CS 162: Operating Systems and Systems Programming

Lecture 10: Sockets & Networking

July 10, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu

Logistics

- Project 1 Milestone Due Tonight
- Homework 1 Due Friday
- Project/HW "Party" Friday 3-5PM
- Midterm Exam: 7/18, 5-7pm
  - 155 Dwinelle
  - No Lecture that Day

Recall: Device Driver

- Software module that interacts directly with hardware (issues commands)
- Provides a *standard interface* to the OS, same kernel I/O system can interact with different devices
- Two parts:
  1. "Top Half" accessed from system calls
     - **`open`**, **`read`**, **`write`**, **`ioctl`**, …
     - Starts I/O
  - "Bottom Half" invoked from interrupt handler
     - Gets input or transfers output when device is ready
     - Responsible for waking blocked threads when I/O is complete

Recall: Life Cycle of An I/O Request

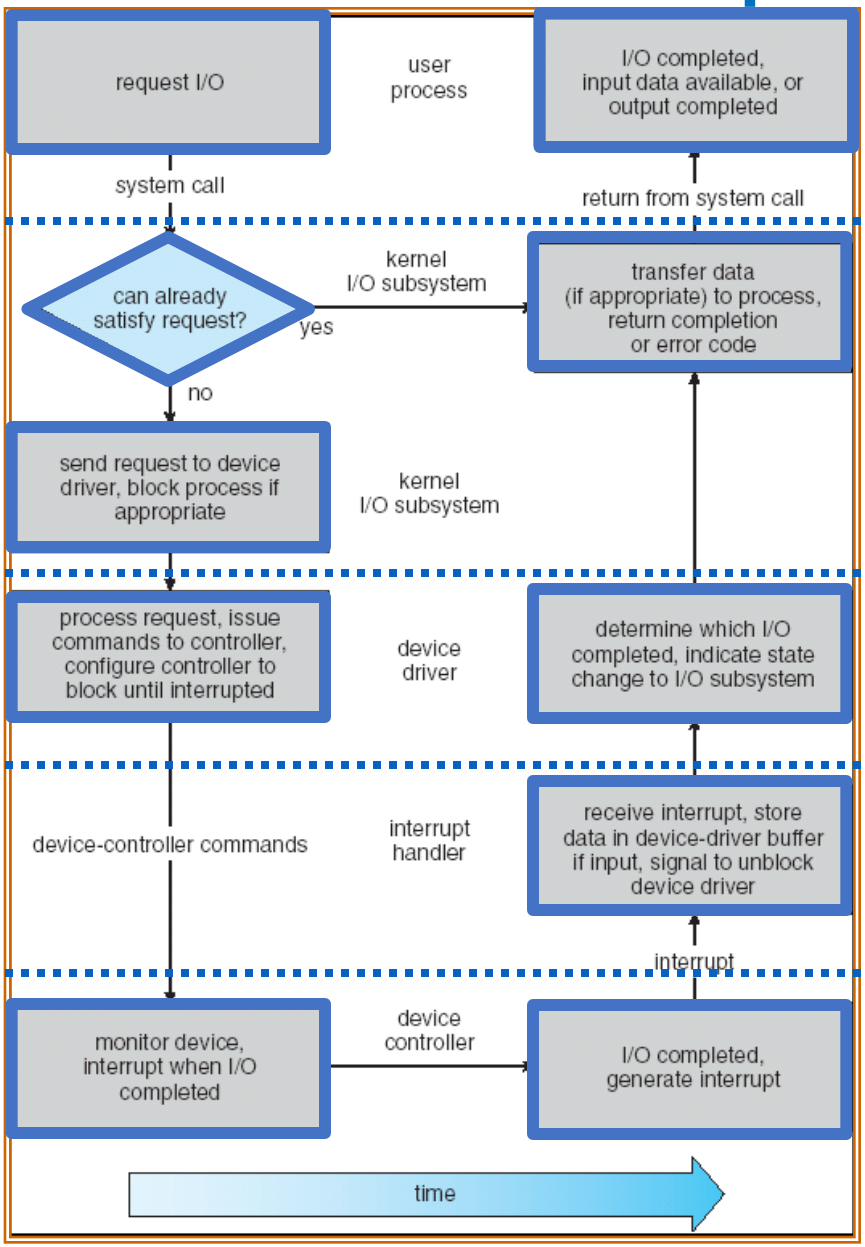
User Program

Kernel I/O Subsystem

Device Driver Top Half

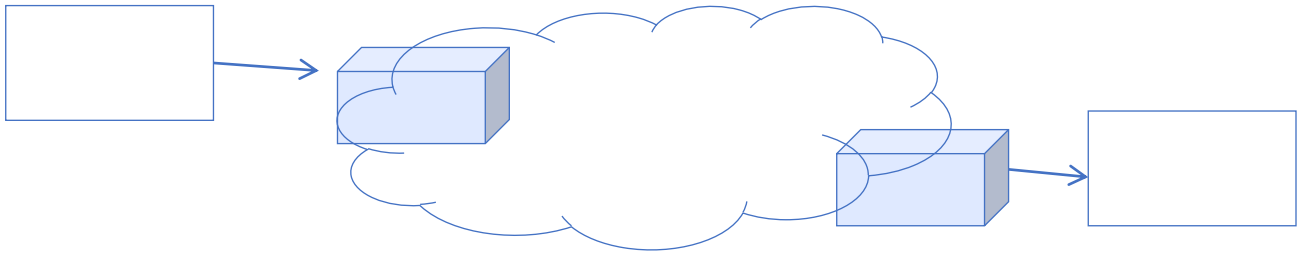
Device Driver Bottom Half

Device Hardware



Recall: Communication Across the World Looks Like File IO

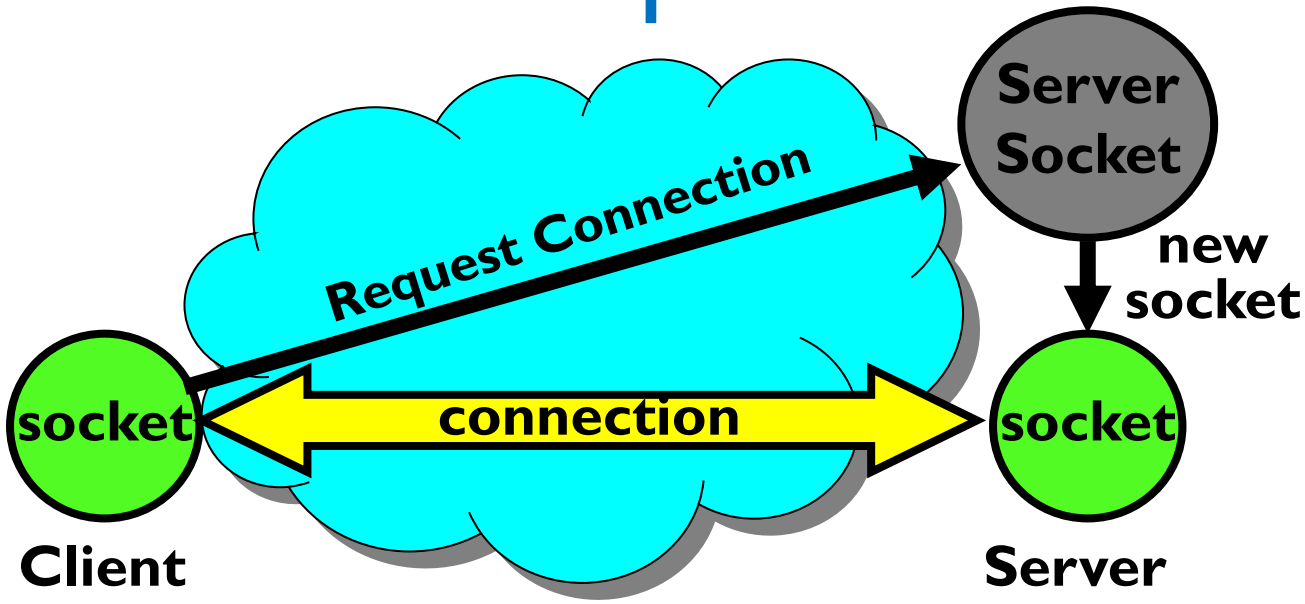
```
write(wfd, wbuf, wlen);
```



```
n = read(rfd,rbuf,rmax);
```

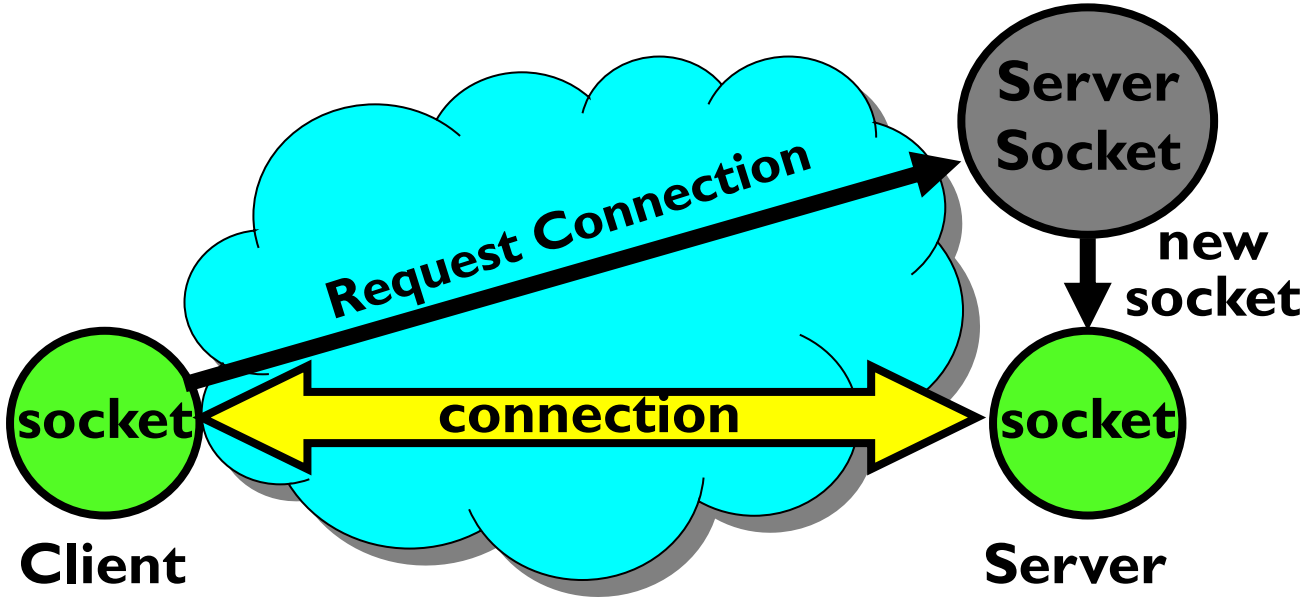
- Sockets: Connected queues over the Internet
  - How to **`open()`**? Filenames?
  - How are they connected in time?

Recall: Socket Setup over TCP/IP



- Special kind of socket: **server socket**
  - Has file descriptor
  - Can't read or write
- Two operations:
  1. `listen()`: Start allowing clients to connect
  2. `accept()`: Create a *new socket* for a *particular* client

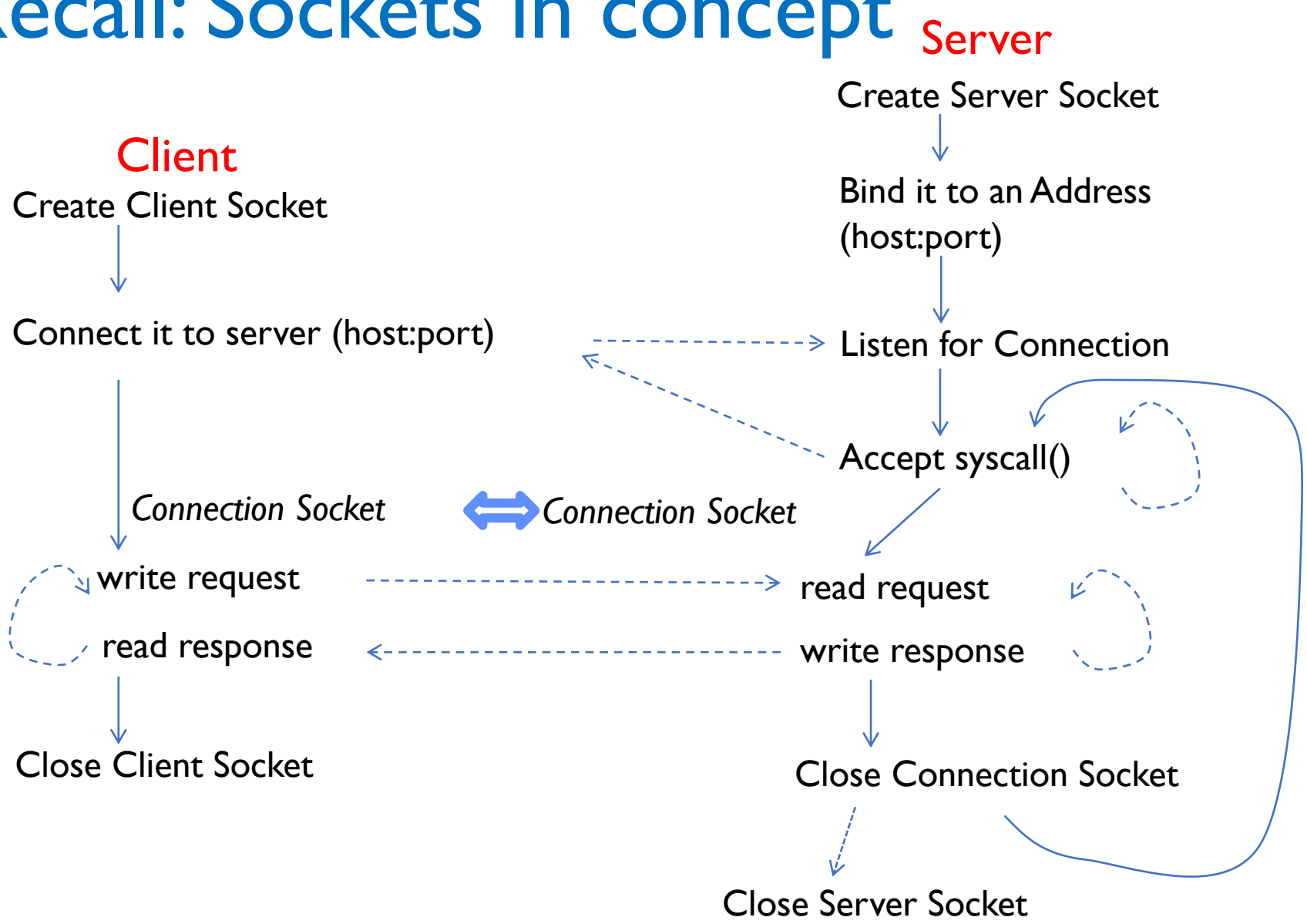
Recall: Socket Setup over TCP/IP



- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

- Server port is "well known"
- Where does client get its port number from?
- Recall: dynamic/private port range

Recall: Sockets in concept



Recall: Client Protocol

```
char *hostname; char* portname;
int sockfd;
struct addrinfo *server;
struct hostent *server;
server = buildServerAddr(hostname, portname);

// Create a TCP socket
// server->ai_family: AF_INET (IPv4) or AF_INET6 (IPv6)
// server->ai_socktype: SOCK_STREAm (byte-oriented)
// server->ai_protocol: IPPROTO_TCP
sockfd = socket(server->ai_family, server->ai_socktype,
                server->ai_protocol)

// Connect to server on port
connect(sockfd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
client(sockfd);

/* Clean up on termination */
close(sockfd);
freeaddrinfo(server);
```

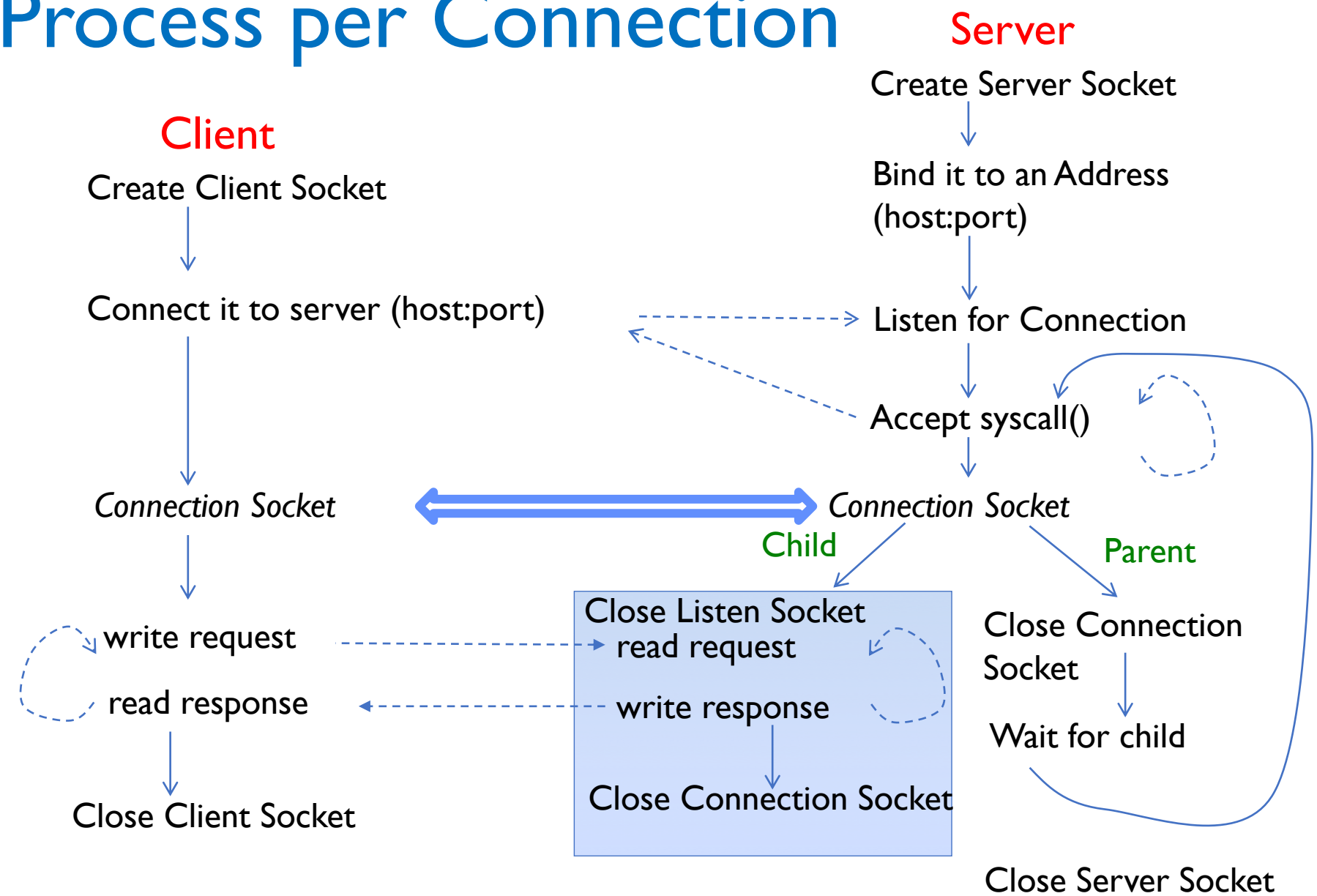
Recall: Server Protocol (Version 1)

```
/* Create Socket to receive requests */
lstnsockfd = socket(server->ai_family, server->ai_socktype,
                    server->ai_protocol);

/* Bind socket to port */
bind(lstnsockfd, server->ai_addr, server->ai_addrlen);
/* Listen for incoming connections */
listen(lstnsockfd, MAXQUEUE);
while (1) {
   /* Accept incoming connection, obtaining a new socket for it */
   consockfd = accept(lstnsockfd, NULL, NULL);

   server(consockfd);

   close(consockfd);
  }
close(lstnsockfd);
```

Handling Multiple Connections

- One Option: Fork a process for each connection
  - Strong isolation between each connection
  - Can accept new connections while others are active
  - ***Very Expensive***
- Second option: Spawn a thread for each connection
  - Or better yet, use a thread pool
- Third option: Event style

Process per Connection



Server Protocol (Version 2)

```
listen(lstnsockfd, MAXQUEUE);
while (1) {
    consockfd = accept(lstnsockfd, NULL, NULL);
    cpid = fork();                /* new process for connection */
    if (cpid > 0) {               /* parent process */
      close(consockfd);
      tcpid = wait(&cstatus);
    } else if (cpid == 0) {        /* child process */
      close(lstnsockfd);          /* let go of listen socket */

      server(consockfd);

      close(consockfd);
      exit(EXIT_SUCCESS);           /* exit child normally */
    }
  }
close(lstnsockfd);
```

Client: Getting the Server Address

```
struct addrinfo *buildServerAddr(char *hostname, char *portname) {
  struct addrinfo *result;
  struct addrinfo hints;
  int rv;
  memset(&hints, 0, sizeof(hints));  // Clear unused hints
  hints.ai_family = AF_UNSPEC;       // IPv4 or IPv6
  hints.ai_socktype = SOCK_STREAM;   // TCP, byte-oriented

  rv = getaddrinfo(hostname, portname, &hints, &result);
  if (rv != 0) {
    // Handle error
  }
  return result;
}
// Later: freeadrinfo(result)
```

Server Address: with getaddrinfo

```
struct addrinfo* server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));        // Clear unused hints
hints.ai_family = AF_UNSPEC;             // IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;         // TCP, byte-oriented
hints.ai_flags = AI_PASSIVE;             // For listening

// NULL for hostname argument
rv = getaddrinfo(NULL, portname, &hints, &server);
if (rv != 0) {
  // Handle error
}

// Later: freeaddrinfo(result)
```

Server Address: Manually (IPv4)

```
int port number = <port number>;
struct addrinfo server;
struct sockaddr_in server_ip_port;

server_ip_port.sin_family = AF_INET;     // IPv4
// Bind to all addresses available on machine
server_ip_port.sin_addr.s_addr = INADDR_ANY;
// htons: Host to network (Big Endian) byte order for short type
server_ip_port.sin_port = htons(port_number);

server.ai_addr = (struct sockaddr*) &server_ip_port;
server.ai_addrlen = sizeof(struct sockaddr_in);
server.ai_family = AF_INET;
server.ai_socktype = SOCK_STREAM;
server.ai_protocol = IPPROTO_TCP; // or 0, means “choose any”
```

Networking Definitions

- **Network**: Physical connection that allows two computers to communicate
- **Frame/Packet/Segment:** Unit of data transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when frame arrives
  - Name depends on which layer (later)

What is a Protocol?

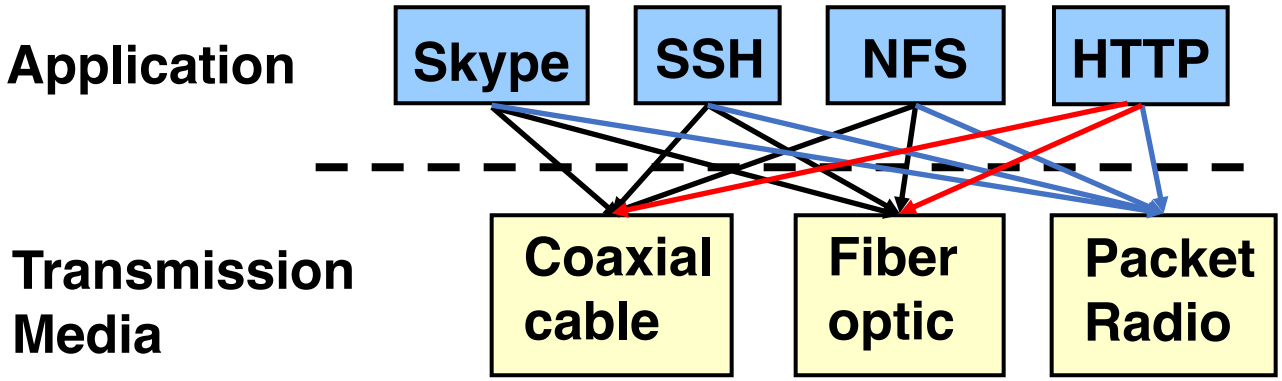
- A protocol is **an agreement on how to communicate**
  - **Syntax:** Format, order messages are sent and received
  - **Semantics:** Meaning of each message

- Often described by a state machine

Networking Challenge

- Many different applications
  - Email, Web, Online Games, etc.
- Many different network styles and technologies
  - Wireless, Wired, Optical, etc.
- How do we organize all of this complexity?

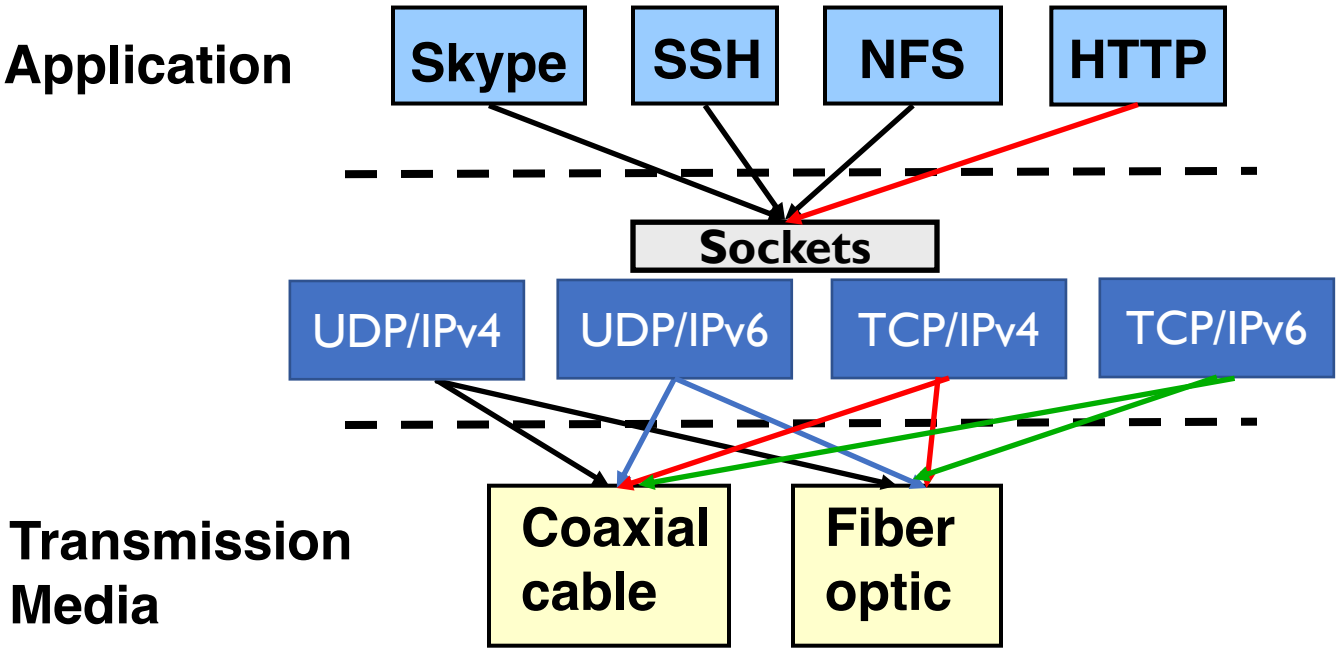
Networking Challenge



- Re-implement every application for every technology?

- No

Networking Challenge



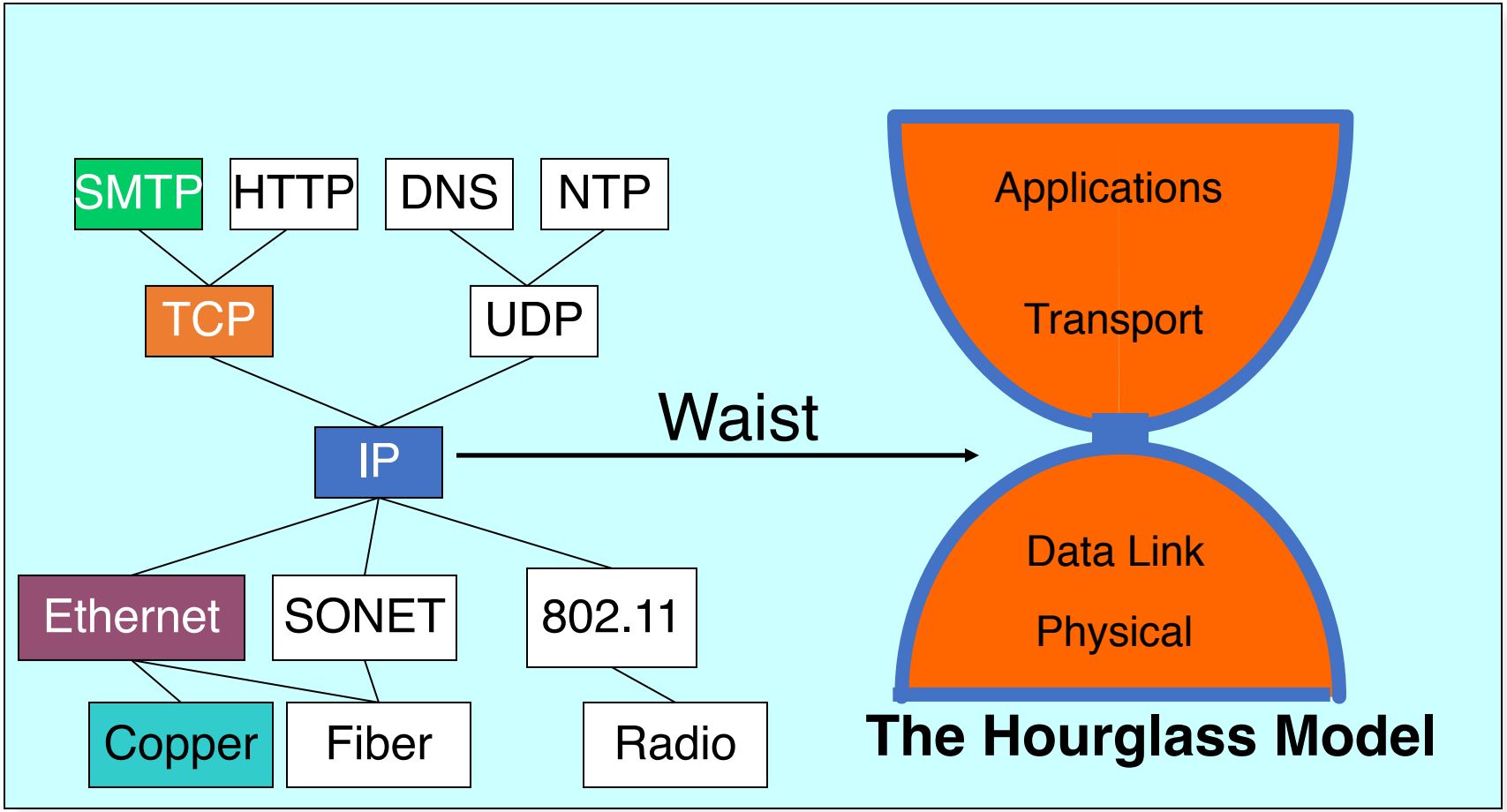
- Re-implement sockets for every technology?

- No

Layering

- Complex services from simpler ones
  1. Physical and Link Layers (Wi-Fi, Ethernet, …)
     - Unreliable, local exchange of limited-size **frames**
  2. Network (IP) – routing between local networks
     - Unreliable, global exchange of limited-size **packets**
  3. Transport (e.g., TCP) – Glue
     - Reliable (with retries), ordering, stream of bytes
  4. Application – Everything on top of sockets

The Internet *Hourglass*



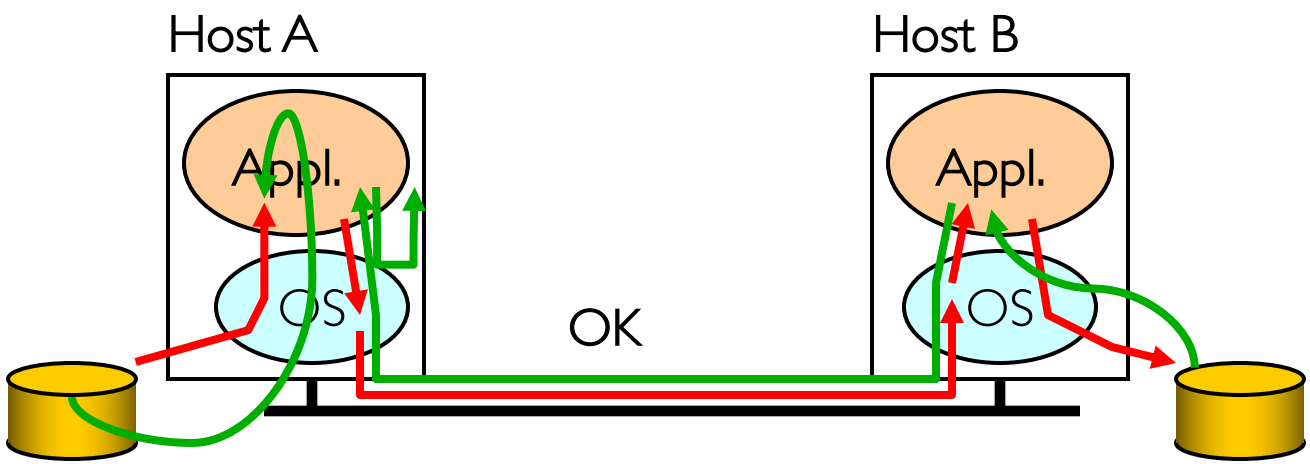
Implications of Hourglass

- There is only **one** Network-Layer Protocol: **IP**
- Allows networks to interoperate
- Above IP: Applications function on all networks
- Below IP: Change network's construction without disturbing applications
- One drawback: Changing IP itself (e.g. transitioning to IPv6) very involved

End-to-End Principal

- Seen as a guiding principle of the Internet
- Some types of network functionality can only be correctly implemented *end-to-end*
  - Reliability, security, etc.
- Implementing complex functionality in the network:
  - Doesn't necessarily reduce complexity on end hosts
  - Does increase network complexity
  - Imposes a cost on all applications, *even if they don't need the functionality*

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them

- Solution 2: end-to-end **check** and try again if necessary

Summary: Network Layers

- Link Layer (local network)
  - Send *frames* addressed to neighboring machines
  - Ethernet, Wi-Fi
- Network layer (connecting local networks)
  - Forwarding between local networks
  - Send packets addressed to machines anywhere
  - IP
- Transport Layer (making streams)
  - Turn sequence of packets into reliable byte stream
  - TCP

Glue: Adding Functionality

Physical Reality: Frames	Abstraction: Stream
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-Machine	Process-to-Process
Only on Local Area Net	Routed Anywhere
Asynchronous	Synchronous

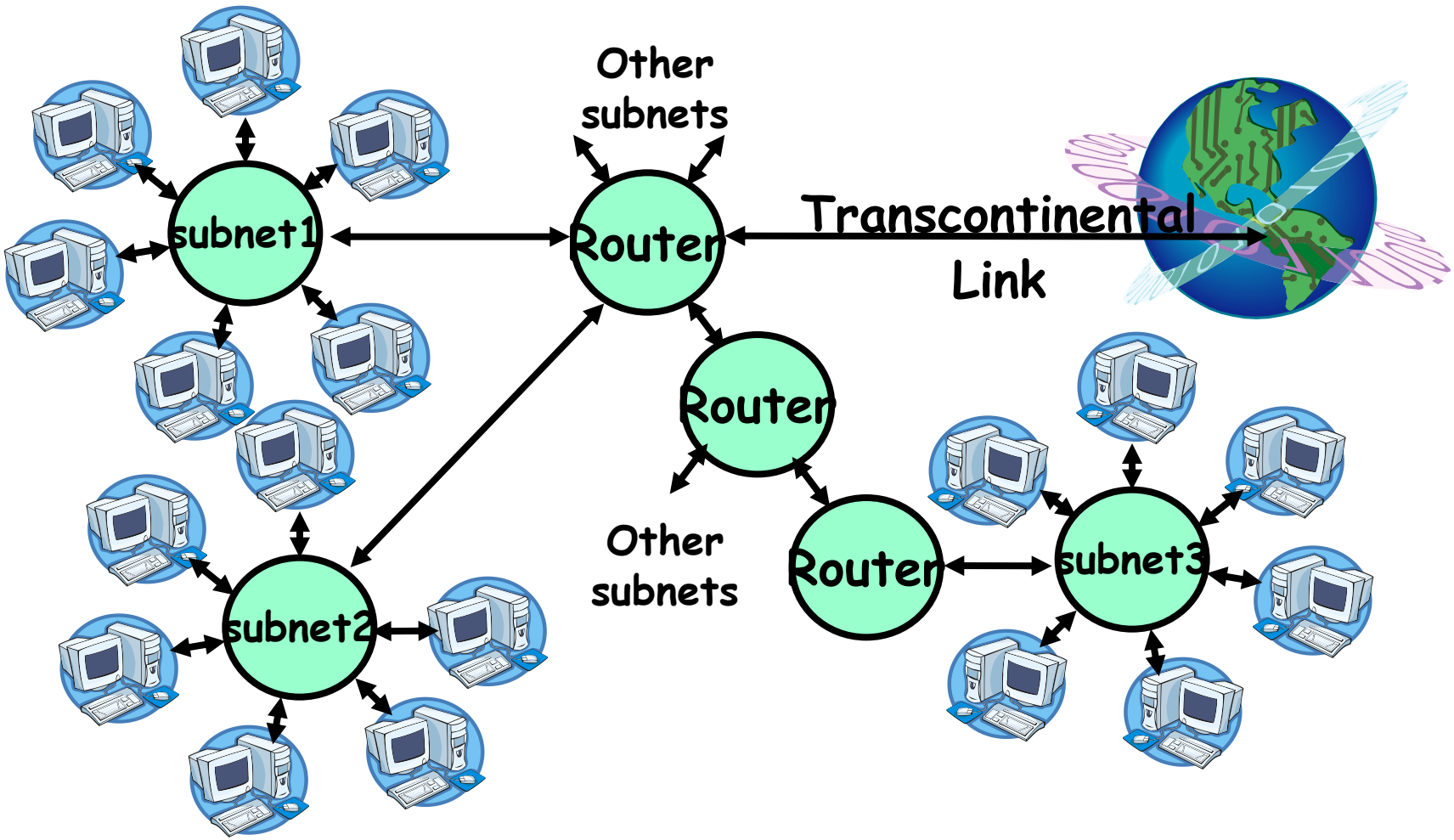
Glue: Adding Functionality

Network Layer: IP

Physical Reality: Frames	Abstraction: Stream
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-Machine	Process-to-Process
Only on Local Area Net	**Routed Anywhere**
Asynchronous	Synchronous

Internet: Network of Networks

Hierarchy of Networks: Scales to millions of hosts



Internet Protocol Features

- Routing – an IP packet goes anywhere
  - Just need the destination IP address
- Fragmentation – split big messages into smaller pieces
  - Think about downloading a file
  - Maximum size 64K
  - Reassemble at destination
  - Hides differences in physical layers
- Multiple protocols running on top
  - ICMP, TCP, UDP, …

Internet Protocol "Non-Features"

- Unreliable Delivery ("Best Effort")
  - IP packet delivery not guaranteed
  - May be lost by underlying physical layer (e.g., radio noise)
  - May be dropped in transit
- Out-of-order/duplicate delivery
  - Tolerance of physical layer retrying transmission
  - Tolerance of multiple paths

Glue: Adding Functionality

Transport Layer: TCP

Physical Reality: Frames	Abstraction: Stream
Limited Size	**Arbitrary Size**
Unordered (sometimes)	**Ordered**
Unreliable	**Reliable**
Machine-to-Machine	**Process-to-Process**
Only on Local Area Net	Routed Anywhere
Asynchronous	Synchronous

Ordered Messages: Problem

- Want to divide a message into packets
- Think about downloading a file over IP
  - 64K max packet size
- IP might reorder these packets
  - Imagine receiving the end of a file before the beginning!

Ordered Messages: Solution

- Simulate ordered messages on top of unordered messages

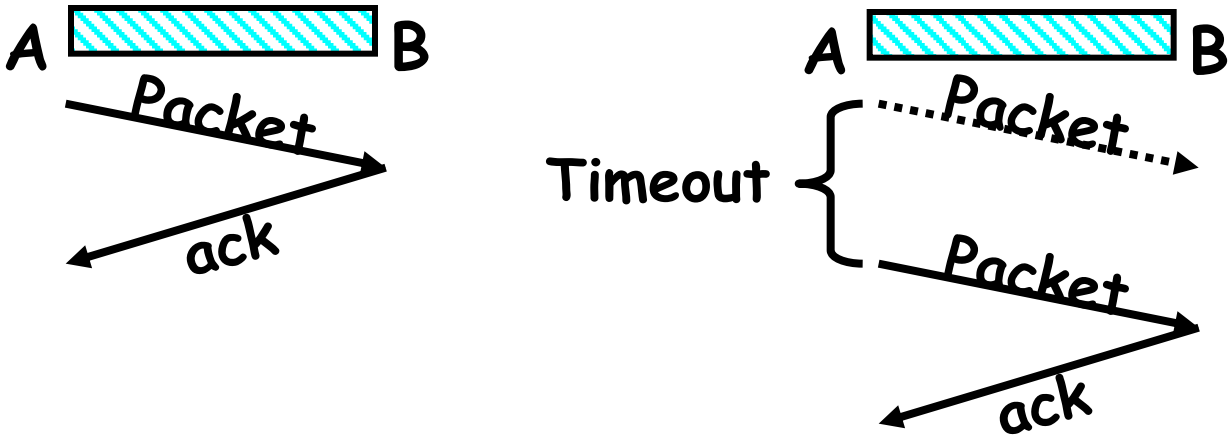
- Assign each packet a sequence number: 0, 1, 2, 3, …
  - If packets arrive out of order, hold on to them
  - Deliver them *in order* to user (through socket interface)

- Example: Hold on to #3 until #2 arrives, etc.

Reliable Message Delivery: Problem

- All physical networks can garble or drop packets
  - Physical hardware problems (bad wire, bad signal)
- Therefore, IP can garble or drop packets
  - It doesn't repair this itself (end-to-end principle!)
- Building reliable message delivery
  - Confirm that packets aren't garbled
  - Confirm that packets arrive **exactly once**

Using Acknowledgements

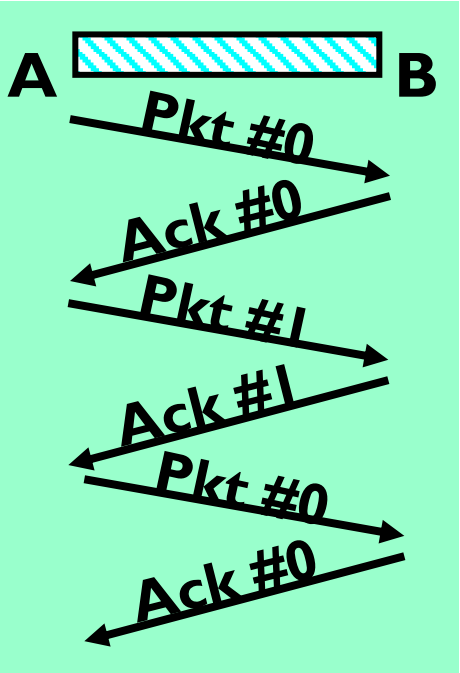


- Checksum: Detect garbled packets
- Receiver sends a packet to acknowledge when a packet received and ungarbled
  - No acknowledgement? **Resend** after timeout
- What if acknowledgement dropped?
  - Packet is resent (wasteful), second chance to acknowledge

What about duplicates?

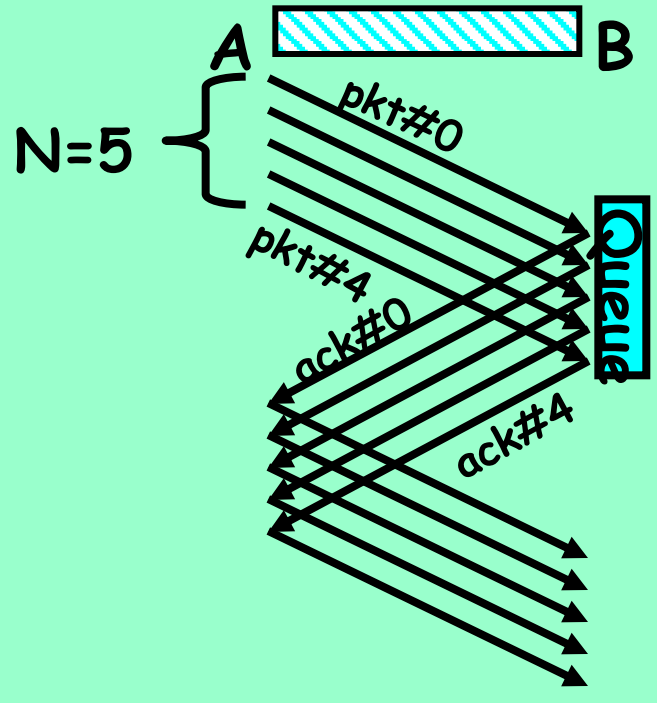
- Recall: Sequence Number
- Simplest Version: Alternating Bit Protocol

- Send only one packet at a time
  - Poor performance

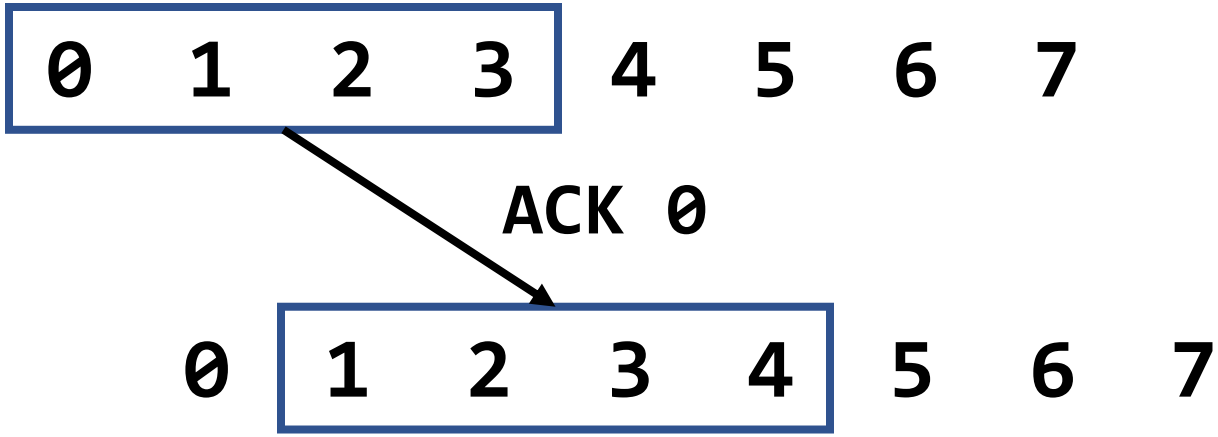


Window-Based Acknowledgements

- Windowing protocol (not quite TCP)
- Send up to *N* packets without ack
  - Allows pipelining of packets
  - *N* limits queue size
- Both source and destination need to store *N* packets (why?)
- Each packet has sequence number
  - ACK says "Received all packets up to number *X*"
  - Advances window



Sliding Window

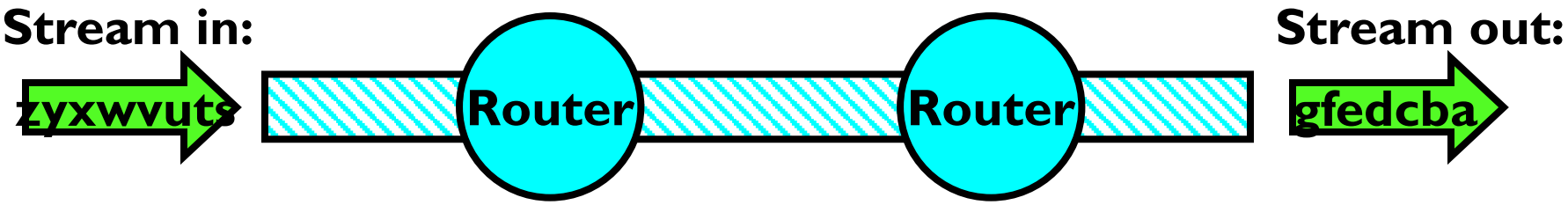


- Window represents packets:
  - That might need to be re-sent (dropped, garbled)
  - That receiver needs to buffer (in-order delivery to user)

Window-Based Acknowledgement

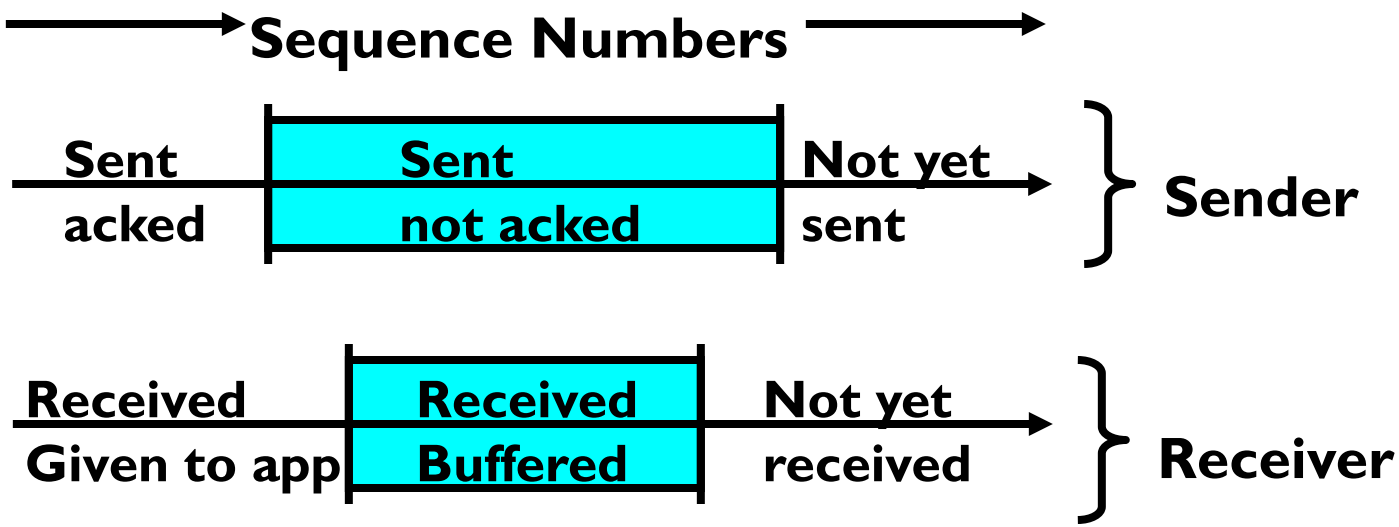
- Packet lost?
  - Resent after timeout (no ACK received)
- Acknowledgement lost?
  - Packet resent, causing ACK to be resent, too
- Discard out-of-order packets?
  - If no, need some way to indicate holes in window

Transmission Control Protocol (TCP)



- Reliable byte stream between two processes on different machines, over the Internet
- Bi-directional: two streams for every connection
- Fragments byte streams into packets, hands those to IP
- Window-based acknowledgement protocol
- Automatically retransmits lost packets
- Adjusts rate of transmission to avoid *congestion*
  - How? **Window Size**

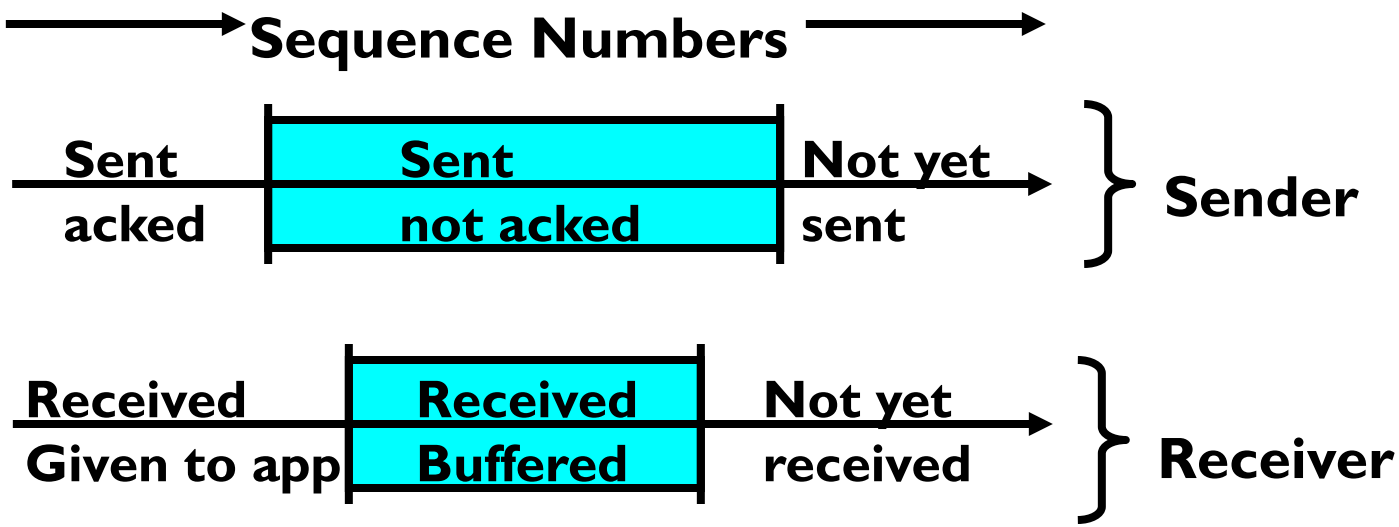
TCP Windows and Seq. Numbers



Sender has three regions:

1. Sent and acknowledged
2. Sent and not acknowledged
3. Not yet sent

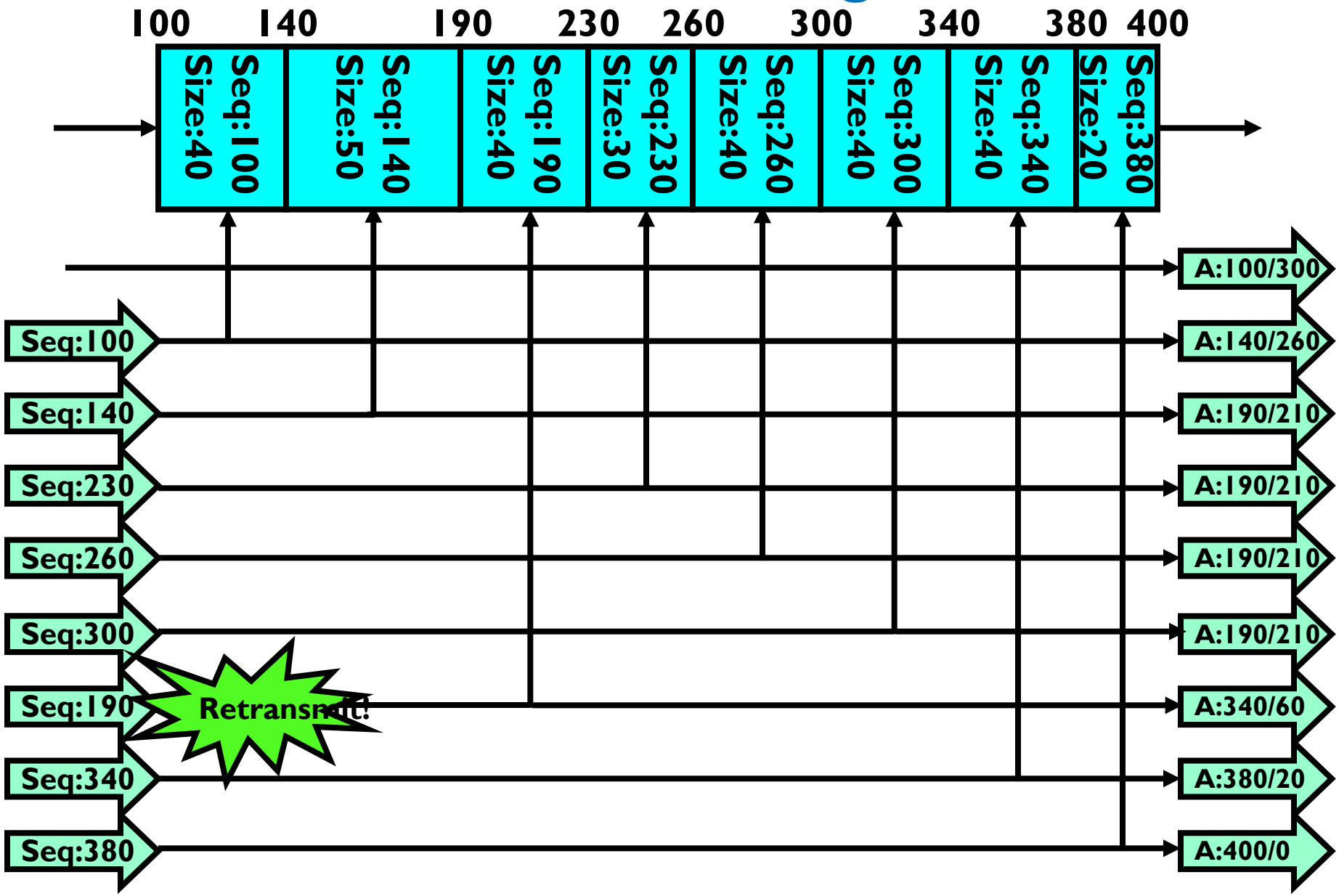
TCP Windows and Seq. Numbers



Receiver has three regions:

1. Received and acknowledged
2. Received and buffered
3. Not yet received

Window-Based Acknowledgements



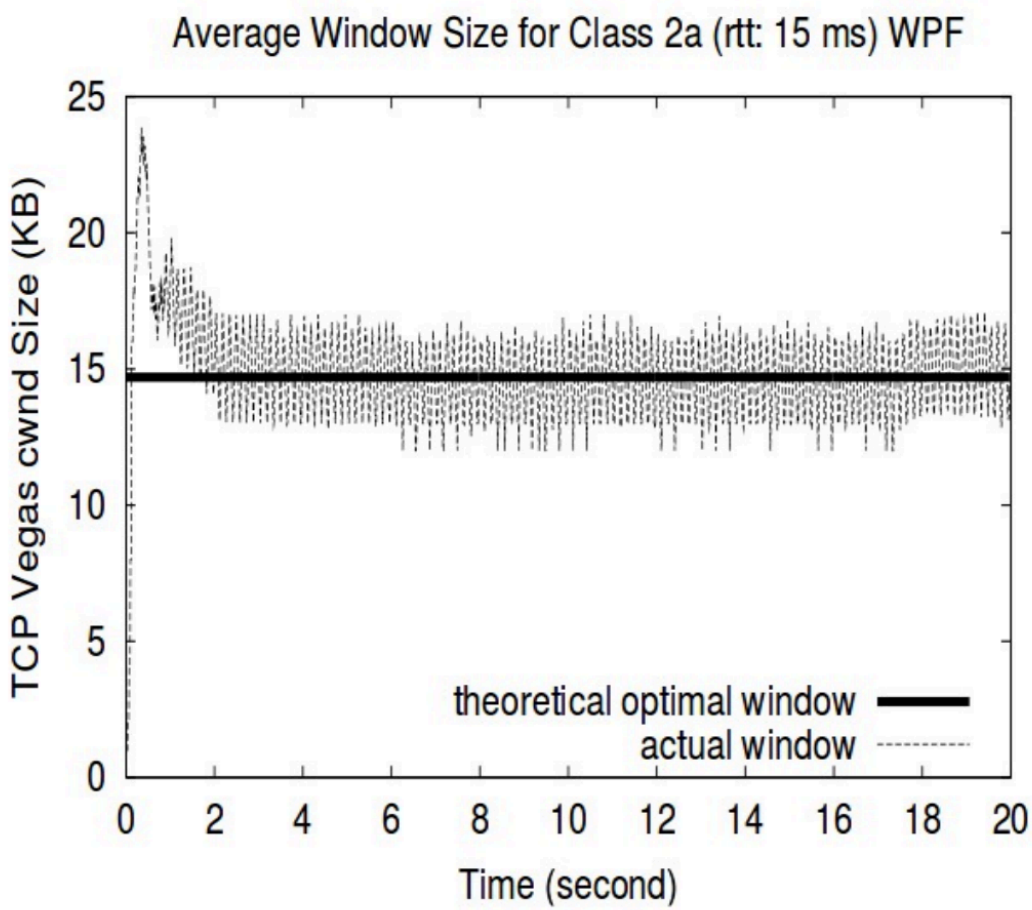
Congestion

- Too much data trying to flow through some part of the network
- IP's solution: **Drop** packets
- Bad for TCP
  - Lots of retransmission – wasted work
  - Lots of waiting for timeouts – underutilized connection

Congestion Avoidance

- Solution: Adjust **Window Size**
- AIMD: Additive Increase, Multiplicative Decrease
  - When packet dropped (missed ack), cut window size in half
  - If no timeouts, slowly increase window size by 1 for each acknowledgement received

Congestion Avoidance: Changing Window



From Low, Peterson, and Wang, "Understanding vegas: Duality Model", J. ACM, March 2002.

Summary: Network Layers

- Link Layer (local network)
  - Send *frames* addressed to neighboring machines
  - Ethernet, Wi-Fi
- Network layer (connecting local networks)
  - Forwarding between local networks
  - Send packets addressed to machines anywhere
  - IP
- Transport Layer (making streams)
  - Turn sequence of packets into reliable byte stream
  - TCP

Summary: TCP

- Use sequence numbers to solve out-of-order delivery problem
- Use acknowledgements to solve reliable delivery problem
- For better utilization, allow a *window* of unacknowledged packets
- Adjust window size in response to perceived congestion events