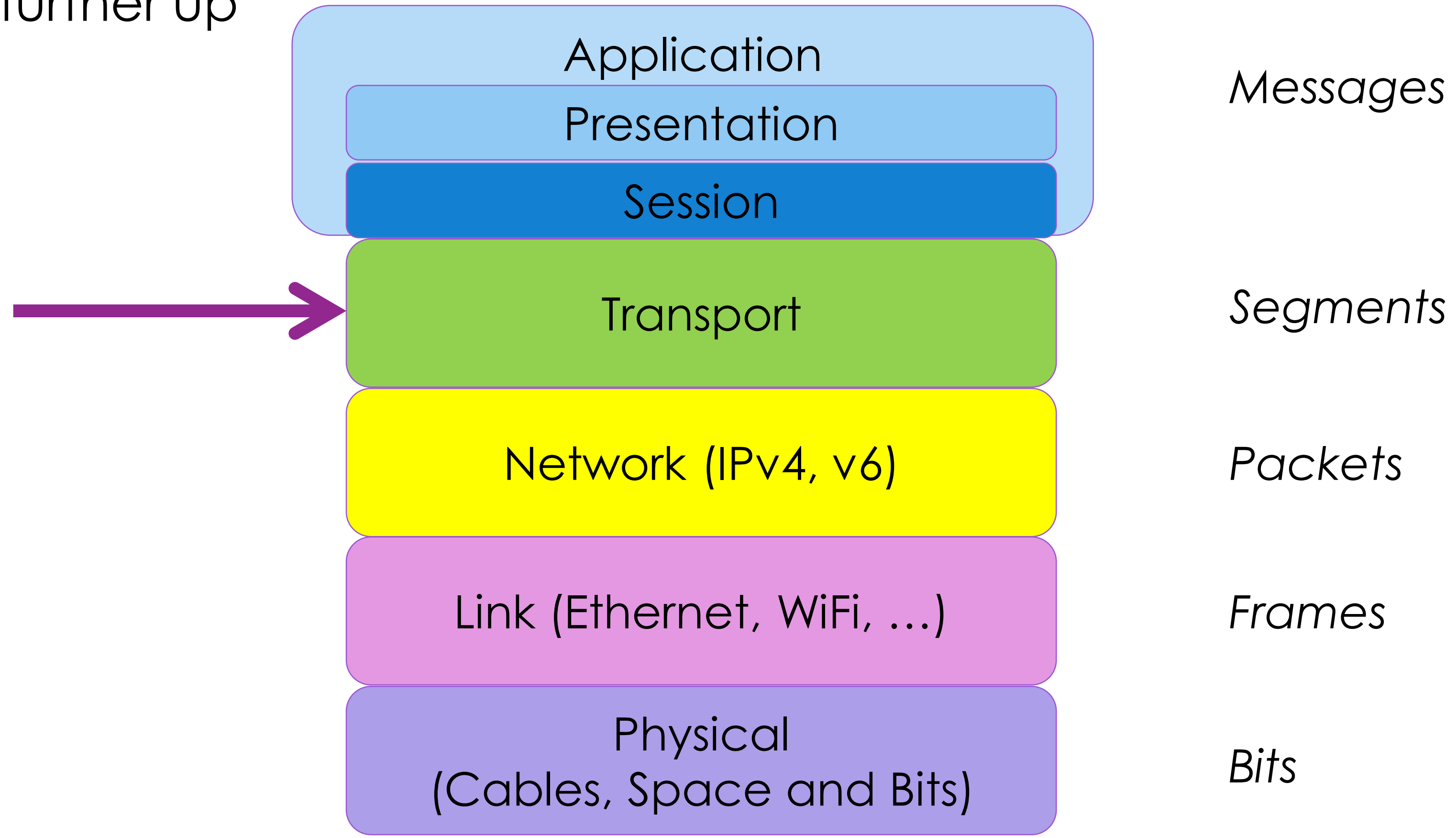
COMP3310/6331 – #12

Transport Layer: TCP/UDP intro

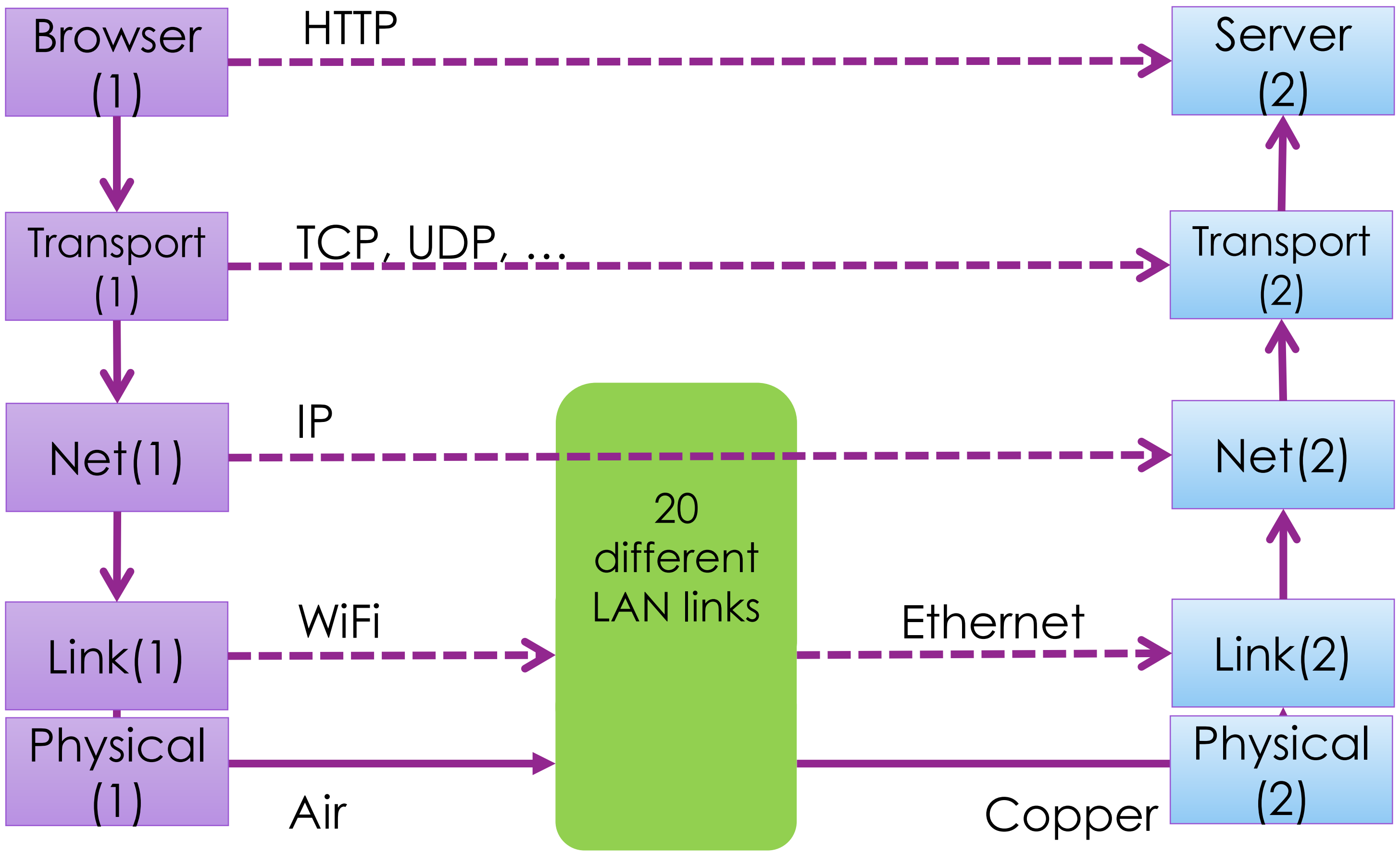
Dr Markus Buchhorn: *markus.buchhorn@anu.edu.au*

Where are we?

- Moving further up



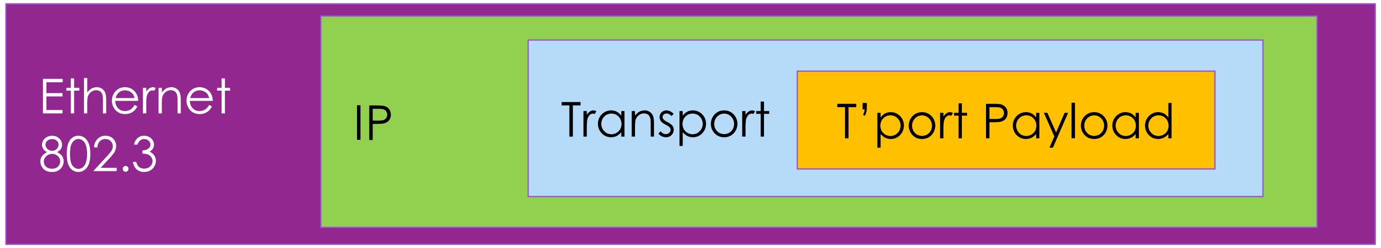
Ignore the network, focus on applications



Applications don't know nor care. Unless there is a performance question.

Getting into the transport layer

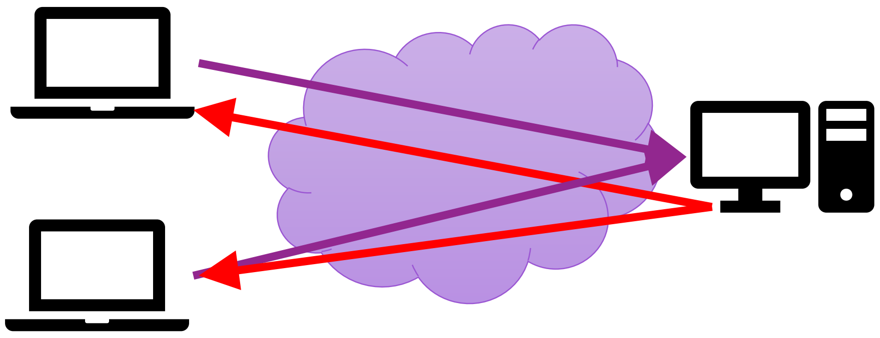
- Leave all the packet to-and-fro to the network layer
- Everything here is a payload for IP packets
  - A **Segment**



- Offers rich functionality (or not) to Applications
  - Reliability, performance, security, and other quality measures – on unreliable IP
- Routers and other network devices do not get in the way
  - They (should) only look at 'the envelope' of a message, not the messages
  - This is pure host-to-host.

Simple client/server model

- Servers offer something,
- Clients connect
  - Send a request
  - Server replies



- Servers can handle multiple clients
- Model breaks in p2p applications – everyone is both.

Transport Services

- What <u>common</u> application needs are there?
- Main decision:
  - **Reliable** - everything has to arrive bit-perfect.
    - Transport layer repairs packet loss, mis-ordering (and other damage)
    - I can wait!
  - **Unreliable**
    - Don't care about eventual perfection,
    - Do care about performance, simplicity, …
- Two types of communication
  - **Messages**: self-contained command and response (post office)
  - **Byte-stream**: generic flow of bytes, chunked into segments (conversation)

Which does what?

	Unreliable	Reliable
Messages	UDP (datagrams)	
Byte-stream		TCP (Streams)

- Could have reliable messages - but can build that on top of TCP
- Could have unreliable byte-streams - but that looks like UDP

Transmission Control Protocol: TCP = IP Protocol 6
User Datagram Protocol: UDP = IP Protocol 17

ICMP = 1, IGMP = 2, IPv6 encapsulation = 41, 130+ more

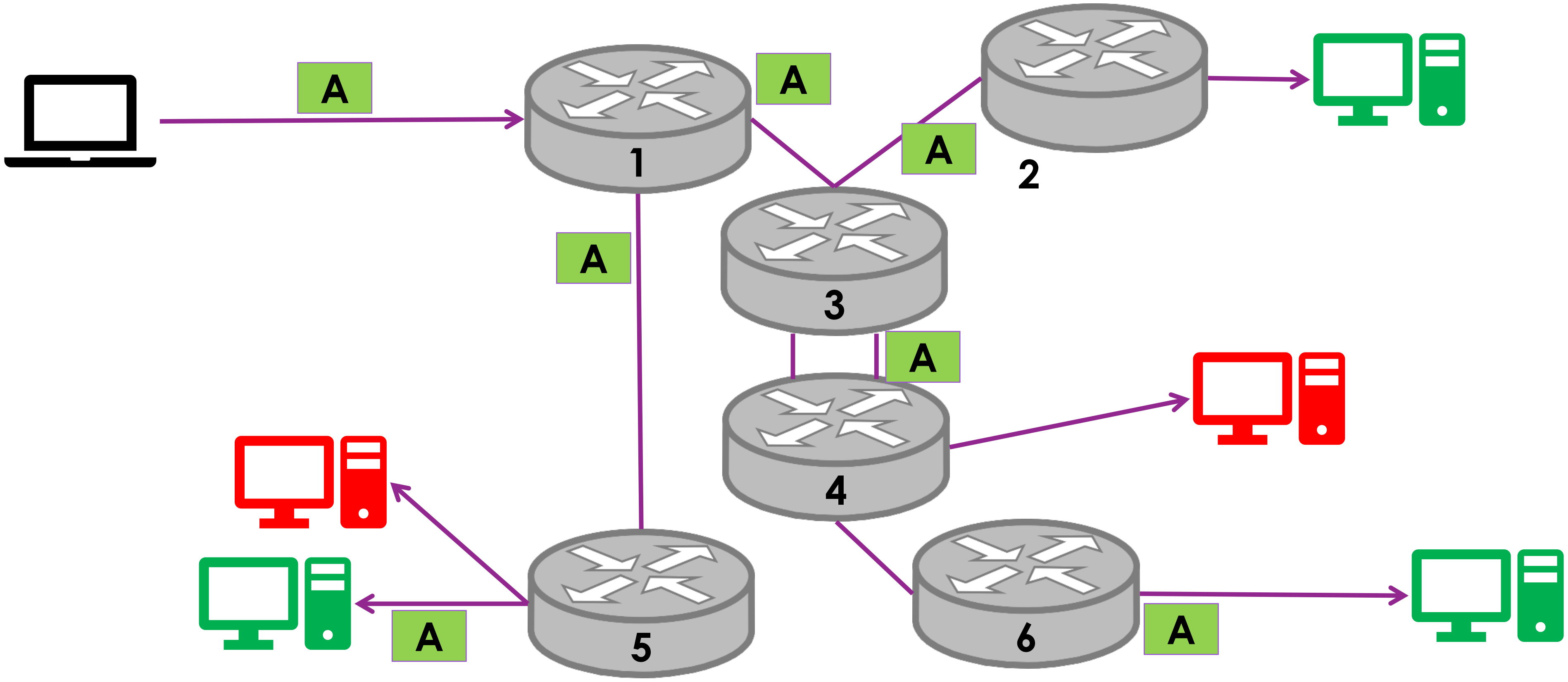
Compare them

TCP	UDP
Connection-oriented *(significant state in transport layer @host)*	Connectionless *(minimal state in transport layer)*
Delivers BYTES: once, reliably, in order *(to your process)*	Delivers MESSAGES: 0-n times, any order
Any number of bytes (in a stream)	Fixed message size
Flow control (sender/receiver negotiate)	Don't care
Congestion control (sender/network negotiate)	Don't care

- UDP is an enhanced IP packet
- TCP is a lifestyle choice – many features

IP Multicast: UDP

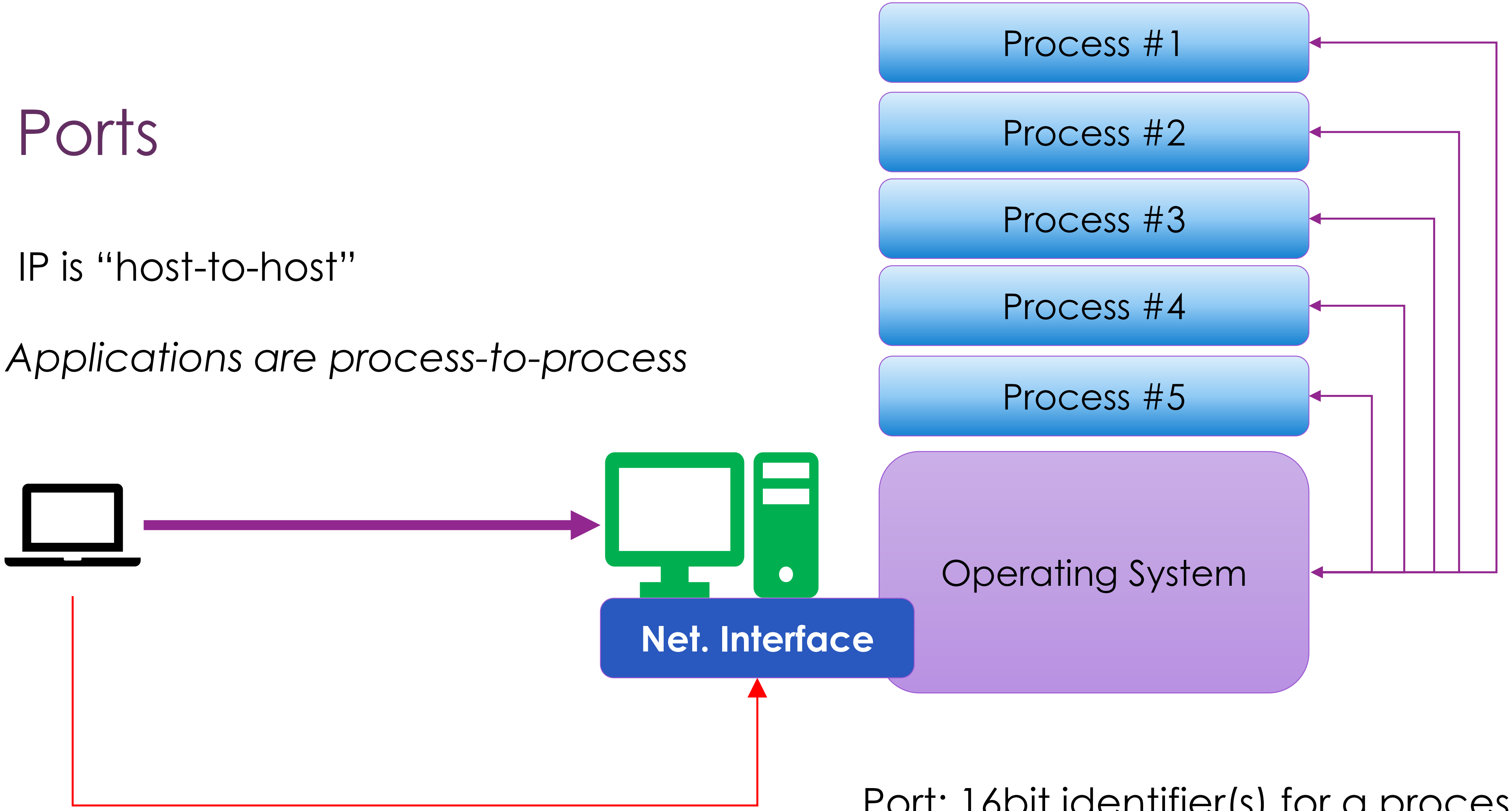
Connectionless, maybe time-sensitive
Replica packets are fine!



Ports

IP is “host-to-host”

Applications are process-to-process



Port: 16bit identifier(s) for a process,
on a host, on an interface,
at each end

Well-known (and other) ports

- https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
- Opening ports below 1024 requires extra privileges

20,21	ftp	File transfer
22	ssh	Secure shell
25	smtp	Email – outbound
80	http	Web
110	pop3	Email – inbound
143	imap	Email – inbound
443	https	Secure-Web

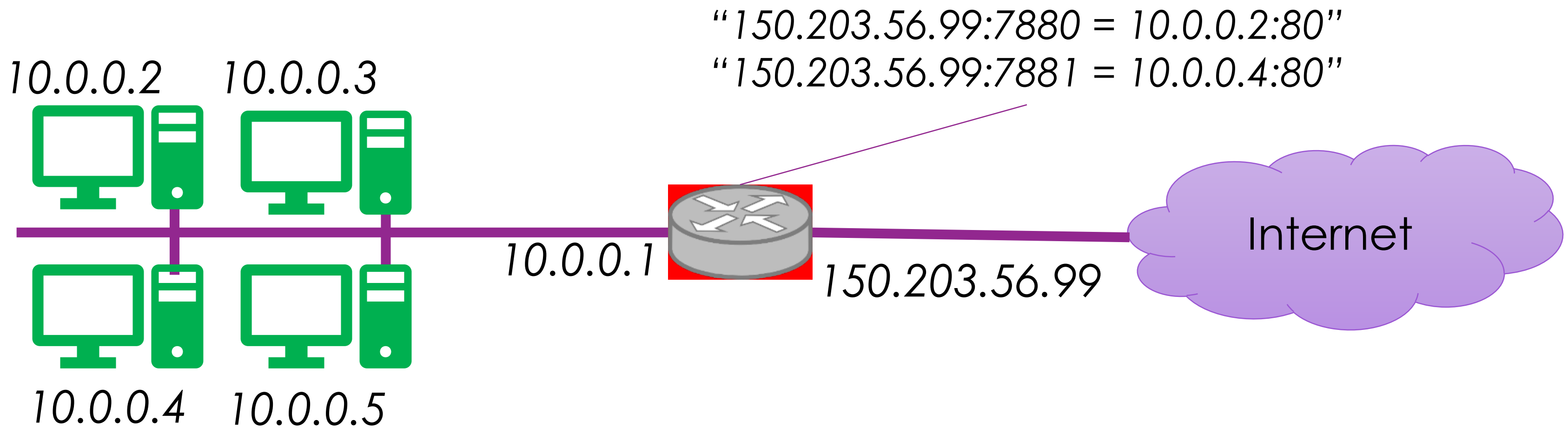
A Port is just a start

- Inetd/xinetd
  - Don’t continually run every server-service somebody may eventually talk to
  - Single service, launch appropriate service on demand
  - Listens to all (registered) ports and protocols (tcp, udp)
  - Spawns the service to have the conversation

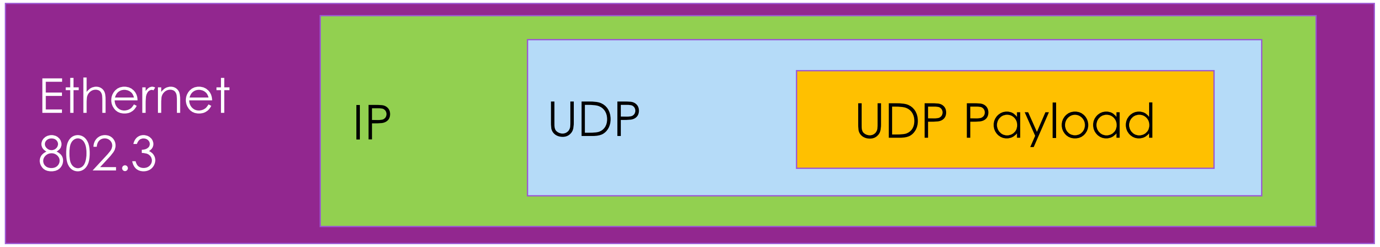
- Port mapping
  - (e.g. remote procedure calls, bittorrent, …)
  - Listen on a well-known port
  - Accept connections
  - Redirect them to a spawned service on another port
    - Services can register with the portmapper

NAT is actually NAPT

- NAT has everyone ‘hiding’ behind a single public IP address
- But everyone wants access to/from the Internet at the same time
- <u>So translate addresses **and** ports</u>
- Router maintains a table
  - Dynamically for outbound. Can be static for inbound.



UDP

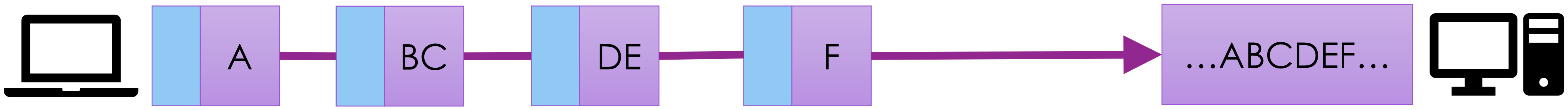


<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="16">Source Port</td><td colspan="16">Destination Port</td></tr>
<tr><td colspan="16">Length</td><td colspan="16">Checksum</td></tr>
<tr><td colspan="32">Payload (…)</td></tr>
</table>

- UDP adds to IP: Ports, payload length and a Checksum
- And nothing else…

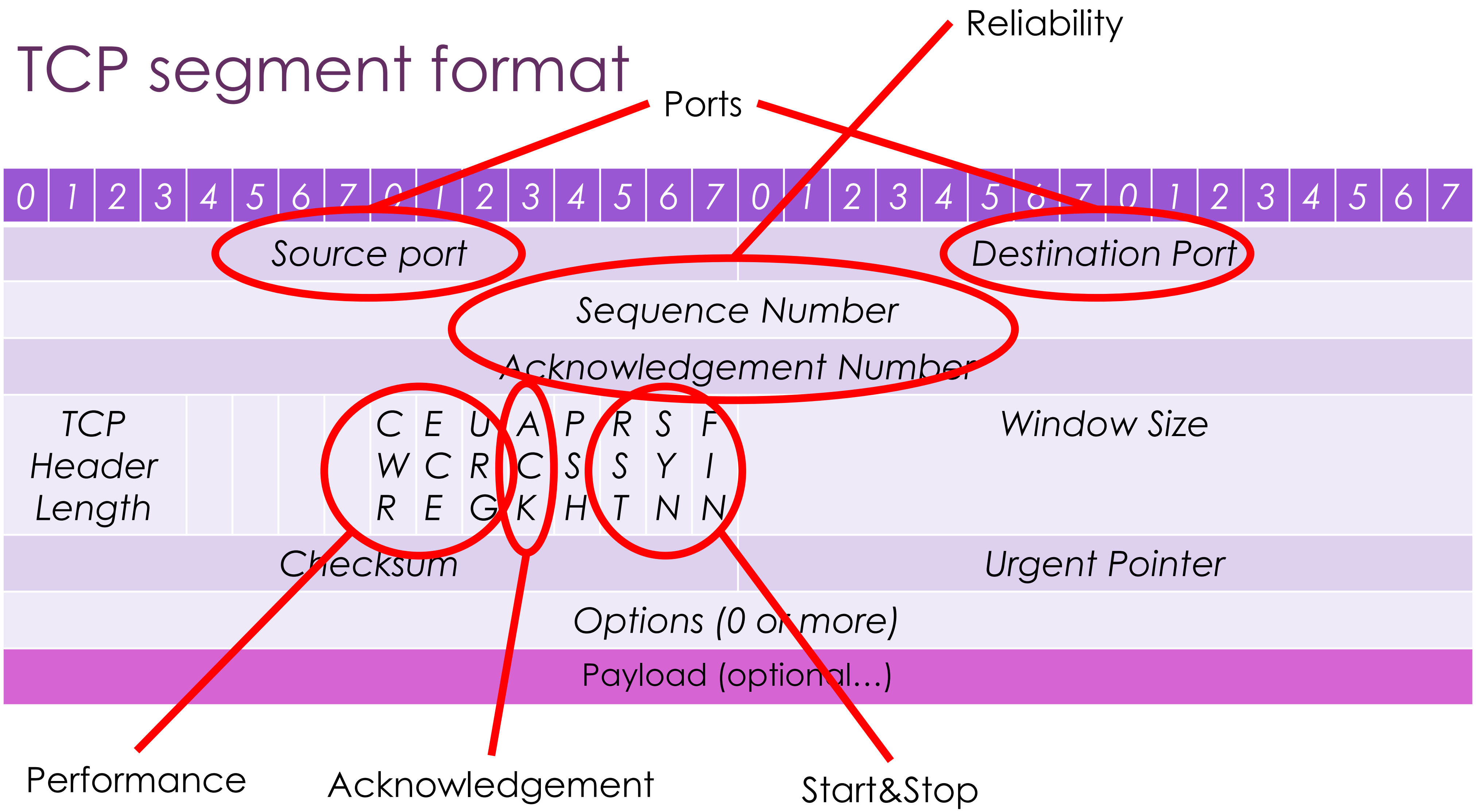
Byte-streams

- TCP segments carry chunks of a byte-stream
  - "Message" boundaries are not preserved
- Sender packetises (eventually) on *write*()
  - Multiple writes can be one packet and vice-versa – buffer dependent



- Receiver unpacks
  - Applications *read()* a stream of bytes
- Hence: <u>Segments</u>

TCP segment format

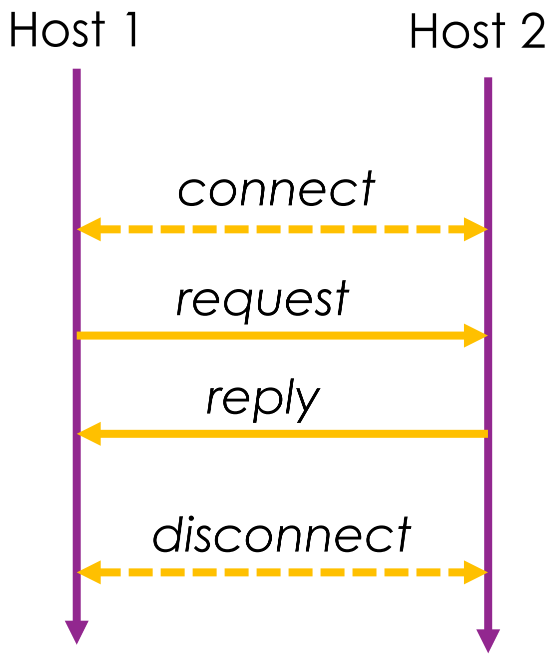


TCP Options

- These actually get used…
- Maximum Segment Size: how much each end is willing to take
- Window Scale: When 64kB is not enough – multiply
- Timestamp: For computing rtt and expanding sequence number space
- Selective Acknowledgement: Like ACK, but better.

Programming connections

- “Socket” programming – an address, a port, and a need to communicate
- *Connections are identified in the Operating System by a ‘5-tuple’*
  - *source/destination ip, source/destination port, protocol*



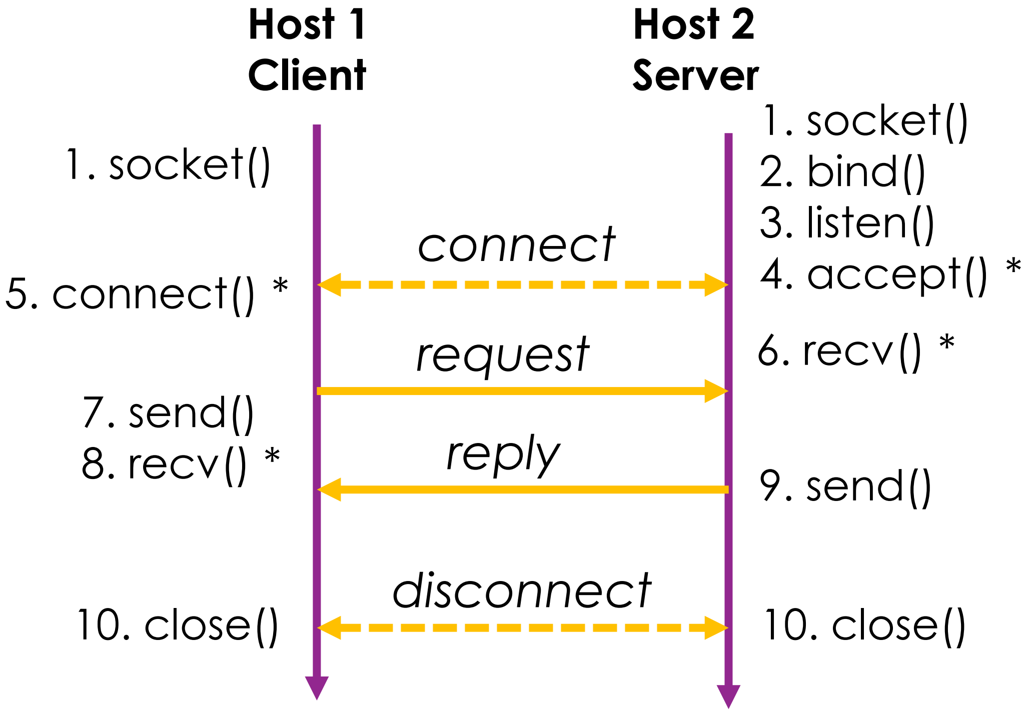
Standard conversation

Socket API

Primitive (function)	What it does
SOCKET	Create an object/descriptor
BIND	Attach a local address and port
LISTEN (tcp)	Tell network layer to get ready
ACCEPT (tcp)	Be ready!
CONNECT (tcp)	… Connect …
SEND(tcp) or SENDTO(udp)	… Send …
RECEIVE(tcp) or RECEIVEFROM(udp)	… Receive …
CLOSE	Release the connection/socket

So…

- Server needs to be prepared for connections
- Client initiates the connection



* = *call blocks*

TCP and reliability

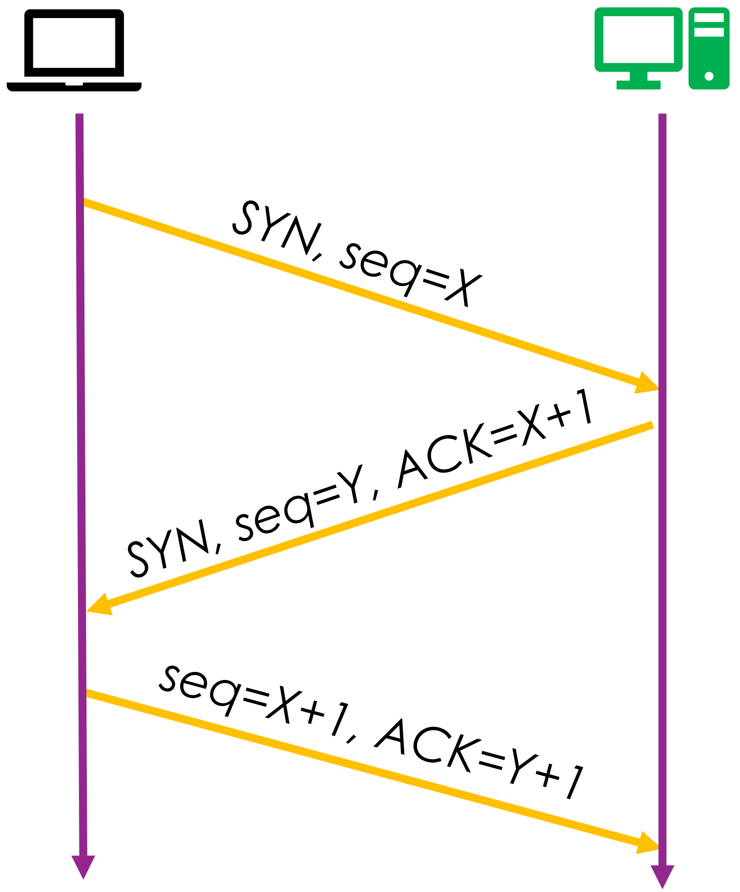
- TCP is a reliable, bidirectional byte-stream
  - Uses <u>Sequence Numbers</u> and <u>Acknowledgements</u> to provide reliability

  - Piggybacks control information on data segments in reverse direction
    - If there's no data, just sends feedback

- <u>Sequence numbers:</u> N-bit counter that wraps *(e.g. …,253, 254, 255, 0, 1, 2…)*
  - Byte count (pointer) in a stream – a **cumulative ACK**
  - Can wrap quickly on high-speed links (2^{32} = 4GB) – can use timestamps too
  - Does not start from zero (for security)
- <u>Acknowledgements:</u> Which bytes have been received/is expected

Getting connected – 3 way handshake

- TCP is full-duplex = two simplex paths
  - Both need to start together(*)
    - Synchronise Sequence numbers in both directions
- Connecting
  - Receiving transport stack decides:
    - anybody listen()ing on that port?
      - If not, ReSeT
      - If yes, passed to receiving process listen()ing,
  - Transport stack ACKnowledges
  - Originator ACKs that SYN/ACK
    - and off they go

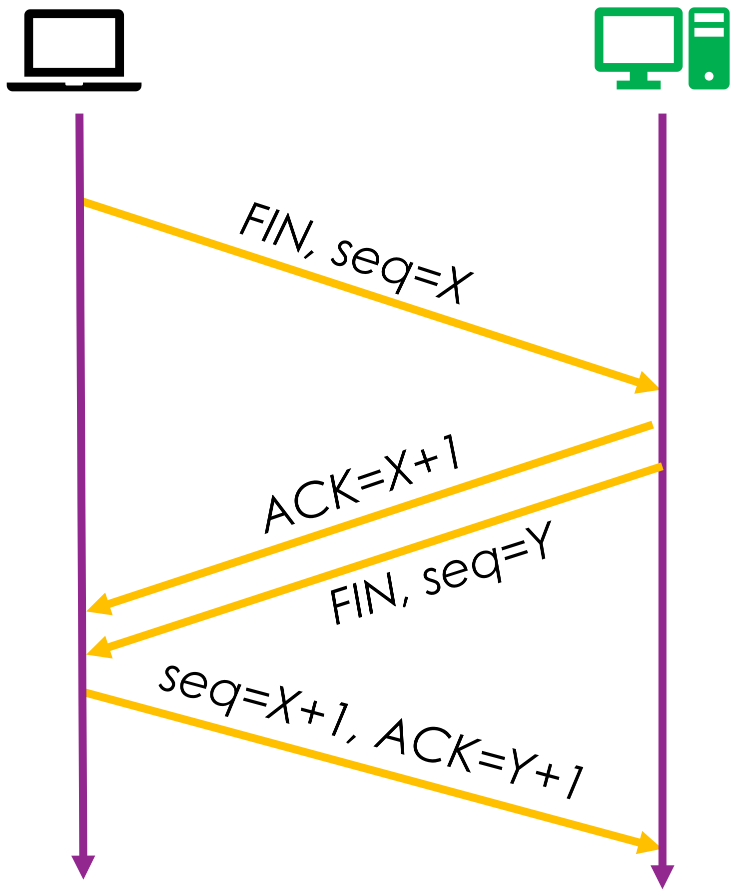


Hanging up

- Both need to end together
  - Ideally…
  - Time to flush buffers

- Disconnecting
  - One side initiates close()
  - Triggers a FIN(alise)
  - Other side ACKs and FINs too

- And if FIN is lost? Resend…



Socket states:

State	Description
LISTEN	Accepting connections
ESTABLISHED	Connection up and passing data
SYN_SENT	Waiting for reply from remote endpoint
SYN_RECV	Session requested by remote, for a listen()ing socket
LAST_ACK	Closed; remote shut down; waiting for a final ACK
CLOSE_WAIT	Remote shut down; kernel waiting for application to close() socket
TIME_WAIT	Socket is waiting after close() for any packets left on the network
CLOSED	Socket is being cleared
CLOSING	Our socket shut; remote shut; not all data has been ACK’ed
FIN_WAIT1	We sent FIN, waiting on ACK
FIN_WAIT2	We sent FIN, got ACK, waiting on their FIN

% netstat -n

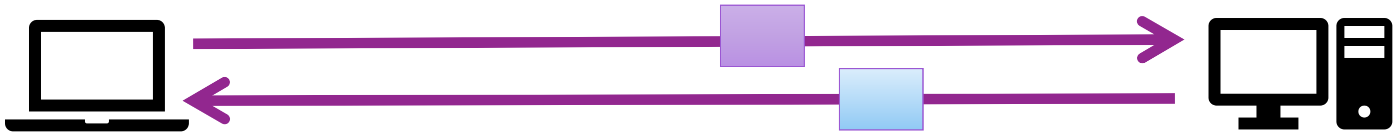
What's happening on my machine?

```
UDP    192.168.178.34:5353    *:*
UDP    192.168.178.34:52848   *:*
UDP    192.168.178.34:61842   *:*
UDP    [::]:123               *:*
UDP    [::]:3702              *:*
UDP    [::]:3702              *:*
UDP    [::]:3702              *:*
UDP    [::]:3702              *:*
UDP    [::]:3702              *:*
UDP    [::]:3702              *:*
UDP    [::]:5353              *:*
UDP    [::]:5353              *:*
UDP    [::]:5355              *:*
UDP    [::]:49533             *:*
UDP    [::]:52366             *:*
UDP    [::]:54902             *:*
UDP    [::]:61002             *:*
UDP    [::]:65404             *:*
UDP    [::1]:1900             *:*
UDP    [::1]:61841            *:*
UDP    [fe80::81c:86ce:5a66:a430%15]:546   *:*
UDP    [fe80::81c:86ce:5a66:a430%15]:1900  *:*
```

```
Proto  Local Address          Foreign Address        State
TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING
TCP    0.0.0.0:5357           0.0.0.0:0              LISTENING
TCP    0.0.0.0:7969           0.0.0.0:0              LISTENING
TCP    0.0.0.0:8501           0.0.0.0:0              LISTENING
TCP    0.0.0.0:8502           0.0.0.0:0              LISTENING
TCP    0.0.0.0:8866           0.0.0.0:0              LISTENING
TCP    0.0.0.0:8968           0.0.0.0:0              LISTENING
TCP    0.0.0.0:9330           0.0.0.0:0              LISTENING
TCP    192.168.178.34:49369   104.24.126.250:443     ESTABLISHED
TCP    192.168.178.34:49371   162.125.83.7:443       CLOSE_WAIT
TCP    192.168.178.34:49379   52.98.0.194:443        TIME_WAIT
TCP    192.168.178.34:49386   52.114.76.34:443       TIME_WAIT
TCP    192.168.178.34:49388   52.98.0.194:443        TIME_WAIT
TCP    192.168.178.34:49390   13.107.3.128:443       TIME_WAIT
TCP    192.168.178.34:49393   162.125.83.3:443       CLOSE_WAIT
TCP    192.168.178.34:49396   52.5.51.106:443        ESTABLISHED
TCP    192.168.178.34:49398   192.168.178.1:49000    TIME_WAIT
TCP    192.168.178.34:49404   40.100.146.18:443      ESTABLISHED
TCP    192.168.178.34:49405   52.98.5.226:443        ESTABLISHED
TCP    192.168.178.34:49409   104.98.4.162:80        ESTABLISHED
TCP    192.168.178.34:49411   54.154.198.3:443       ESTABLISHED
TCP    192.168.178.34:49412   162.125.34.137:443     ESTABLISHED
TCP    192.168.178.34:50471   52.230.7.59:443        ESTABLISHED
TCP    192.168.178.34:53421   104.154.164.197:443    ESTABLISHED
TCP    192.168.178.34:53451   74.125.68.125:5222     ESTABLISHED
TCP    192.168.178.34:53568   52.98.0.34:443         ESTABLISHED
TCP    192.168.178.34:53591   34.226.253.48:443      ESTABLISHED
TCP    192.168.178.34:54258   192.168.178.44:445     ESTABLISHED
TCP    192.168.178.34:54451   54.84.185.96:443       ESTABLISHED
TCP    192.168.178.34:56177   35.229.34.229:443      ESTABLISHED
TCP    192.168.178.34:61528   40.100.146.18:443      ESTABLISHED
TCP    192.168.178.34:62647   13.112.202.196:443     ESTABLISHED
TCP    192.168.178.34:64245   216.58.200.106:443     CLOSE_WAIT
TCP    192.168.178.34:64328   172.217.167.74:443     CLOSE_WAIT
TCP    192.168.178.34:65180   162.125.34.129:443     ESTABLISHED
TCP    192.168.178.34:65439   52.98.0.194:443        TIME_WAIT
TCP    192.168.178.34:65496   162.125.34.129:443     ESTABLISHED
TCP    192.168.178.34:65503   52.98.0.146:443        ESTABLISHED
TCP    192.168.178.34:65511   52.98.0.178:443        ESTABLISHED
```

TCP Sliding Windows

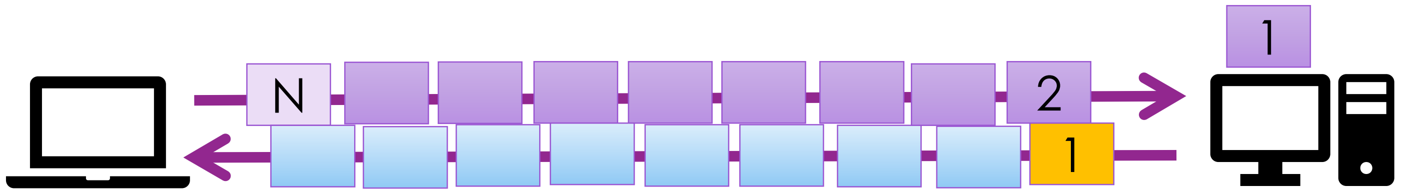
- Want reliability **and** throughput (of course!)
- Start with ARQ – stop-and-wait
  - Single segment outstanding = problem on high bandwidth*delay networks



- Say one-way-delay=50ms so round-trip-time (RTT)=2d=100ms
- Single segment per RTT = 10 packets/s
  - Typical packet ? Say 1000 bytes = ~10,000 bits -> 100kb/s
- Even if bandwidth goes up, throughput doesn't!

TCP Sliding Windows

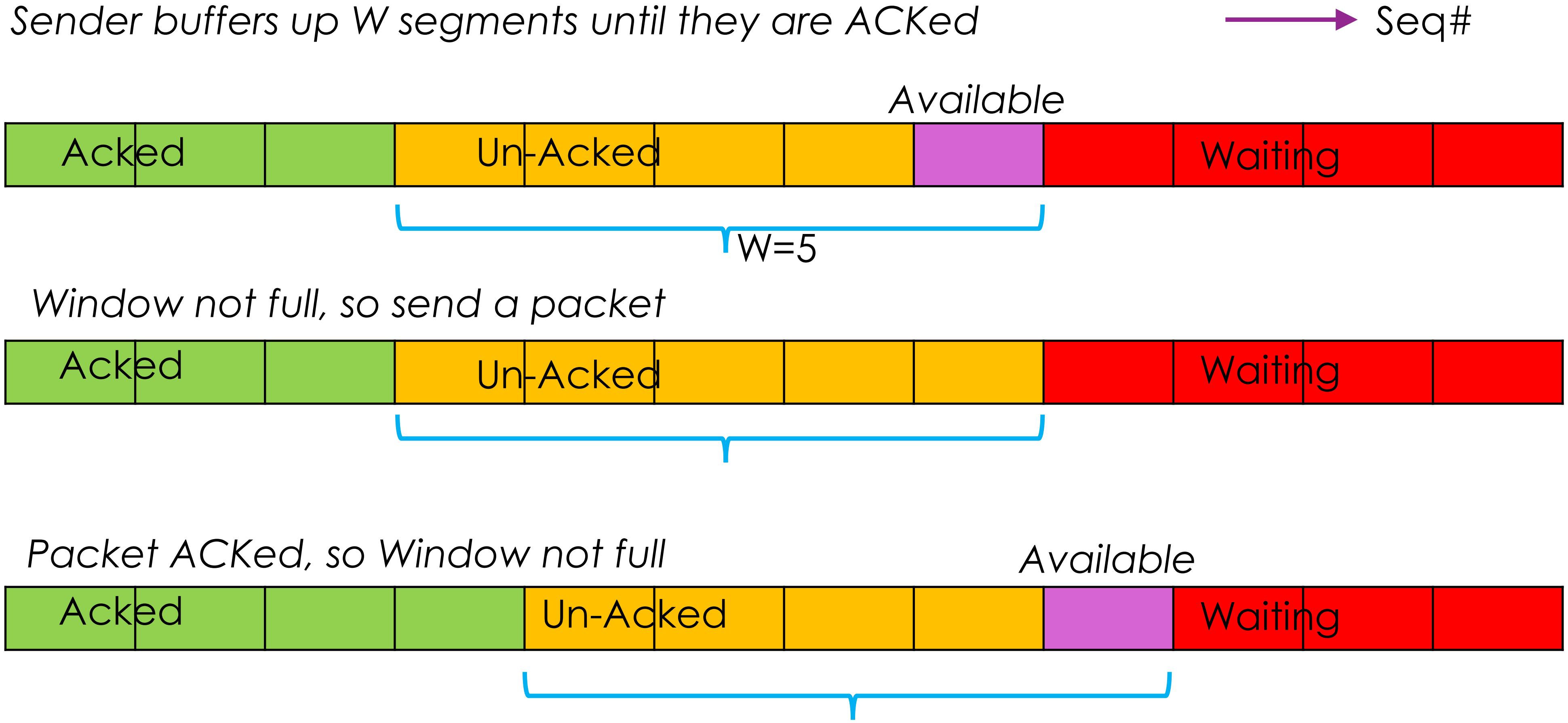
- Allow W segments to be ‘outstanding’ (unACKed) per RTT
  - Fill a pipeline/conveyor-belt with segments



- Set up a ‘window’ of W segments
- W=2*Bandwidth*delay
- At 100Mb/s, delay=50ms means <u>W=10Mb</u>
  - Assuming same 10kb segments, W=1000 segments
  - 500 are out there somewhere!

Sliding Window approach

Sender buffers up W segments until they are ACKed



If(lost) then: ARQ – "Go Back N"

- **Receiver** buffers just a single segment
- If it's the next one in sequence, ACK it, everyone happy
- If it's not, drop it,
- Let sender retransmit what I'm actually waiting for

1	2	3	4	5

- **Sender** has a single timer. After timeout, resend (all) from (first) ACK-less.
- Really simple, but somewhat inefficient

ARQ – "Selective Repeat"

- **Receiver** buffers many segments
  - Reduce retransmissions
- ACK what has been received in order
- And also ACK received segments that aren't
  - Any gaps indicates missing segment!
  - *Selective ACK (SACK)*
  - TCP header has an ACK flag (1bit), and a SACK Option (32bits…)
  - **3 duplicate ACKs (plus SACKs) trigger resend**

- **Sender** has a timer per unACKed-segment
  - As each timer expires, resend that segment
- Cope with (some) misordering. Way more efficient, now widespread

Everybody runs the same TCP…?

- No. There is no single TCP stack

- Many years of various optimisations, experiments, algorithms, …
  - Suited to various circumstances
  - And as vulnerabilities have been found and mitigated (and found and …)

- Doesn’t impact the network, only hosts, so you can do what you want…