CLIENT SERVER ARCHITECTURE

EN.600.444/644

Spring 2019

Dr. Seth James Nielson

COMPUTING 1960-1980 (ISH)

NETWORK

"DUMB" TERMINAL

MAINFRAME

COMPUTING 1980-2000 (ISH)

COMPUTING 2000 – PRESENT

NETWORK

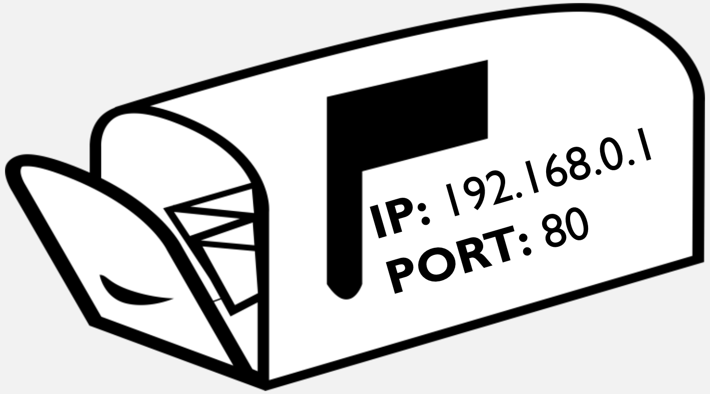
GENERAL IDEAS BEHIND CLIENT-SERVER

- Put a bunch of resources in a high-performance, centralized machine
- Clients can be much “dumber” *by comparison*
- Much more efficient
  - Sharing data between devices, applications, and people (and marketing)
  - Access from multiple locations (including hackers!)
  - Time-sharing a central machine is more scalable and cost-effective

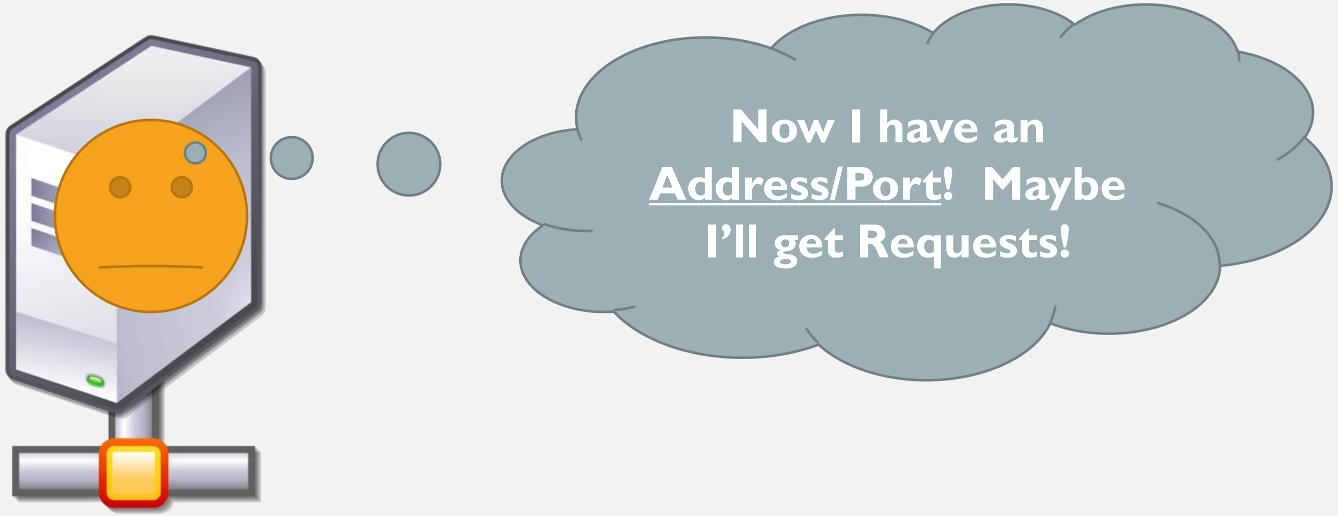
SERVER ABSTRACTION

SERVER ***LISTENS*** FOR INCOMING REQUESTS

PREVIEW OF TCP/IP



This Photo by Unknown Author is licensed under CC BY-NC



This Photo by Unknown Author is licensed under CC BY-SA

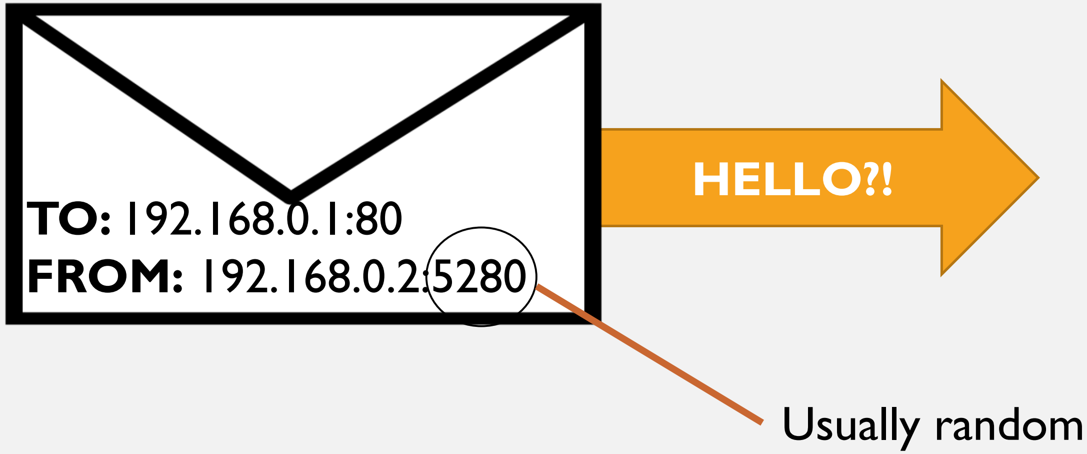
SERVER HAS AN **IP ADDRESS** AND **TCP PORT**

MEANWHILE, CLIENT ABSTACTION

This Photo by Unknown Author is licensed under CC BY-NC

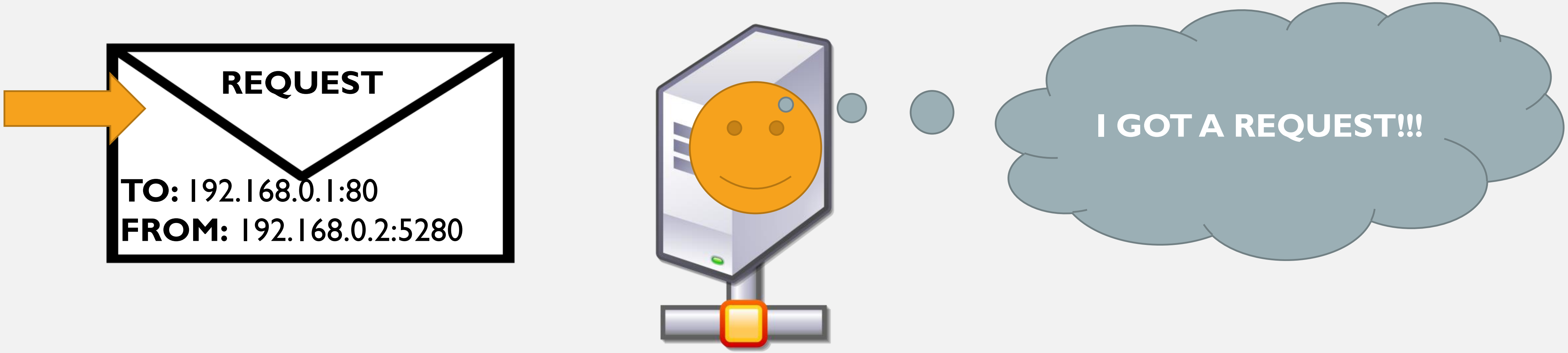
CLIENT ***CONNECTS*** TO MAKE OUTBOUND REQUESTS

TCP/IP AGAIN



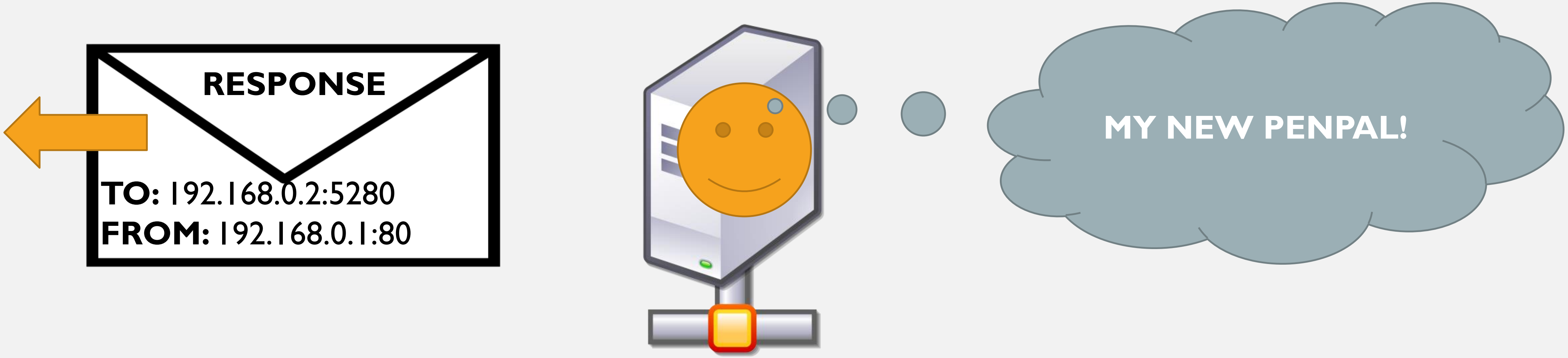
CLIENT ***CONNECTS*** TO MAKE OUTBOUND REQUESTS

INCOMING REQUEST



SERVER RECEIVES REQUEST

REQUEST RESPONSE



SERVER **INVERTS TO/FROM** FOR RESPONSE

SERVER LISTENS TO MANY REQUESTS

SERVER USES **(SRC IP, SRC PORT, DST IP, DST PORT)*** TO MULTIPLEX

This is how one server on one port (e.g., webserver) handles many clients

SOCKETS

- Sockets are a simple abstraction of client-server
- Thus, there is a "client" socket and "server" socket
- The server socket *listens* for incoming connections
- The client socket makes an outbound connection to server
- The server *accepts* the incoming connection and spawns a connected socket

WHAT IS A PROTOCOL?

- A protocol is the set of rules that govern the interaction of two or more parties
- In the context of networking, it defines how two nodes communicate
  - When a party can communicate
  - What a party can communicate, *including message structure*
  - How a party responds to received communications
- ***Certain outcomes or results are guaranteed when the rules are followed***

OVERLOADED TERM

- Actually, a protocol often refers to two separate things
- **FIRST**, the rules/specification referred to on the previous slide
- **SECOND**, the computer module that *implements* the rules

COMMON CONTEMPORARY PROTOCOLS

- HTTP – HyperText Transfer Protocol
- IP – Internet Protocol
- SMTP – Simple Mail Transport Protocol

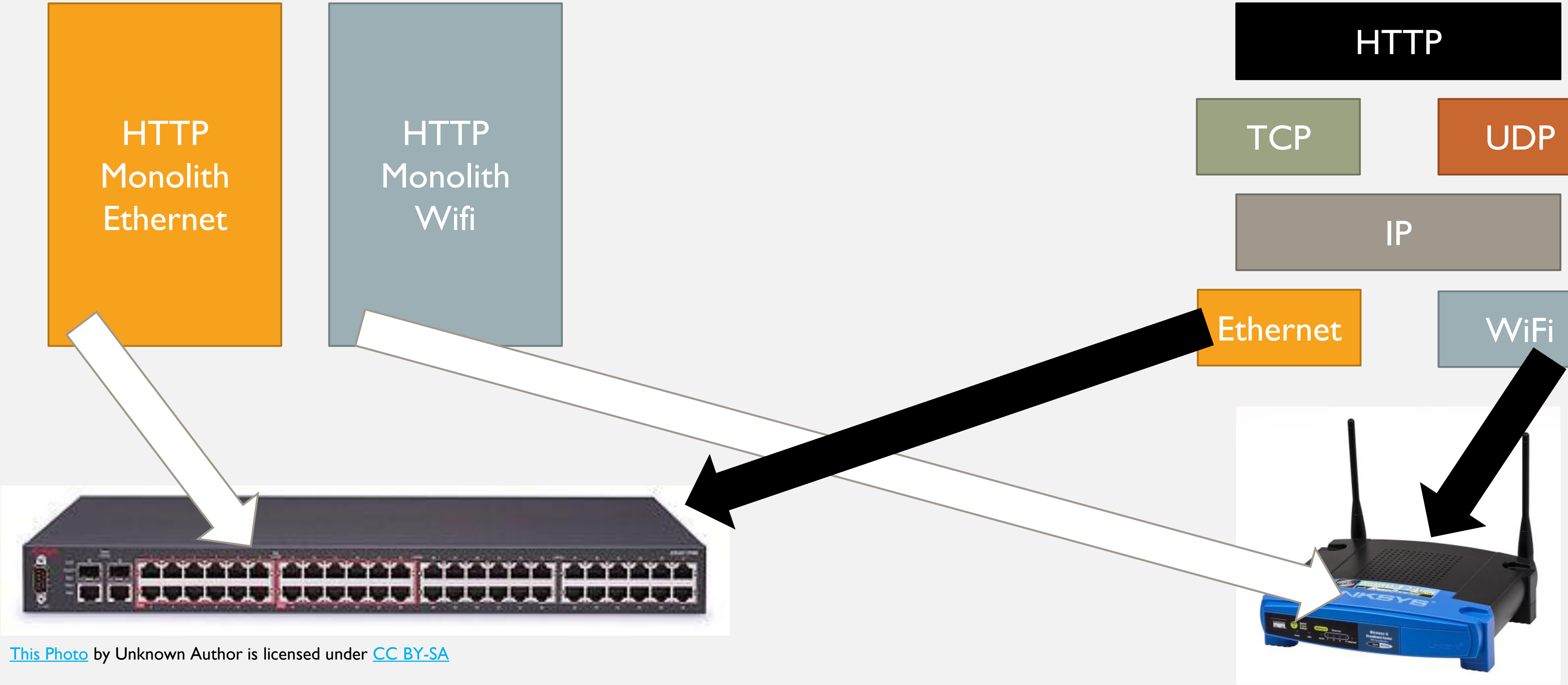
ONE PROTOCOL IS NOT ENOUGH

- There are too many rules for any one protocol to handle
- Also, behavior/rules need to change for different hardware/goals
- For example, consider HTTP
  - HTTP protocol shouldn't need to worry about the IP protocol rules
  - HTTP definitely shouldn't need to worry about Ethernet rules
  - And HTTP should work even after a switch from Ethernet to Wifi

PROTOCOL *STACKS*

- Object-oriented design has been around long before object-oriented programming
  - Modularity
  - Abstraction
  - Information hiding
- Protocols are designed in an object-oriented fashion
  - Protocols are combined to solve more complex problems
  - Each protocol should focus on one purpose/goal (High Cohesion)
  - Different component protocols can be swapped (Low coupling)
- We call a group of protocols that work together a *protocol stack*
- In computer networking, a *network protocol stack* or a *network stack*

MONOLITHIC VS MODULAR

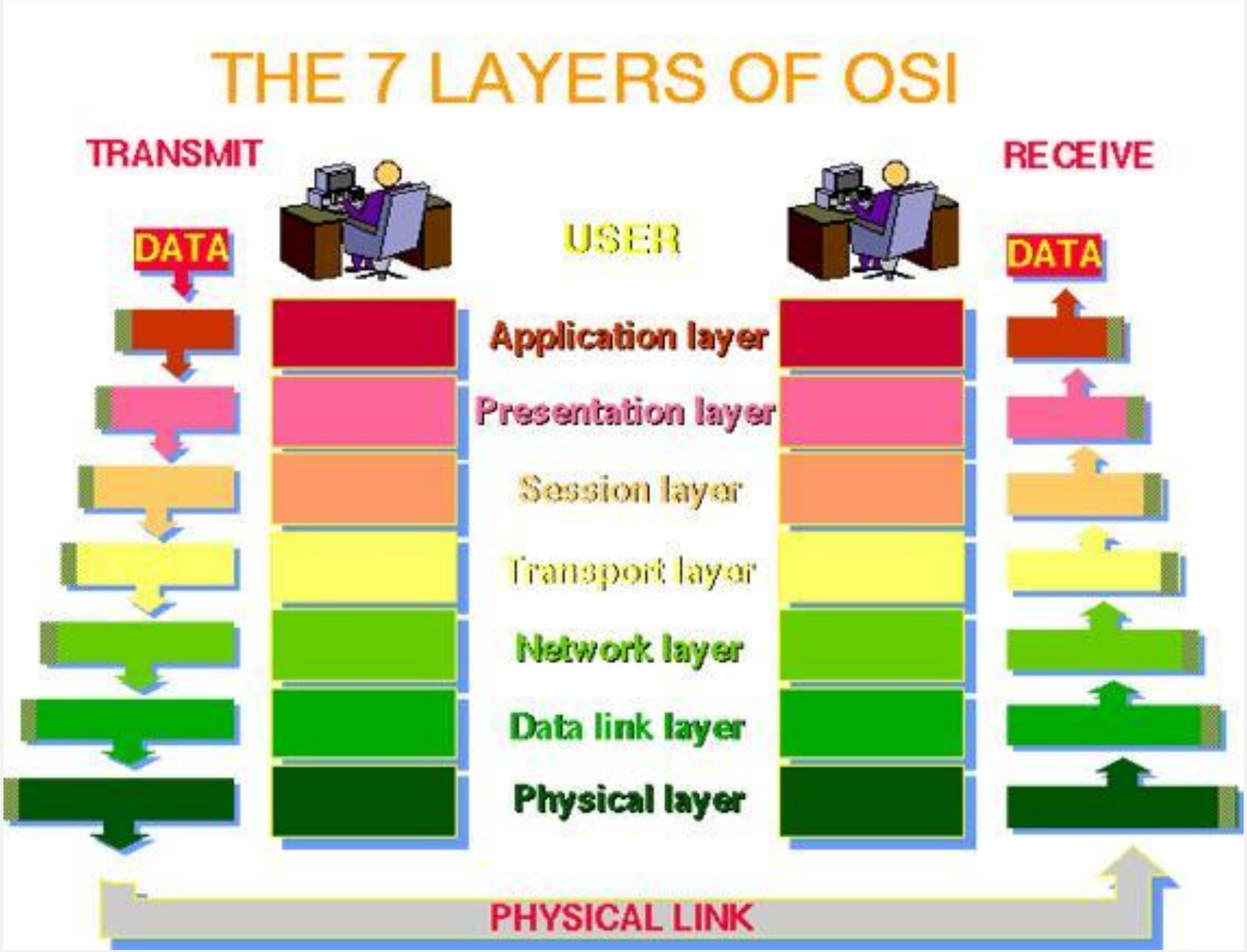


OTHER PROBLEMS WITH MONOLITHIC

- No separation of user/kernel space components
- Code cannot be reused; code bloat
- NxM combinations
- Patching nightmare
- Testing limitations
- List goes on and on

OSI MODEL

- Good object-oriented design is implementation independent
- ISO defined a guide for any given network stack called the OSI Model
- It has seven layers:
  - 7: Application
  - 6: Presentation
  - 5: Session
  - 4: Transport
  - 3: Network
  - 2: Data Link
  - 1: Physical


THE 7 LAYERS OF OSI
TRANSMIT
RECEIVE
DATA
USER
DATA
Application layer
Presentation layer
Session layer
Transport layer
Network layer
Data link layer
Physical layer
PHYSICAL LINK

THE OSI MODEL IN PRACTICE

- Like most OO-designs, the abstraction often breaks down
- Many stacks have multiple protocols in "one layer", and none in another
- Modularity/abstraction/information hiding break down
- The TCP/IP stack really only uses the following layers:
  - Application (Layer 7; example: HTTP)
  - Transport (Layer 4; TCP)
  - IP (Layer 3; IP)
  - Data Link (Layer 2; example: Ethernet)
- NOTE: It's common to just refer to a layer by it's number (e.g., a layer-4 protocol)

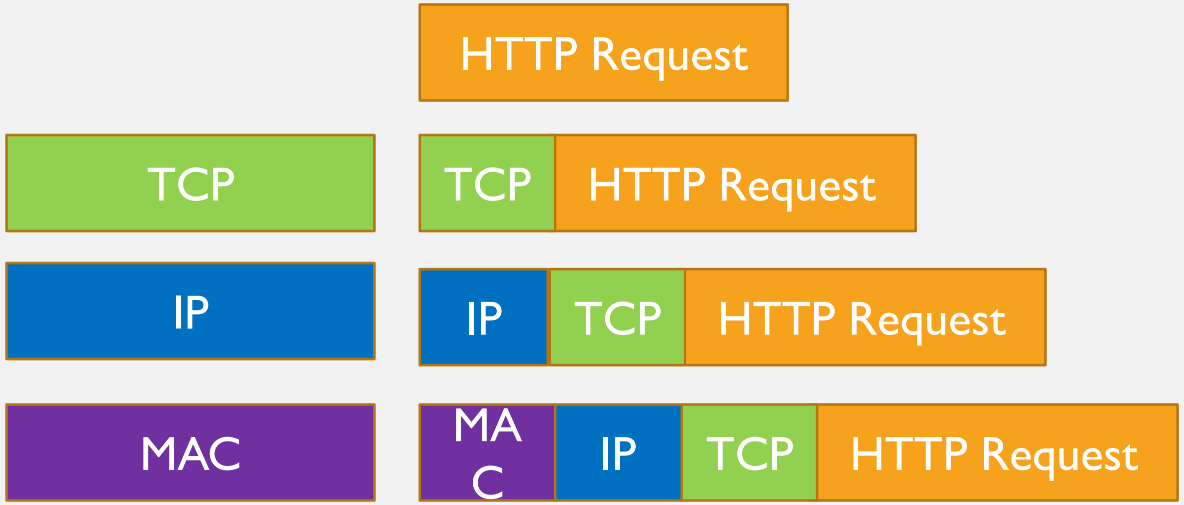
TCP/IP STACK

- For our purposes, we will focus on TCP/IP and TCP/IP-like stacks
- The TCP and IP layers are, obviously, fixed for layers 3 and 4.
- But layers 7 and 2 vary widely
- Millions of networked applications work over TCP/IP at layer 7
- Many layer 2 protocols such as WiFi, Ethernet
  - Networked applications work over WiFi or Ethernet without any change
  - Sometimes called a MAC protocol (Media Access Protocol)
  - TCP/IP work over a walkie-talkie with an appropriate MAC protocol

HOW DOES DATA MOVE IN A STACK?

- To send, data is inserted (pushed) at the top-most protocol
- The receiving protocol
  - Processes the data, potentially splitting, recoding, etc
  - Derives one or more chunks of data
  - Typically affixes a header to each, but sometimes a footer and/or other meta-data
  - Each chunk, along with the meta-data is a "packet"
  - The packet is inserted (pushed) down to the next layer
- When data is received, the process is reversed

TCP/IP STACK EXAMPLE



DIVISION OF LABOR IN TCP/IP

- At the lowest layer, the MAC protocol simply connects two endpoints. Typically:
  - Has its own addressing scheme (MAC address)
  - Controls who talks when
  - Provides error detection and *error correction*
- IP (Internetwork Protocol)
  - Connects many different networks of different media types
  - Global addressing scheme
- TCP
  - Reliable, in-order delivery (Session)
  - Multiplexing

INTEROPERABILITY

- No one company writes all TCP modules; How do they work together?
- Protocol specifications are approved by the IETF (Internet Engineering Task Force)
  - You can find the specifications in RFC's (Request For Comments)
  - RFC 793 was the first specification of TCP (1981)
- So long as an implementation follows the spec, it will be interoperable

RFC 793 OVERVIEW

- Data broken into "segments" in section 2.2
- Network layers in section 2.5 (a little different from our usage)
- Section 2.6 lays out critical goal: Reliability
  - Data is delivered reliably (i.e., delivery is assured)
  - Data is delivered in-order
  - How? Sequence numbers and acknowledgements on segments
- Section 2.7 identifies another goal: Multiplexing
  - Different flows get different ports
- Section 2.8 indicates that this is a *stream* based protocol

TCP Header Format

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |           |U|A|P|R|S|F|                               |
   | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
   |       |           |G|K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Checksum            |         Urgent Pointer        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             data                              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

TCP Header Format

Note that one tick mark represents one bit position.

Figure 3.

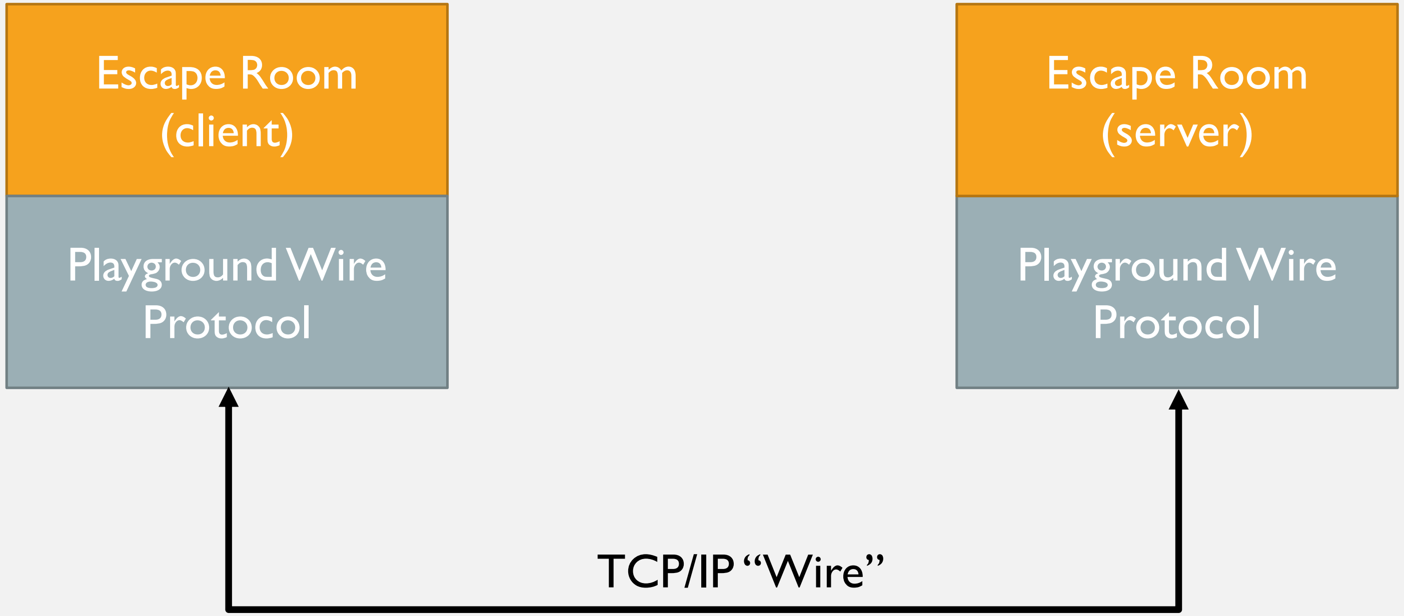
PROTOCOLS AND STATE MACHINES

- It is often useful to model a protocol as a finite state machine (FSM)
  - The protocol starts in an initial state
  - When it receives data, it processes the data and moves to a new state
- For TCP, a state machine is defined in section 3.2
- If you don't know what a FSM is, or how it works, you should probably look it up

PLAYGROUND NETWORKING

- In Playground, we also have a network stack:
  - Application Layer (e.g., Escape Room, Bank)
  - Playground Wire Protocol (Layer2/3)
  - TPC/IP as our Mac Protocol

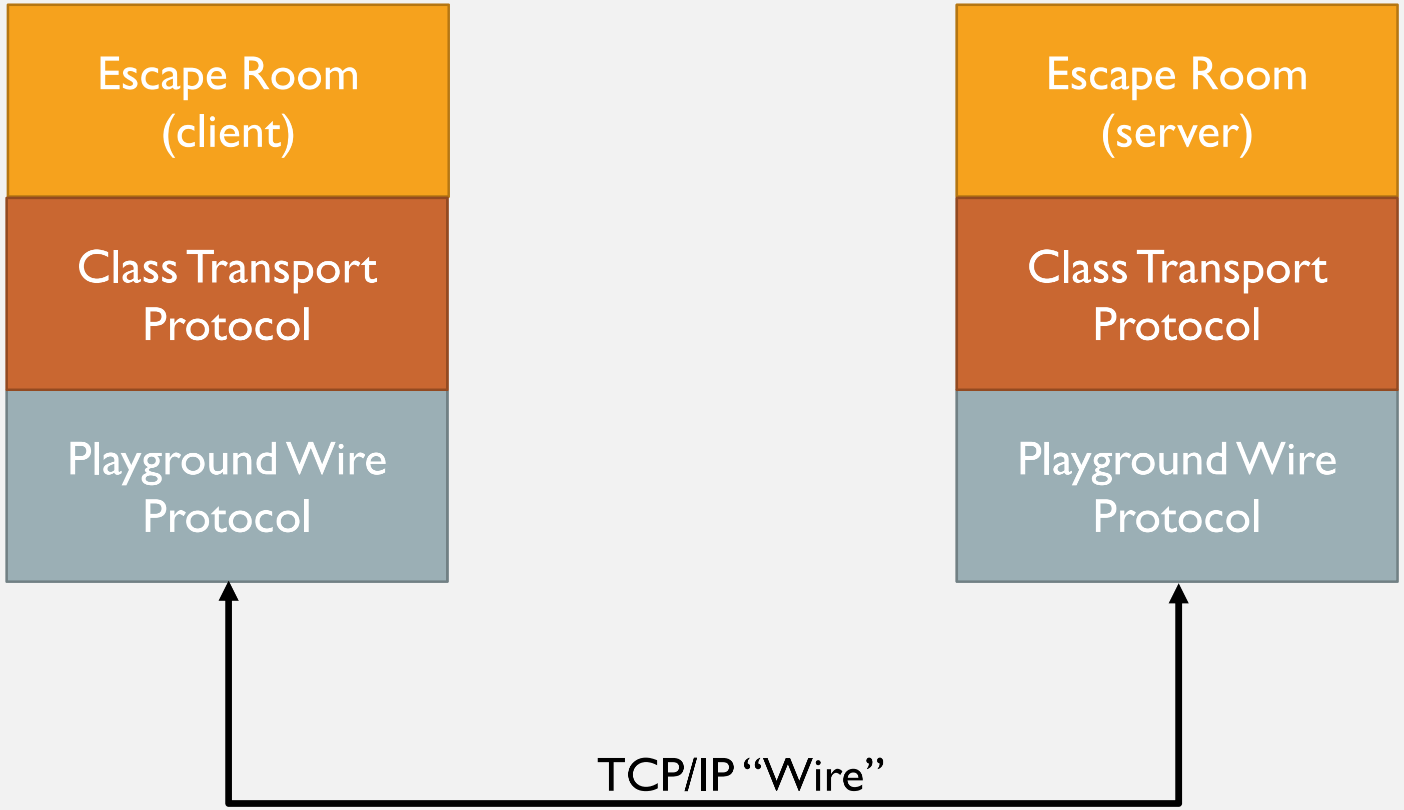
  NOTE: This is important! TCP/IP, for our overlay network, is just a "wire"


Escape Room
(client)
Playground Wire
Protocol
Escape Room
(server)
Playground Wire
Protocol
TCP/IP “Wire”

PLAYGROUND NETWORKING PT 2

- Notice we don't have a layer 4 (session)
- Packets transmitted using the playground wire protocol are not connected together
- It's just luck (!!!!) that certain applications "appear" to work right
- And, once we turn on packet loss, there's no guarantee anymore!
- You will ***DESIGN*** and ***IMPLEMENT*** a *reliable* transport protocol for Playground.

THE UPCOMING LAB1: *RELIABLE DELIVERY*



PRFC

- Every group will create a PRFC to describe your protocol (Playground RFC)
- It needs to define the protocol clearly enough that anyone in the class can implement it!
- You will draft your PRFC as you create your protocol.
- In creating the protocol, you need to define your packet structure.

REMEMBER THIS?

TCP Header Format

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |           |U|A|P|R|S|F|                               |
   | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
   |       |           |G|K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Checksum            |         Urgent Pointer        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             data                              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                            TCP Header Format

          Note that one tick mark represents one bit position.

                               Figure 3.
```

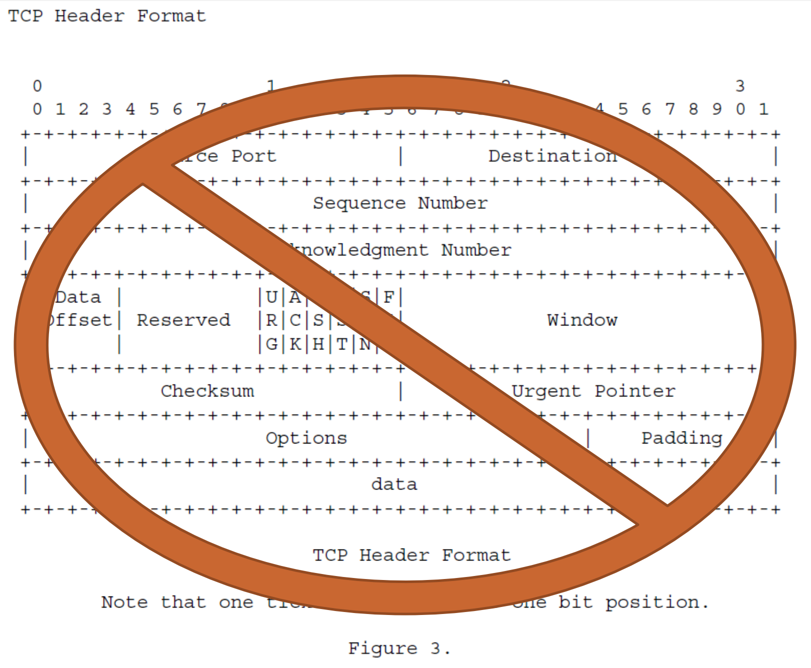
PACKET DEFINITION

- ***WE ARE NOT GOING TO DEFINE PACKETS AT THE BIT/BYTE LEVEL***
- To make this easier, Playground provides a simple method for defining, creating, and processing packets
- You start by defining a packet structure using the PacketType class

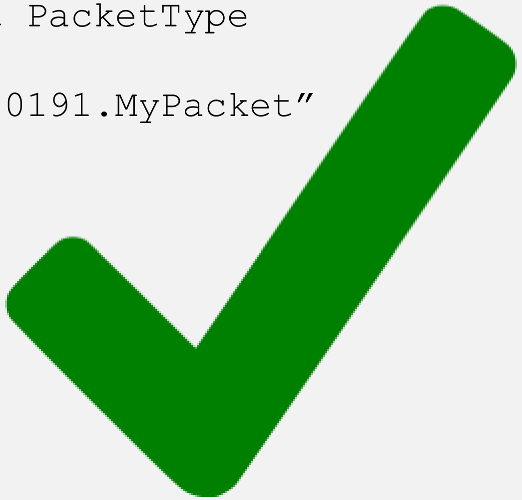
```
from playground.network.packet.fieldtypes import UINT32, STRING
from playground.network.packet import PacketType
class MyPacket(PacketType):
    DEFINITION_IDENTIFIER = “netsec.20194.MyPacket”
    DEFINITION_VERSION = “1.0”
    FIELDS = [ (“src”, STRING),
               (“port”, UINT32) ]
```

PACKETTYPE AND PRFC

- Do ***NOT*** put byte descriptions in your PRFC for your packet definitions
- Instead, put in the playground packet type class definition!




```
from playground.network.packet.fieldtypes import UINT32, STRING
from playground.network.packet import PacketType
class MyPacket(PacketType):
    DEFINITION_IDENTIFIER = “netsec.20191.MyPacket”
    DEFINITION_VERSION = “1.0”
    BODY = [ (“src”, STRING),
             (“port”, UINT32) ]
```



THE PACKET DEFINITIONS DEFINE A STANDARD

<u>But they are also practical and useful:</u>

```
packet = MyPacket()
packet.src = “20164.1.2.3”
packet.port = 80
transport.write(packet.__serialize__())
```

DESERIALIZING PACKETS

```
def __init__(self):
    self._buffer = MyPacket.Deserializer()

def recv(self, data):
    self._buffer.update(data)
    for pkt in self._buffer.nextPackets():
        print(pkt.src)
        print(pkt.port)
```