15-441

Computer Networking

UDP & TCP:
Transport Protocols
Oct. 27, 2004

Topics

- **What's a Transport Protocol?**
- **Internet architectural history reminder**
  - **TCP/UDP split**
- **UDP and applications**
- **TCP overview**

Slides – Randy Bryant, Hui Zhang, Dave Eckhardt

Readings

Section 2.5

- "Reliable Transmission"
  - Issues, stop&wait, sliding window

Chapter 5

- 5.1 UDP, 5.2 TCP
- 5.3 (RPC) will be addressed later (though reading early is ok)
- 5.4 (Performance) shouldn't be too painful

Architectural Reminder

CerfKahn74

- A Protocol for Packet Network Intercommunication
- Lays out fundamental Internet architectural assumptions
- Subnets will vary in terms of addressing, size, protocol
- Application protocols will be end-to-end
  - All hosts will speak same application protocols
  - File-format translation as part of one file-transfer protocol
  - No “file translation gateways” at campus boundaries
- “One protocol to bind them” - IP
- Particular “division of labor”
  - Error control is a host matter
  - Fragmentation compromise – changed by IPv6

CerfKahn74 vs. IPv4

Addresses are larger

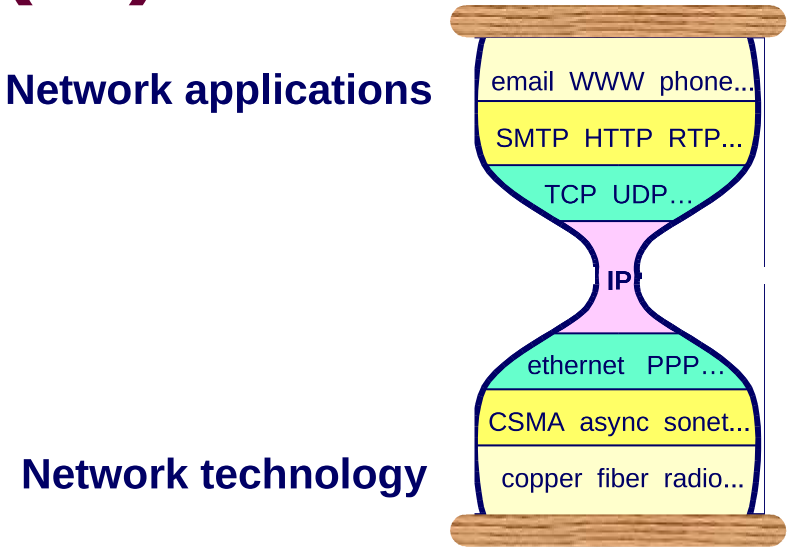
- Paper
  - 8 network bits
    - “seems sufficient for the forseeable future”
  - 16 host bits
    - “seems more than sufficient for any given network”
- IPv4 – 32 bits
- IPv6 128 bits
  - “Often” 64 network bits, 64 host bits (MAC address)

CerfKahn74 vs. IPv4

Layering split

- Paper presented “Transmission Control Program” protocol
  - One reliable in-order message-stream protocol
  - One header, so routers understood everything
- Paper's TCP split into
  - IP – host addressing, data delivery
  - TCP – reliable in-order byte-stream protocol
    - (note: “message-stream” got lost)
  - UDP – unreliable un-ordered packet protocol

Internet Protocol (IP)



Steve Deering, CISCO

IP Delivery Model

- Connectionless datagram
  - Each packet independent entity
  - Each packet contains source & destination address
- Best effort service
  - Packets may be dropped, duplicated, delivered out of order
  - No performance guarantee

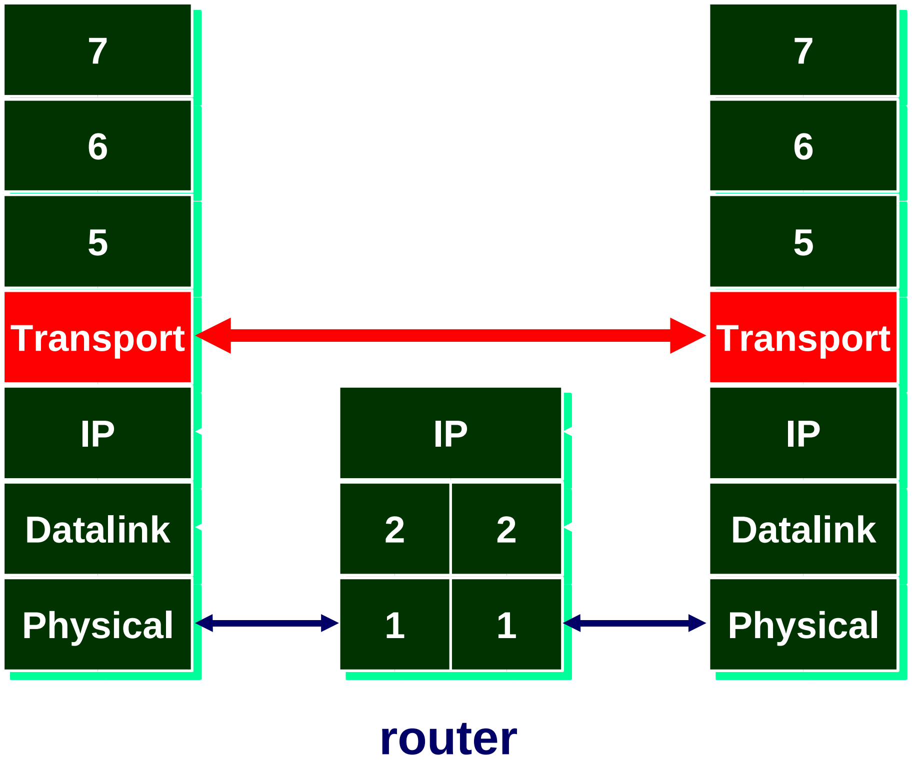
Transport Protocols

Lowest level end-to-end protocol.

- Header generated by sender is interpreted only by the destination
- Routers view transport header as part of the payload

Adds functionality to the best-effort packet delivery IP service.

- Make up for the “shortcomings” of the core network



(Possible) Transport Protocol Functions

Multiplexing/demultiplexing for multiple applications.

- “Port” abstraction abstracts OS notions of “process”

Connection establishment.

- Logical end-to-end connection
- Connection state to optimize performance

Error control.

- Hide unreliability of the network layer from applications
- Many types of errors: corruption, loss, duplication, reordering.

End-to-end flow control.

- Avoid flooding the receiver

Congestion control.

- Avoid flooding the network

User Datagram Protocol (UDP)

- **Transforms IP's connectionless datagram into... connectionless datagram!**

Addressing used for (de)multiplexing.

- **Port numbers = connection/application endpoint**

End-to-end reliability via end-to-end checksum.

- **Protects against data corruption errors between source and destination (links, switches/routers, memory bus)**
- **Does not protect against packet loss, duplication or reordering**
- **Checksum chosen to be efficient in software (vs. CRC)**
  - **Optional in theory, but you'd better use it in practice**

Source Port	Dest. Port
Length	D. Checksum

Two-Level Multiplexing

- How does the protocol stack know which application should receive a particular packet?

Each IP datagram contains “protocol ID” (UDP, TCP, ...)

- Specifies transport protocol (kernel module) to get packet

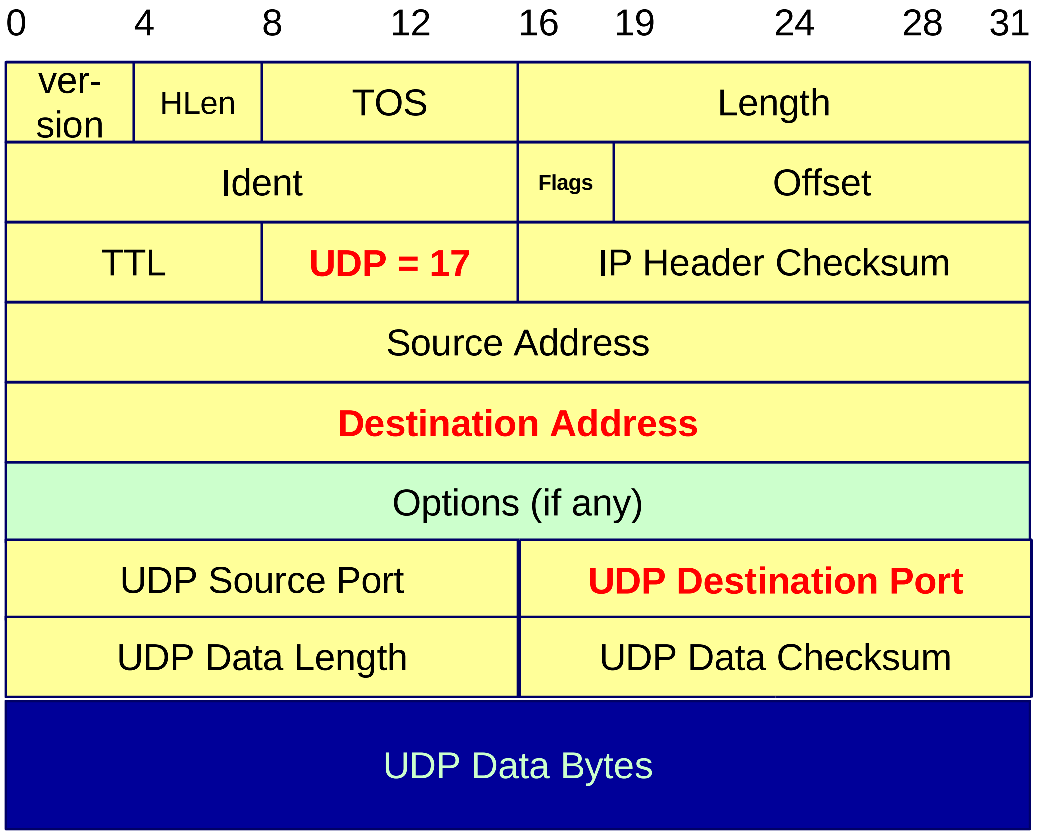
Transport layer uses the “port” field of transport header to identify the application socket.

- (Destination IP, destination port) mapped to socket
- Port numbers 0-1023 are “well-known” port numbers

UDP packets delivered to a socket can come from various sources (connectionless)

- To reply, we swap source (IP,port) with destination (IP,port)

Two-Level Multiplexing



Uses of UDP

1. Original motivator

- Experimental packet-voice protocol doesn't want TCP
  - TCP “helpfully” imposes in-order delivery
  - Audio-data packets have *independent* deadlines
    - Once packet #37 is late, it's late
    - Don't delay playing packet #38 until #37 is retransmitted

2. Architectural role

- Lab for experimental transport protocols
  - Getting a new IP-level protocol number requires results
- Use the port addressing provided by UDP
- Implement new & improved reliability, flow control, ordering, congestion control

Uses of UDP

3. Request/Response for vital Internet protocols

- DNS, NTP, DHCP, Kerberos, AFS, Zephyr, TFTP, SNMP
- Remote procedure calls
- Distributed computing communication libraries
- Easy to overlook, but...
  - Internet depends on UDP-based infrastructure protocols

Why use UDP?

- TCP connection is impossible
- TCP connection is too expensive
- TCP connection expense is wasteful
- Communication pattern isn't point-to-point

UDP Case Studies

DHCP – Dynamic Host Configuration Protocol

- TCP connection is impossible
  - We don't have an IP address yet!

DNS – Domain Name System

- TCP connection is too expensive
  - Everybody on the planet talks to root name servers
  - That would be a lot of kernel socket buffers!
- TCP connection expense is wasteful
  - TCP connection costs 5 packets (2 RTT) by itself
  - DNS query/response needs only 2 packets, 1 RTT

NTP – Network Time Protocol

- Setting your clock requires estimating latency to peer
- TCP buffering interferes with estimation

UDP Case Studies

SNMP – Simple Network Management Protocol

- TCP connection is too expensive
  - Workgroup router can't afford connection state...
  - ...would be easy denial-of-service attack

Kerberos, Zephyr

- Like DNS: many clients, request/response pattern
- TCP connection is too expensive & wasteful

TFTP

- TCP *implementation* is too expensive
  - Boot code in BIOS...size is limited

UDP Case Studies

AFS - “Andrew File System” (or not)

- Counts as “experimental transport protocol”
- In 1980's, many TCP implementations had poor throughput
- Easier to implement a similar protocol than to fix kernels
- Unclear what the “right” answer is

NFS – Sun's “Network File System”

- Similar reasons, judgement to AFS
- Lots of people run NFS over TCP

UDP Case Studies

RPC (Remote Procedure Call) libraries

- SunRPC, CORBA, DCOM, etc.
- Many operate over both UDP and TCP
- Application often selects via flag
  - Application, not library, knows how many calls to same server

Special-purpose communications

- Examples
  - ISIS distributed-computation library
  - IP multicast
- Communication pattern isn't point-to-point

Byte Stream?

TCP provides a "reliable byte-stream connection"

- **What's that?**

Byte Stream

TCP provides a “reliable byte-stream connection”

- **Connection**
  - Information is part of a “session” or “association” which lasts for longer than a single packet
  - Bytes arrive “on a connection”, not “from the network”
- **Byte-stream: write(server, “abc”, 3); write(server, “def”, 3);**
  - Server will receive 'a' before 'b', 'b' before 'c', ..., 'e' before 'f'
  - read(client, buf, 10) may receive
    - “abc”, 3
    - “abcdef”, 6
    - “a”, 1
- **Reliable**
  - Even if network loses the “abc” packet the 1st time (and 2nd...)
  - Even if network delivers “def” packet before “abc” packet

Fatal Errors

TCP provides a "reliable byte-stream connection"

- Reliable
  - Even if an asteroid lands on the server?
    - Well, no.

How do TCP applications learn about "fatal errors"?

- write(server, "query\n", 6) ⇒ -1
- read(server, answerbuf, sizeof (answerbuf)) ⇒ -1
- errno says...
  - ETIMEDOUT, ECONNRESET, ENETDOWN, EHOSTDOWN, EHOSTUNREACH

How do UDP applications learn about "fatal errors"?

- maybe just silence!
- maybe read()/write() errors as with TCP (see "ICMP")

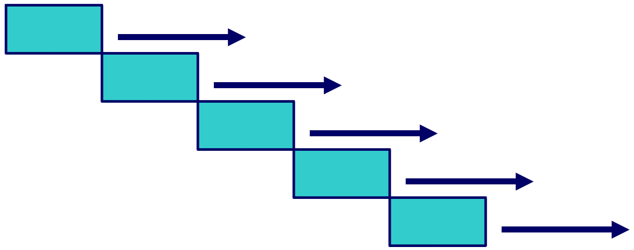
Common Byte Stream Flows

Data Transfer

- Application wants to transfer a lot of bytes from one machine to another:

Mechanism

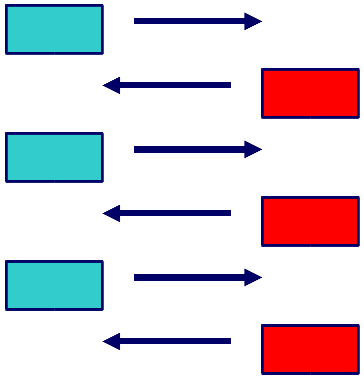
- Break into smaller segments
- Send in succession



- Reassemble at other end

Request/Response

- Interactive application involves exchange of short messages between two hosts



Mechanism

- Send each message as separate packet

TCP's Jobs

Reliable bi-directional byte stream

Connections established & torn down

Multiplexing/ demultiplexing

Error control

End-end flow control

Congestion avoidance

TCP's Jobs – In 20 bytes...

Reliable bi-directional byte stream

Connections established & torn down

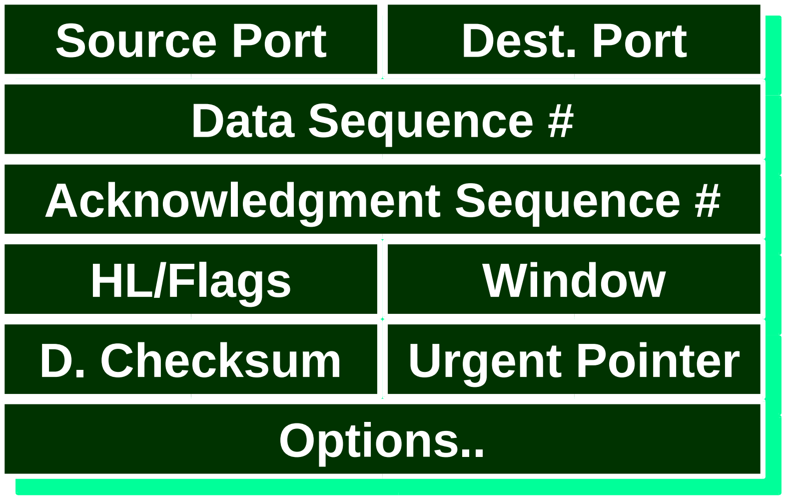
- Analogy: setting up & terminating phone call

Multiplexing/ demultiplexing

- Ports at both ends

Error control

- Users see correct, ordered byte sequences



End-end flow control

- Avoid overwhelming machines at each end

Congestion avoidance

- Avoid creating traffic jams within network

Connection Life Cycle

Choosing ports

Establishing connection

Transmitting data

Tearing down connection

Choosing Ports

"Well-known ports" used for many applications

- Mail servers listen on
  - Port 25 – SMTP (Simple Mail Transfer Protocol)
  - Port 110 – POP3 (Post Office Protocol, v3)
  - Port 143 – IMAP (Internet Mail Access Protocol)
- See "/etc/services" on a Unix machine

Random port numbers used by "clients"

- If you don't bind() before you connect(), kernel gives you one

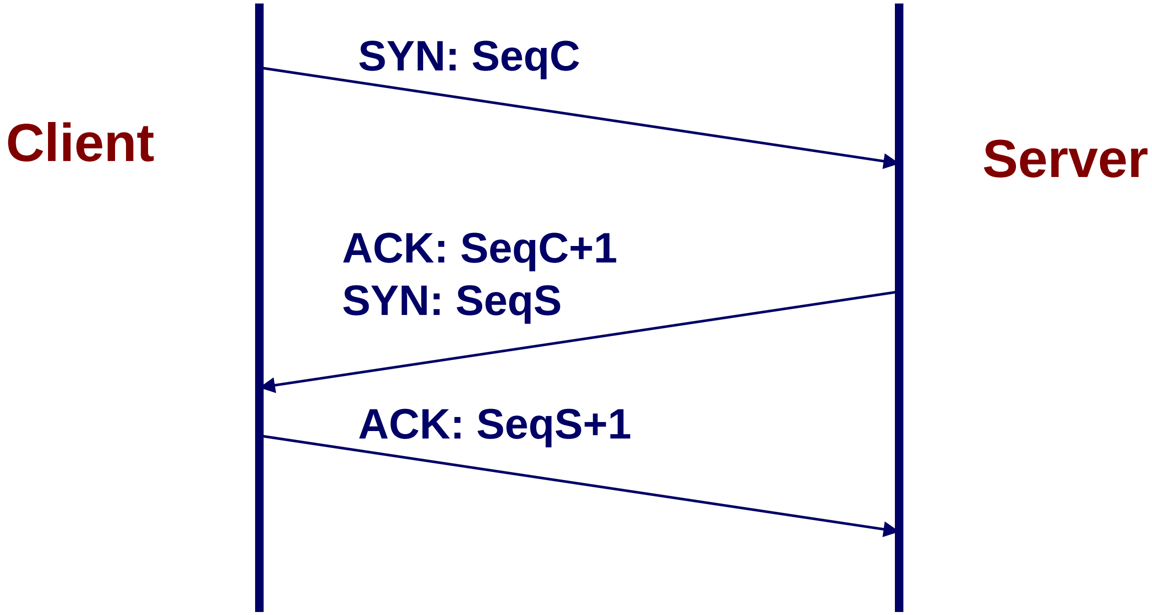
TCP connection defined by 4-tuple

- (IP1, Port1, IP2, Port2)

  (pa-mtlebanon3a-39.pit.adelphia.net, 4093,

  piper.nectar.cs.cmu.edu, 22)

Establishing Connection



Three-Way Handshake

- Each side notifies other of starting sequence number it will use for sending
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- Can "piggy-back" second SYN with first ACK

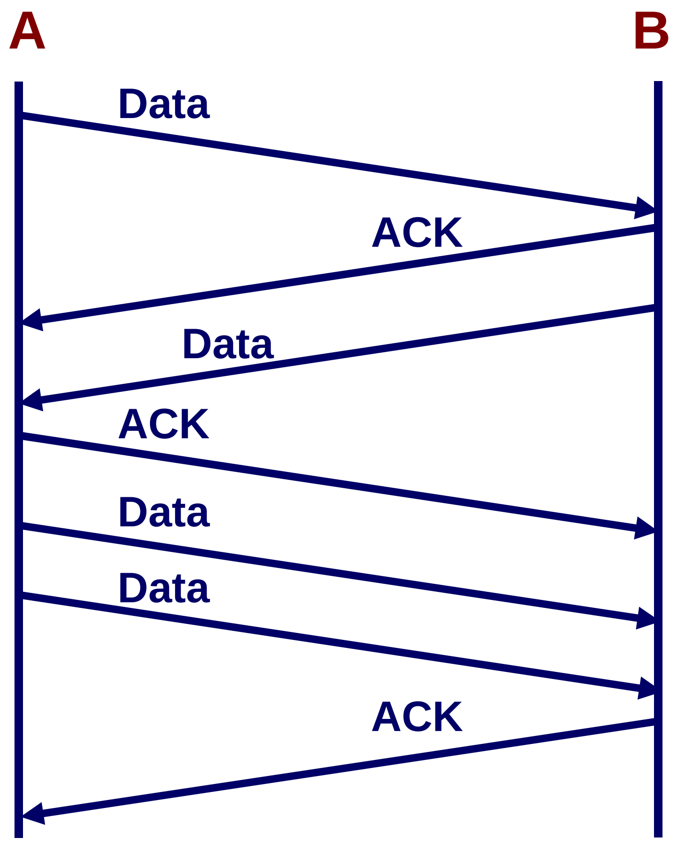
Transmitting Data

Both sides may send data

- Really *two* byte streams

"Free-form" acks

- Need not Ack every Data
- Sometimes Ack repeatedly
- Complicated!!
  - Not for today



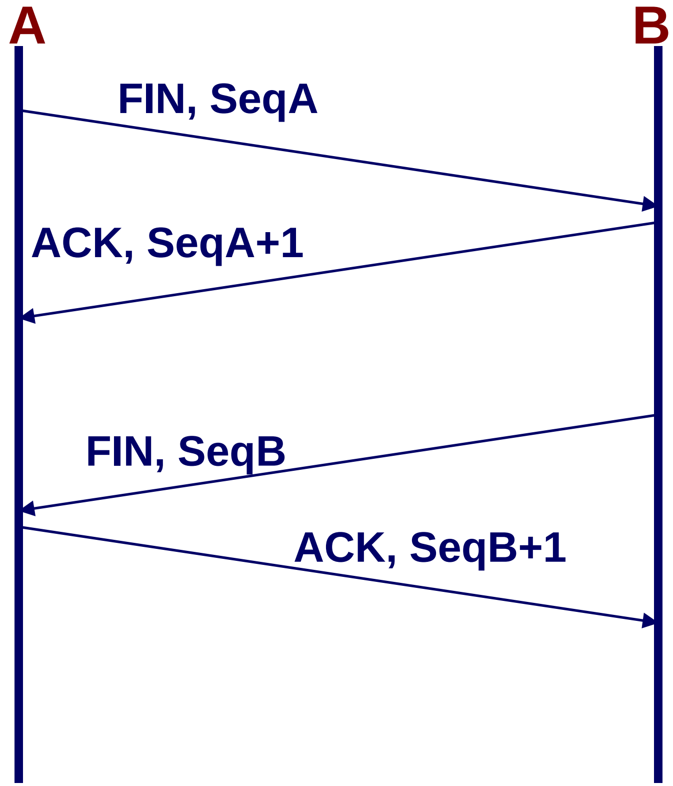
Tearing Down Connection

Either side can initiate teardown

- Send FIN signal
- “I'm FINished sending”

Other side typically agrees

- >>> QUIT
- <<< 220 Goodbye
- Both sides FIN
- Kernels sort things out



Byte Counting

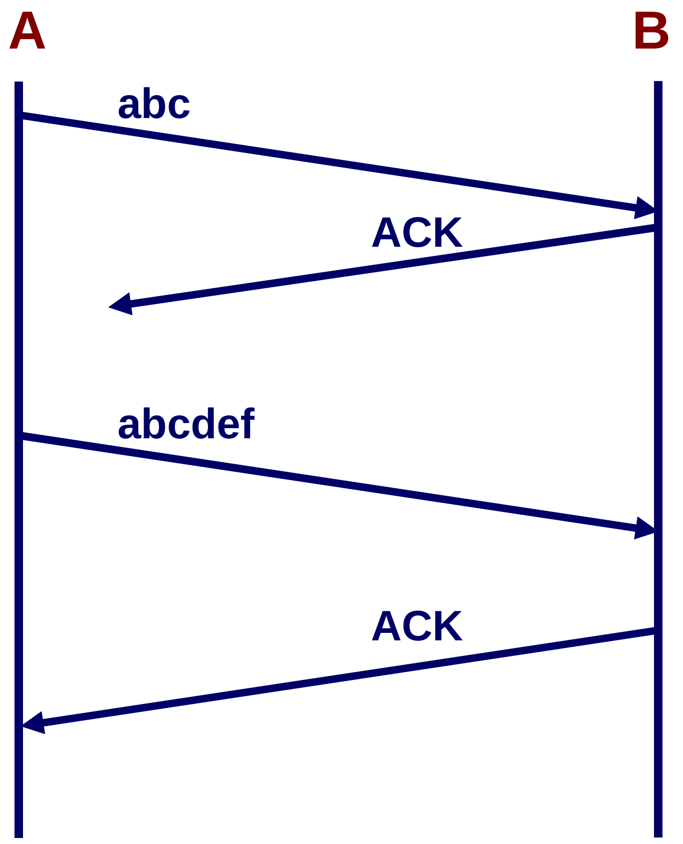
TCP sequence numbers count *bytes*, not packets

Good news

- More-efficient retransmissions

Bad news

- More-complicated receiver processing
  - Must deliver each byte to user exactly once!
- Similar to IP fragment reassembly



To Nagle or not to Nagle?

Problem (Nagle, RFC 896, 1984)

- Sending a TCP packet when a user types one character considered harmful
- 1 byte of data, 40 bytes of header...4000% overhead
- Cost of processing a packet at a router has large fixed component
- Already-busy network may be driven to “congestion collapse”

Approach

- write() shouldn't always result in sending a packet
- Sometimes TCP sender should buffer data w/o sending
- Old solution: buffer for some amount of time (e.g., 200 ms)
- Problem: hard to set the threshold one way for everybody

To Nagle or not to Nagle?

Suggestion (Nagle, RFC 896, 1984)

- When new bytes arrive from user program, examine TCP transmit status
- If you are still waiting for an Ack for some data, buffer the bytes, send the next time you send something anyway
  - Typically on receipt of an Ack
- Otherwise, connection was idle, may as well send

Results

- Dramatic decrease in number of tiny packets
- Annoying for some borderline connection latencies

Who cares?

- Easy to do with byte-oriented protocol, hard if packet-based

Summary

What's a Transport Protocol?

- Internet architectural history reminder
  - TCP/UDP split
- UDP and applications
- TCP overview