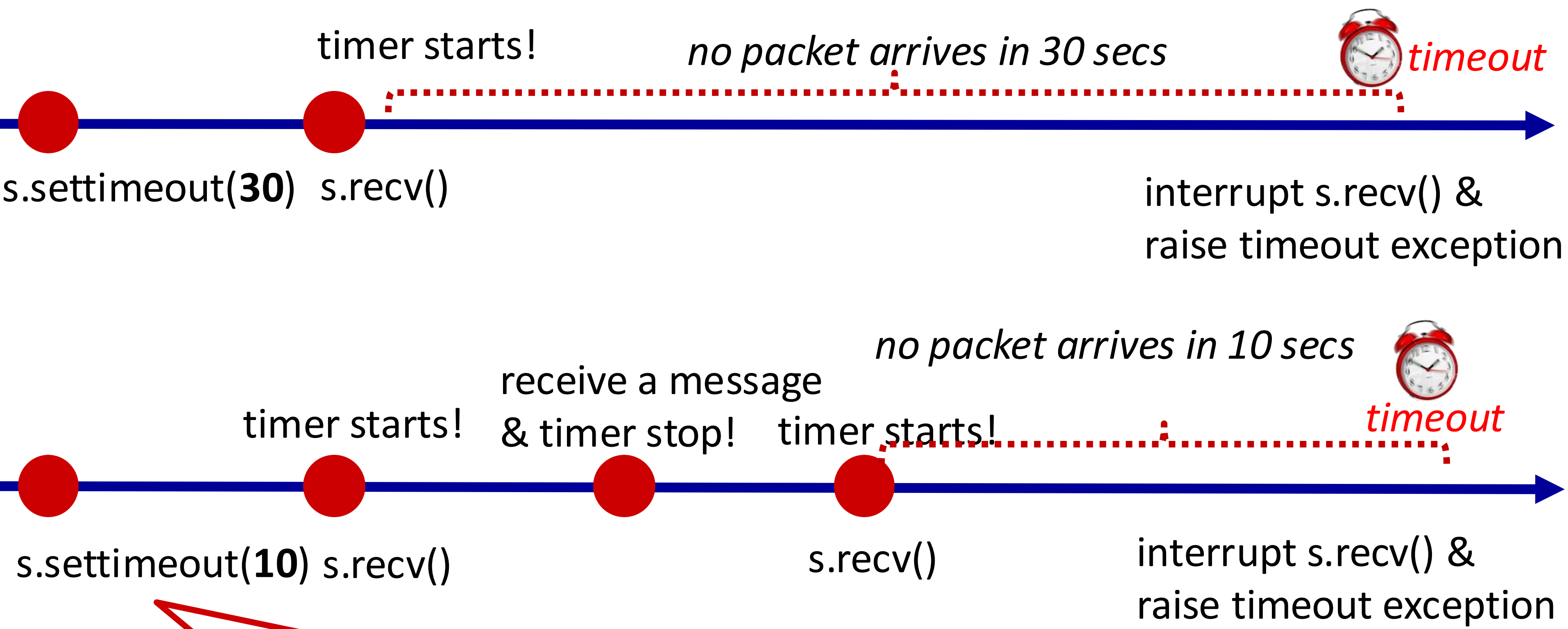
We can set timeouts on sockets

- Sometimes a program must wait for one of several events to happen, e.g.,:
  - Wait for either (i) a reply from another end of the socket, or (ii) timeout: timer
  - Wait for replies from several different open sockets: select(), multithreading
- Timeouts are used extensively in networking
- Using timeouts with Python socket:



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Sockets timeout if no message is received



Set a timeout on all future socket operations of that specific socket!

Timeout exception raised if no response from server in 5 seconds

settimeout.py

```
# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Set a timeout of 5 seconds for the socket operations
sock.settimeout(5)

# Example usage (sending and receiving data with timeout)
serverAddressPort = ("127.0.0.1", 9090)
bytesToSend = str.encode("Hello UDP Server")

try:
   sock.sendto(bytesToSend, serverAddressPort)
   msgFromServer, address = sock.recvfrom(1024)
   print(f"Message from server: {msgFromServer.decode()}")
except socket.timeout:
   print("Socket operation timed out.")
except socket.error as e:
   print(f"Socket error: {e}")
finally:
   sock.close()
```

Set timeout to 5 seconds

Begin timer on recvfrom
Throw exception if reply not received in 5 seconds

Catch exception thrown if no reply

Agenda

1. Review: network layers
2. Sockets
3. Scapy
4. Exercises

Scapy makes it easy to sniff and transmit

Scapy
A Python-based packet manipulation program

Enables:

- Packet crafting (build any kind of packet)
- Sniffing (capture packets from the wire)
- Sending (inject custom packets)
- Dissection (decode fields of captured packets)

Works at Layers 2–7 (Ethernet to Application)

Takes care of details like calculating checksums for you!

Requires root privileges to run:
```
$ sudo python3 prog.py
```

Install with:
```
$ pip3 install scapy
```

Scapy uses Sockets under the hood but gives you control over header fields

Uses raw sockets for full control over network frames

Can list fields available at each network level with *ls(<layer name>)*

```
$ sudo python3
>>> from scapy.all import *
>>> ls(Ether)
dst    : DestMACField      = ('None')
src    : SourceMACField    = ('None')
type   : XShortEnumField   = ('36864')
>>> ls(IP)
version : BitField  (4 bits) = ('4')
ihl     : BitField  (4 bits) = ('None')
tos     : XByteField         = ('0')
len     : ShortField         = ('None')
id      : ShortField         = ('1')
flags   : FlagsField         = ('<Flag 0 ()>')
frag    : BitField  (13 bits)= ('0')
ttl     : ByteField          = ('64')
proto   : ByteEnumField      = ('0')
chksum  : XShortField        = ('None')
src     : SourceIPField      = ('None')
dst     : DestIPField        = ('None')
options : PacketListField    = ('[]')
```

```
>>> ls(TCP)
sport      : ShortEnumField     = ('20')
dport      : ShortEnumField     = ('80')
seq        : IntField           = ('0')
ack        : IntField           = ('0')
dataofs    : BitField  (4 bits) = ('None')
reserved   : BitField  (3 bits) = ('0')
flags      : FlagsField         = ('<Flag 2 (S)>')
window     : ShortField         = ('8192')
chksum     : XShortField        = ('None')
urgptr     : ShortField         = ('0')
options    : TCPOptionsField    = ("b''")
>>>
```

Scapy calculates checksums and other fields for you!
Chooses "reasonable" values for fields

You can set or read any of these fields

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

sniff_scapy.py

Scapy built on top of PCAP

```
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='en0', 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

Sniff on this interface

Filter out packets other than these

Only capture 10 packets

Run

```
$ sudo python3 sniff_scapy.py
```

Another terminal

```
$ ping 8.8.8.8
```

Uses promiscuous mode, so need sudo privilege

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

sniff_scapy.py

```
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='en0', filter='icmp or udp'
```

Run

```
$ sudo python3 sniff_scapy.py
```

```
Packet number 0
###[ Ethernet ]###
  dst       = 00:50:56:f5:9e:21
  src       = 00:0c:29:a8:d0:96
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 54952
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xf3c6
     src       = 192.168.159.129
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xa818
        id        = 0xb
        seq       = 0x1
        unused    = b''
###[ Raw ]###
           load      =
b'\x86\xb8\xd1h\x00\x00\x00\x00*\xe7\x0e\x00\x00\x00\x00\x00\x10\x
11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./01234567'
```

Demo: packet spoofing is easy with Scapy

spoof_udp.py

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

We could have (but didn't here) create an Ether layer and spoof Layer 2 MAC addresses

You will do this in Lab 1

```
#!/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)
```

Create IP layer 3 with spoofed src and dest

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

Send packet

Stack layers to create UDP over IP layer with data

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

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

Demo: packet spoofing is easy with Scapy

spoof_udp.py

Terminal 1 (sender)

$ sudo python3 spoof_udp.py

Terminal 2 (sender)

$ sudo python3 sniff_scapy.py ens160

```
#!/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 L
data = "Hello UDP!\n"                   # Payload
pkt = ip/udp/data      # Construct the complete pac
pkt.show()
send(pkt,verbose=0)
```

```
###[ Ethernet ]###
  dst       = 00:50:56:f5:9e:21
  src       = 00:0c:29:a8:d0:96
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 39
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = udp
     chksum    = 0x6abb
     src       = 1.2.3.4
     dst       = 10.0.2.5
     \options   \
###[ UDP ]###
        sport     = 8888
        dport     = 9090
        len       = 19
        chksum    = 0xd62b
###[ Raw ]###
           load   = b'Hello UDP!\n'
```

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)
```

Create a phantom computer on the network that responds to pings from 10.0.2.15!

Steps

- **Receive packet**
- **Extract details**
- **Spoof a new packet**
  - **Reverse src and dst**
  - **Keep same id and seq number**

Send a ICMP reply after ping request to 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

Agenda

1. Review: network layers
2. Sockets
3. Scapy
4. Exercises

Exercise

Implement a chat program where:

- VPN to Dartmouth's network first (Dartmouth blocks traffic)!
- The client takes input from the keyboard and sends it to the server over UDP
- Server responds back to client with message converted to all uppercase
- Start with send_udp.py/receive_udp.py

Time permitting

- Implement the same in C
- Start with send_udp.c/receive_udp.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

DE	AD	BE	EF	

Increasing addresses →

Little endian

EF	BE	AD	DE	

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 – 191.255.255.255

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 – 191.255.255.255

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 – 191.255.255.255

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 – 191.255.255.255

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