

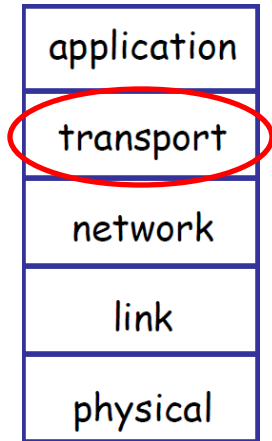
Network Applications and Transport Services

Outline

- Context/overview
- Network application design principles
- Applications and protocols (Application Layer)
- Transporting application messages (Transport Layer)
 - Transport services
 - Transport vs. network layer
 - Multiplexing and demultiplexing
 - Control-functionalities (brief introduction)
 - Transport layer in the Internet: UDP and TCP

Internet protocol stack

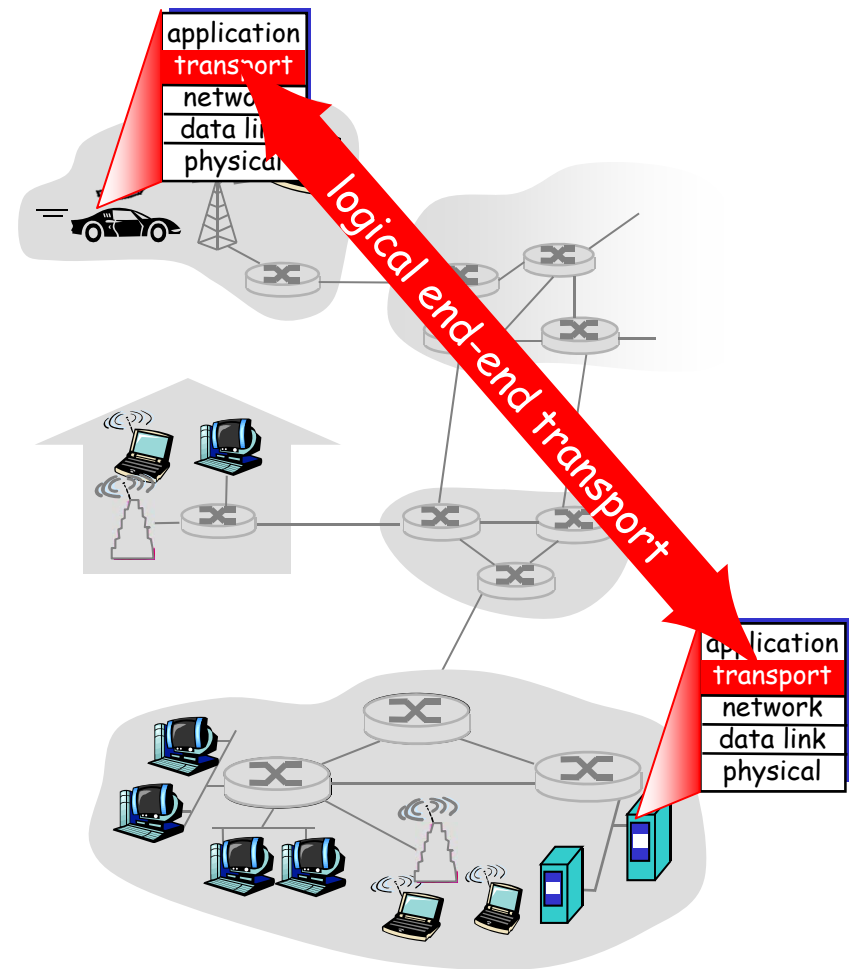
Transport Layer



- Transports application data end-to-end from a process in a host to a process in another host
 - Recall that a host/machine may have many processes!
- In the Internet, there are two transport layer protocols:
 - TCP: Transmission Control Protocol
 - UDP: User Datagram Protocol
- It provides two types of services to its applications: connection oriented service (TCP) and connectionless service (UDP)
 - Services provided by TCP include:
 - Reliable stream transfer (error control)
 - Flow control
 - Congestion control
 - UDP services:
 - Quick-and-simple single-block transfer
- Port numbers enable multiplexing. Transport layer uses the port number to pass the data to the respective process in a machine (many processes in a host!).
- Message segmentation and reassembly
- Data entities transmitted at the transport layer are referred to as **segments**.

Transport services and protocols

- ❑ provide *logical communication* between app processes running on different hosts
- ❑ transport protocols run in end systems
 - send side: breaks app messages into **segments**, passes to network layer
 - rcv side: reassembles segments into messages, passes to app layer
- ❑ more than one transport protocol available to apps
 - Internet: TCP and UDP



Transport vs. network layer

- ❑ *network layer*: logical communication between hosts
- ❑ *transport layer*: logical communication between processes
 - relies on, enhances, network layer services

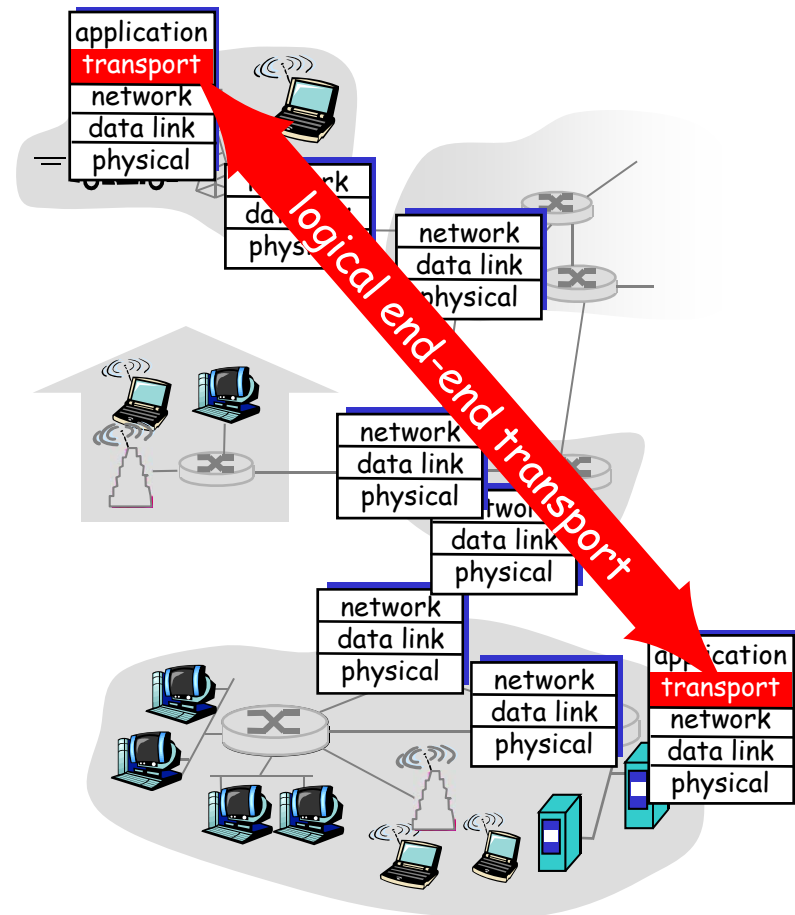
Household analogy:

12 kids sending letters to 12 kids

- ❑ processes = kids
- ❑ app messages = letters in envelopes
- ❑ hosts = houses
- ❑ transport protocol = Ann and Bill
- ❑ network-layer protocol = postal service

Internet transport-layer protocols

- ❑ reliable, in-order delivery (TCP)
 - congestion control
 - flow control
 - connection setup
- ❑ unreliable, unordered delivery: UDP
 - no-frills extension of "best-effort" IP
- ❑ services not available:
 - delay guarantees
 - bandwidth guarantees




Multiplexing/demultiplexing

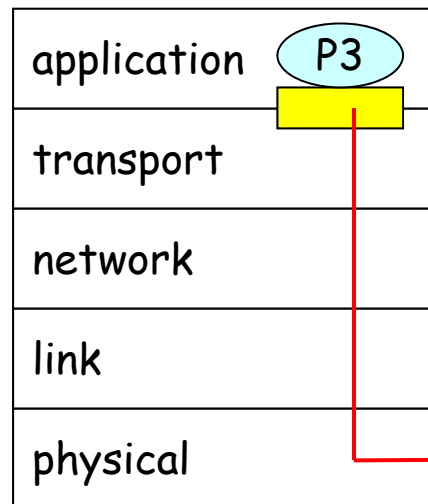
Demultiplexing at rcv host:

delivering received segments to correct socket

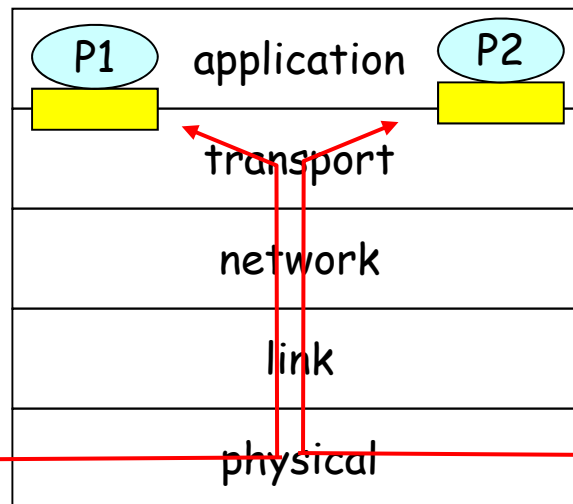
Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

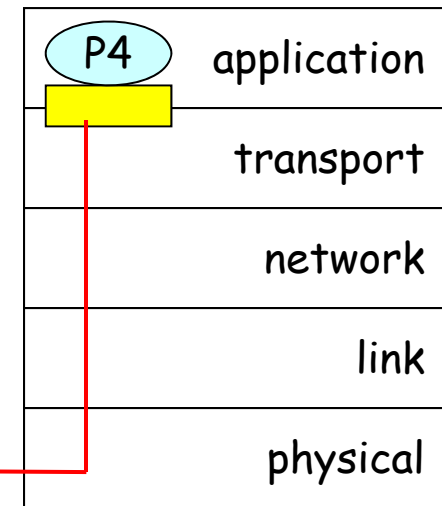
 = socket  = process



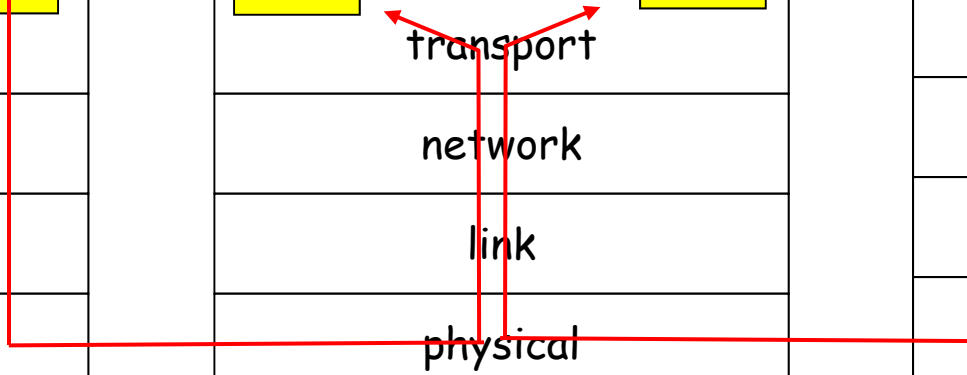
host 1



host 2



host 3

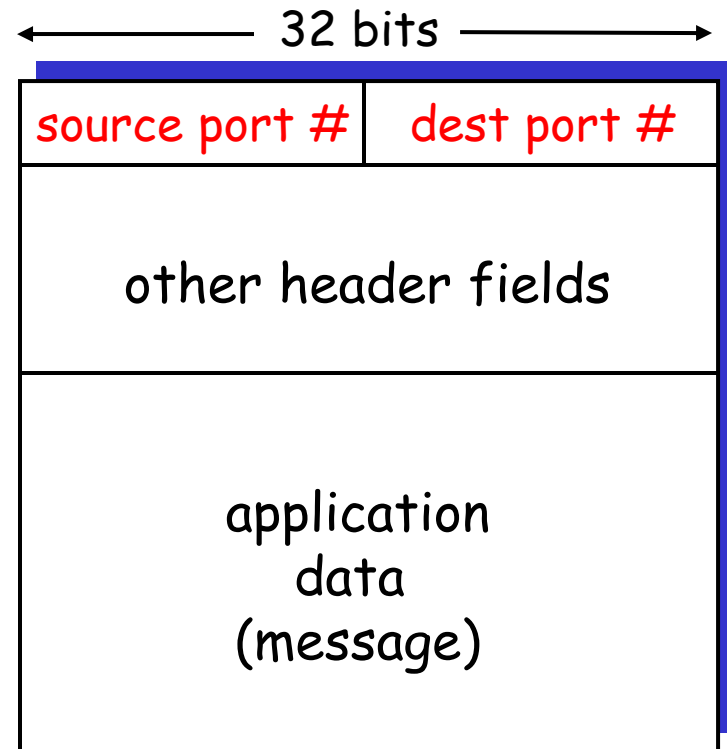


How demultiplexing works

□ host receives IP datagrams

- each datagram has source IP address, destination IP address
- each datagram carries 1 transport-layer segment
- each segment has source, destination port number

□ host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- ❑ Create sockets with port numbers:

```
DatagramSocket mySocket1 = new  
    DatagramSocket(12534);
```

```
DatagramSocket mySocket2 = new  
    DatagramSocket(12535);
```

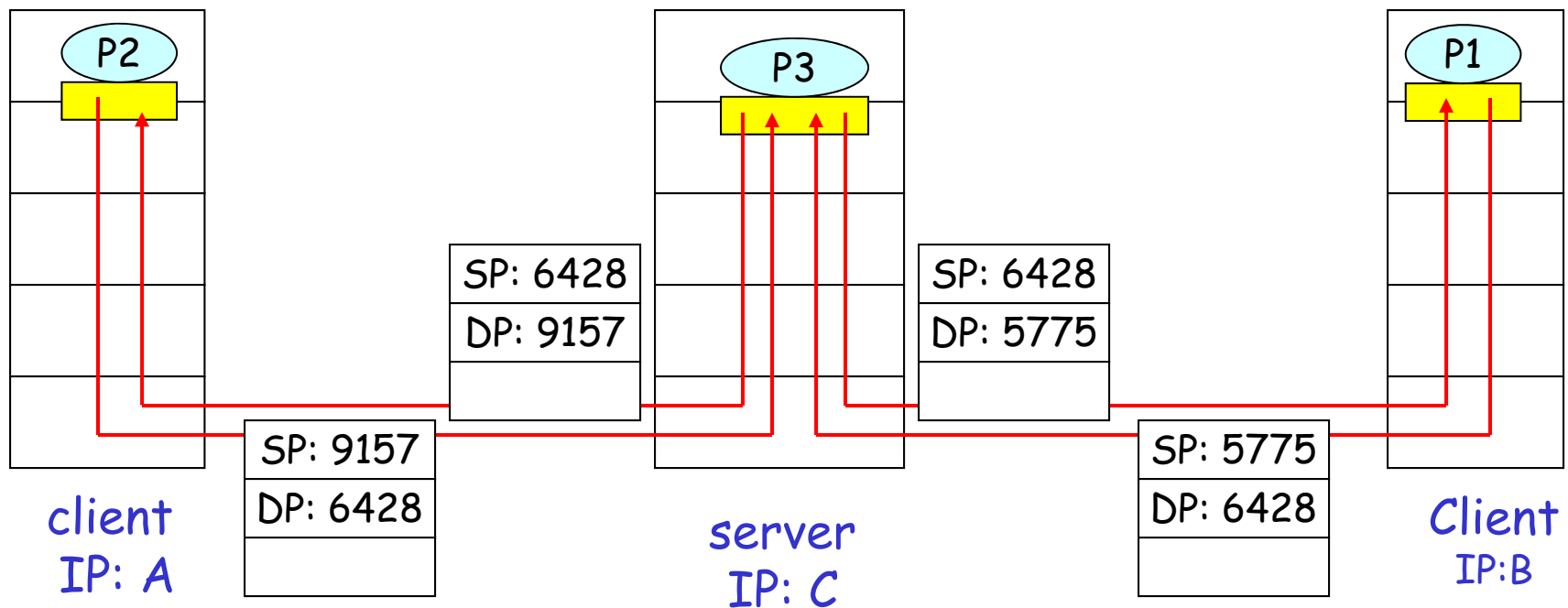
- ❑ UDP socket identified by two-tuple:

(dest IP address, dest port number)

- ❑ When host receives UDP segment:
 - checks destination port number in segment
 - directs UDP segment to socket with that port number
- ❑ IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

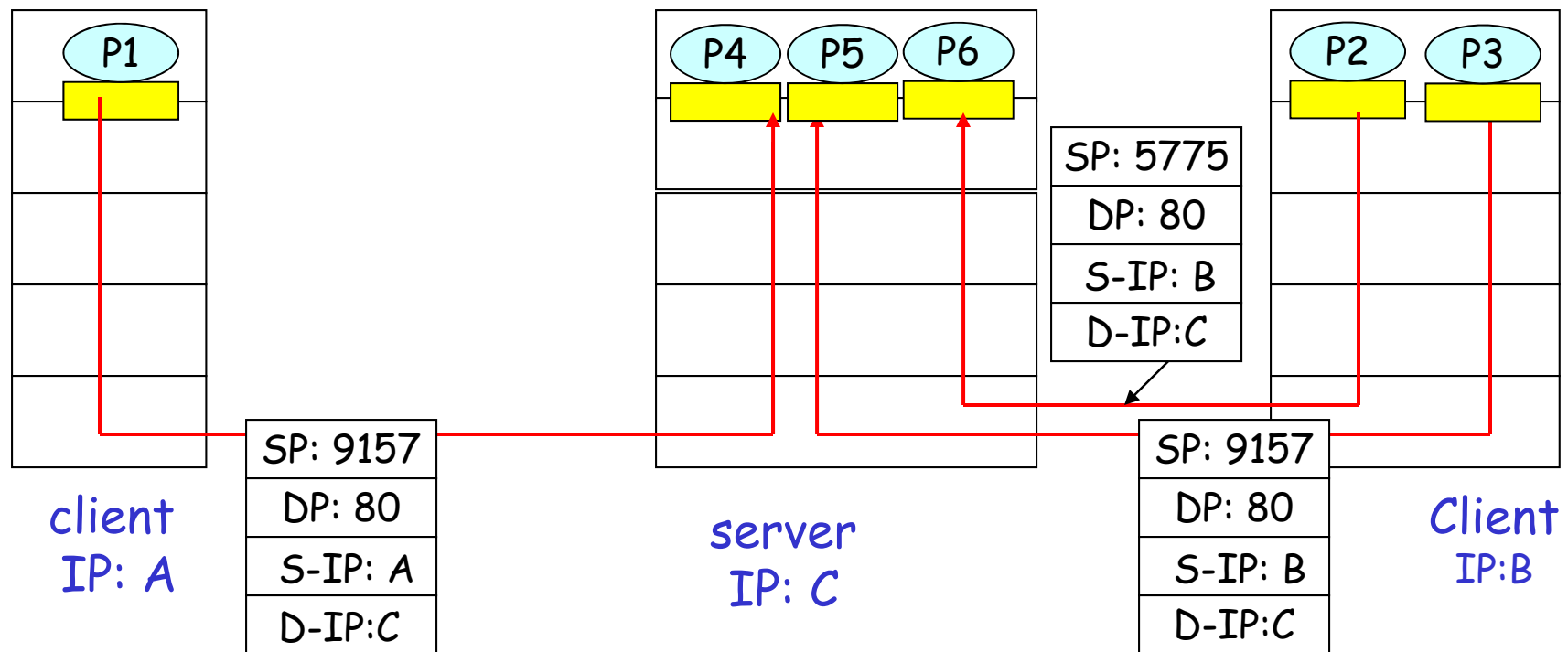


SP provides "return address"

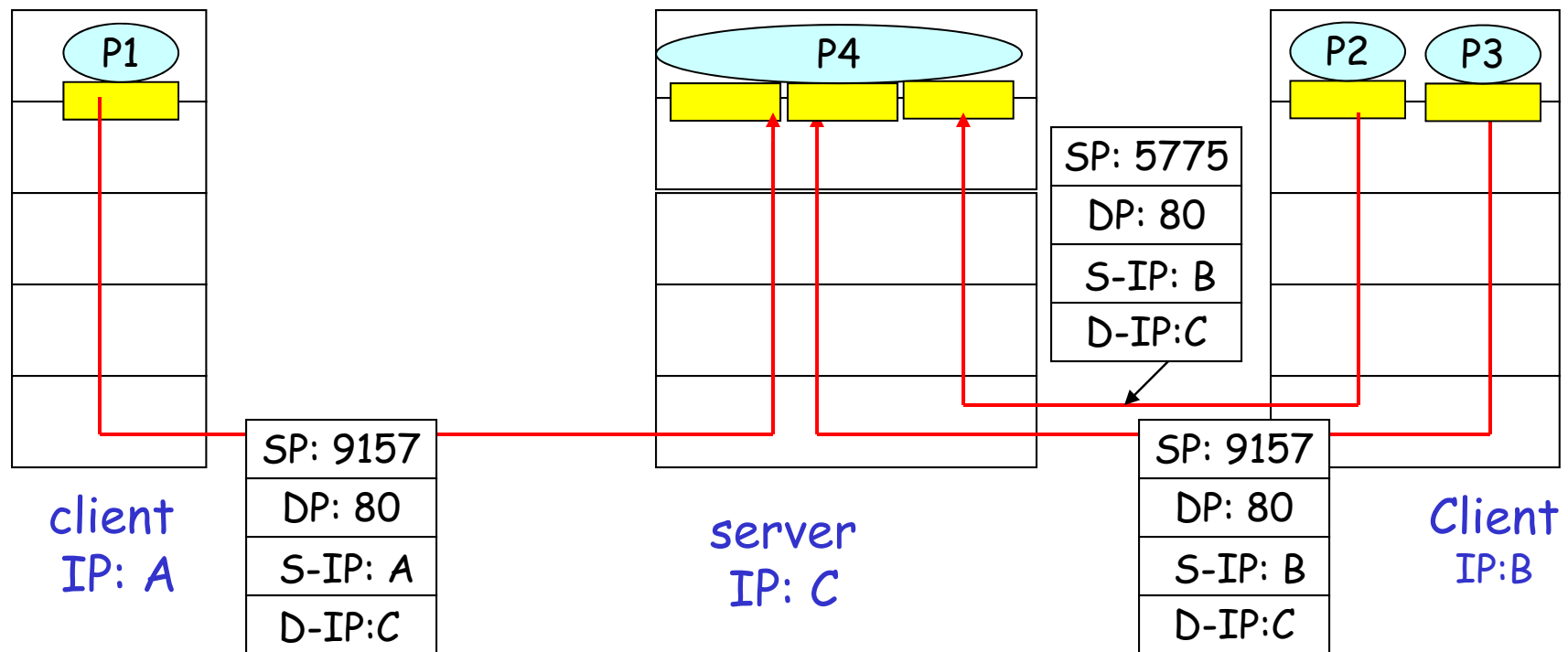
Connection-oriented demux

- ❑ TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- ❑ recv host uses all four values to direct segment to appropriate socket
- ❑ Server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
- ❑ Web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



Control-functionalities (TCP)

- Brief introduction:
 - Reliability (error and loss control):
 - (1) No bit errors (flipped from 0 to 1, or vice versa) (corrupt data/bits)
 - (2) No loss
 - (3) In-order delivery
 - Flow control: sender should not overflow receiver's buffer by transmitting too much, too fast
 - Congestion control: Prevent problems such as delays and loss due to higher rate of inputs to a router/node than outputs
- Control functionalities in other layers such as Link Layer and Application Layer.
- End-to-end versus hop-by-hop application of control functionalities.

UDP: User Datagram Protocol [RFC 768]

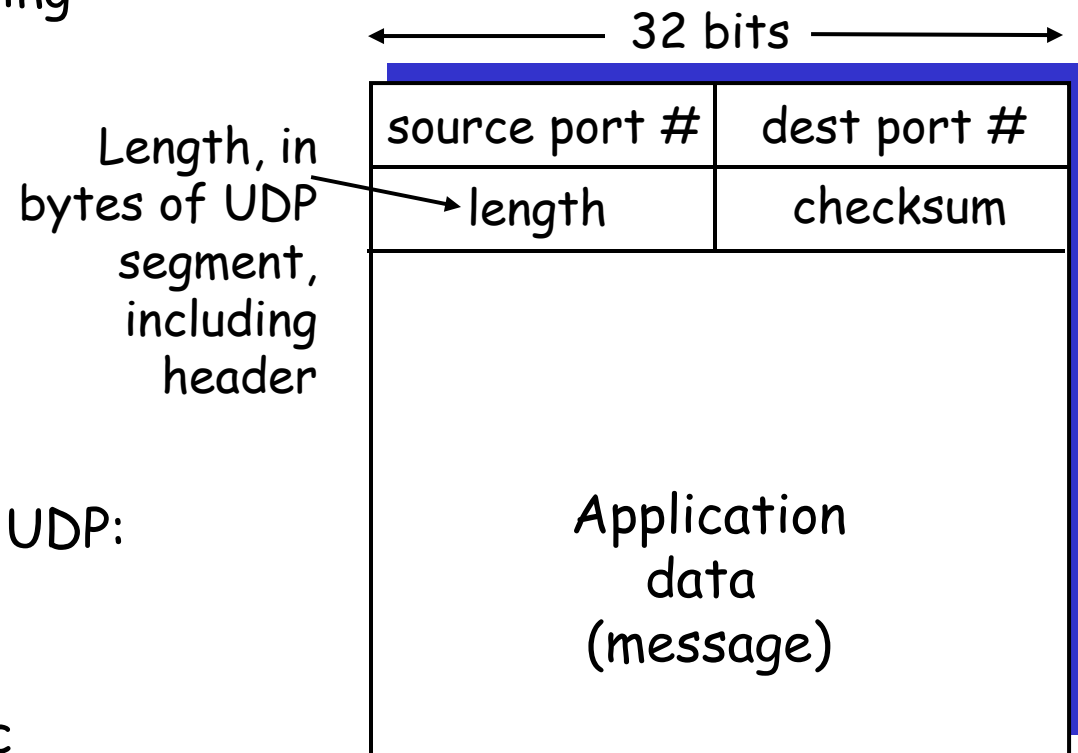
- ❑ “no frills,” “bare bones” Internet transport protocol
- ❑ “best effort” service, UDP segments may be:
 - lost
 - delivered out of order to app
- ❑ *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

Why is there a UDP?

- ❑ no connection establishment (which can add delay)
- ❑ simple: no connection state at sender, receiver
- ❑ small segment header
- ❑ no congestion control: UDP can blast away as fast as desired

UDP: more

- ❑ often used for streaming multimedia apps
 - loss tolerant
 - rate sensitive
- ❑ other UDP uses
 - DNS
 - SNMP
- ❑ reliable transfer over UDP:
add reliability at application layer
 - application-specific error recovery!



UDP segment format

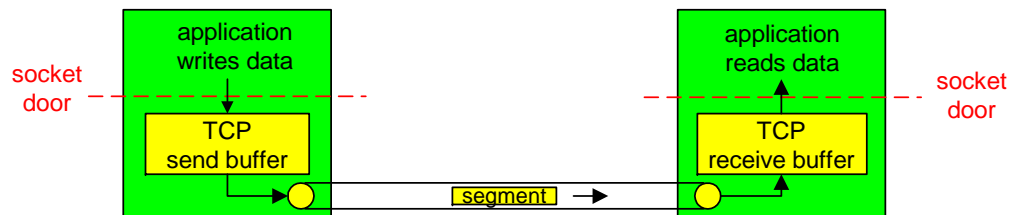
Popular Internet applications and their underlying transport protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- ❑ **point-to-point:**
 - one sender, one receiver
- ❑ **reliable, in-order *byte stream*:**
 - no "message boundaries"
- ❑ **pipelined:**
 - TCP congestion and flow control set window size
- ❑ ***send & receive buffers***



- ❑ **full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- ❑ **connection-oriented:**
 - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- ❑ **flow controlled:**
 - sender will not overwhelm receiver

TCP connection

- TCP is **connection-oriented**
 - Three phases: connection set up, message/data transmission and connection release
 - Before one application process can begin to send data to another using TCP, the two processes must first “handshake” with each other—that is, they must send some preliminary segments (control messages) to each other to establish the parameters of the ensuing data transfer.

More details:

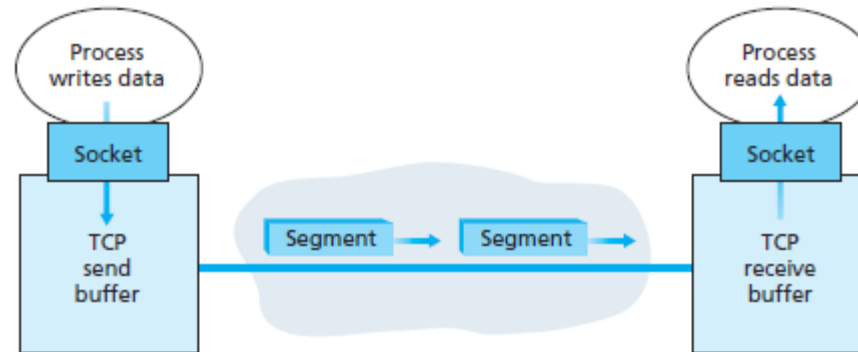
- How a TCP connection is established between a client process and a server process?
- The client application process first informs the client transport layer that it wants to establish a connection to a process in the server (a client program can do this by issuing a command)
- TCP in the client then proceeds to establish a TCP connection with TCP in the server which involves the following steps:
 - 1) The client first sends a special TCP segment (with no payload or application-data)
 - 2) The server responds with a second special TCP segment (no payload or application-data)
 - 3) Finally the client responds again with a third special segment (may carry payload or application-data)
- Because three segments are sent between the two hosts, this connection-establishment procedure is often referred to as a **three-way handshake**.
- In this TCP connection establishment, both sides of the connection will initialize many TCP **state** variables (such as sequence numbers), and buffers associated with the TCP connection (used for functionalities such as reliability, flow control, ...).
- Once a TCP connection is established, the two application processes can send data to each other.

TCP connection...

- The TCP “connection” is
 - **not** an end-to-end TDM or FDM circuit as in a circuit switched network
 - **not** a virtual circuit as the connection state resides entirely in the two end systems
- The TCP protocol runs only in the end systems and not in the intermediate network elements (routers and link-layer switches) and **the intermediate network elements do not maintain TCP connection state** (the intermediate routers are completely oblivious to TCP connections. They see datagrams, not connections)
- A TCP connection provides a **full-duplex service** (bi-directional data flow (simultaneous) in the same connection)
- A TCP connection is also always **point-to-point** , that is between a single sender and a single receiver (‘multicasting’- the transfer of data from one sender to many receivers in a single send operation- is not possible with TCP)

TCP connection...

- More details on 'sending data from the client process to the server process':



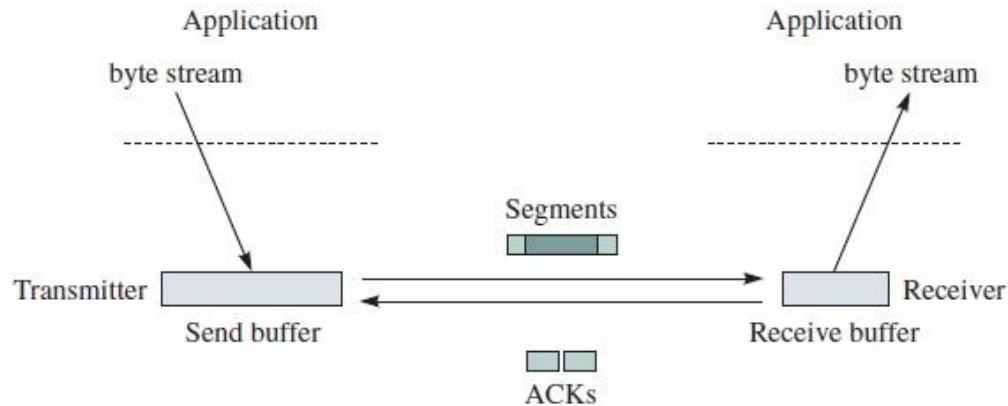
- The client process passes a stream of data through the socket (the door of the process).
- Once the data passes through the door, the data is in the hands of TCP running in the client.
- TCP directs this data to the connection's **send buffer** (set aside during the initial handshake).
- From time to time, TCP will grab chunks of data from the send buffer, make segments by attaching headers, and pass them to the network layer.
 - The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS)**.
 - The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame.
 - Both Ethernet and PPP link-layer protocols have an MSS of 1,500 bytes.
- At the receiver, when TCP receives a segment, the segment's data is placed in the TCP connection's receive buffer. The application reads the stream of data from this buffer.
- Each side of the connection has its own send buffer and its own receive buffer.
- Variables, buffers (set during handshake), and control info in the segment headers are used for providing functionalities such as reliability, flow control etc.

TCP connection...

- A TCP connection consists of buffers, variables, and a socket connection to a process in one host, and another set of buffers, variables, and a socket connection to a process in another host.
- As mentioned earlier, no buffers or variables are allocated to the connection in the network elements (routers, switches, and repeaters) between the hosts.

TCP: reliable, in-order byte stream

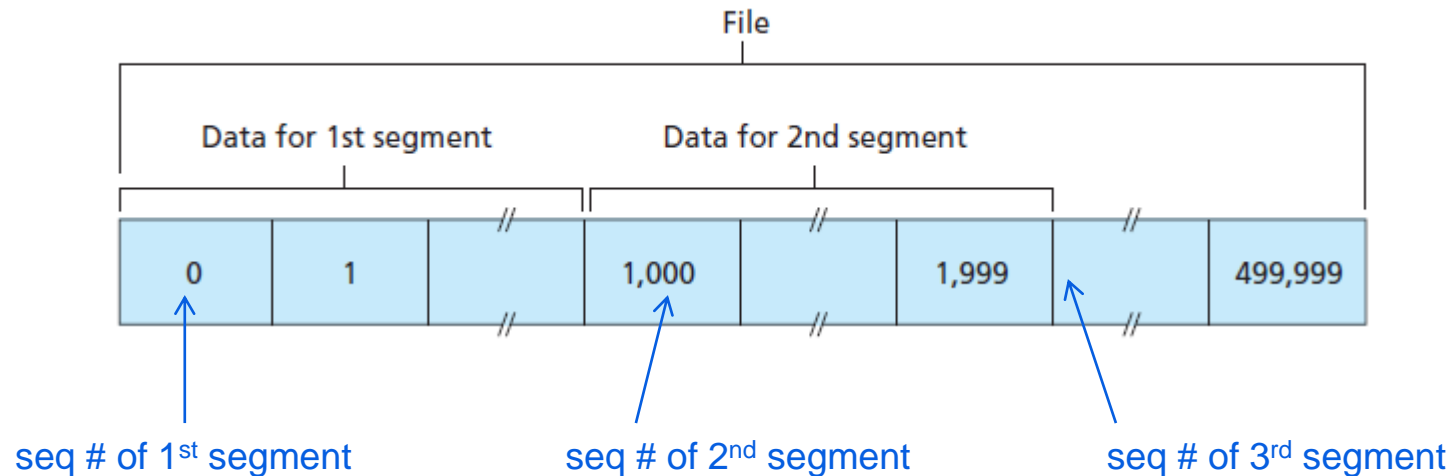
- TCP provides reliable, in-order **byte stream**



- TCP views application data as a stream of bytes
- TCP does NOT preserve “message boundaries” and treats the data it gets from the application layer as a byte stream.
 - E.g., A 3000 byte message may be transmitted in 2 segments or in 3 segments,...
- For instance, in file transfer, sender is viewed as inserting a byte stream into the TCP send buffer and TCP’s job is to ensure transfer of the byte stream to the receiver (reliable transfer)
- TCP’s use of sequence numbers reflects its view of data as ‘stream of bytes’
 - Sequence # are over stream of transmitted bytes and not over segments

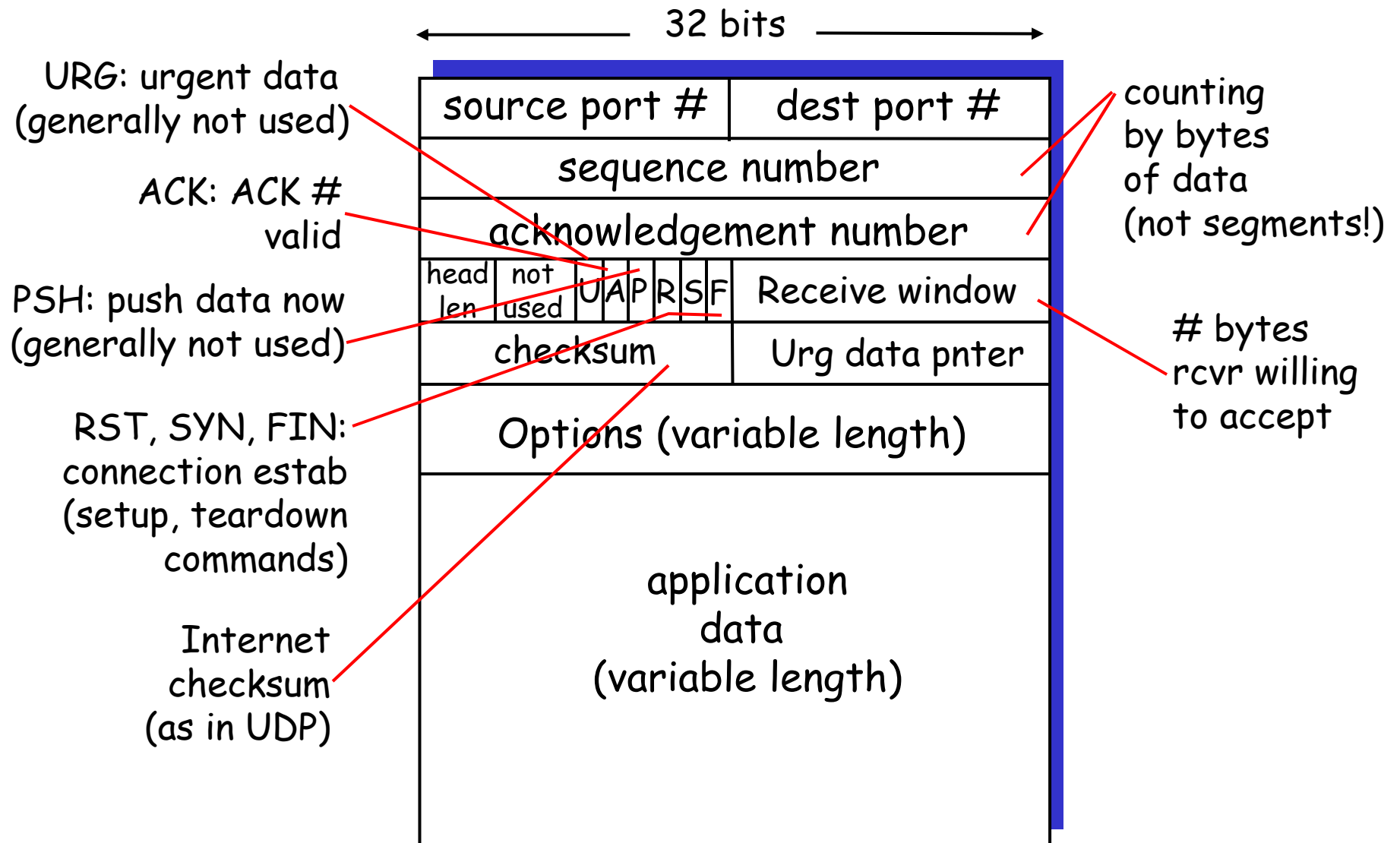
TCP: reliable, in-order byte stream

- E.g., Dividing a file into segments
 - File size: 500,000 bytes (stream of bytes from 0 to 499,999, Assume initial sequence # is 0)
 - MSS: 1000 bytes (TCP constructs 500 segments out of the data stream)
 - 1st segment contains bytes from 0 to 999
 - 2nd segment contains bytes from 1000 to 1999
 - ...
 - Assume initial sequence # is 0. In practice, a random number is selected.
 - Note that sequence # of segments are **NOT** 0, 1, 2, 3, ... but **0, 1000, 2000, ...**



- The **sequence number for a segment** is the byte-stream number of the first byte in the segment.
- TCP views application data as a stream of bytes. TCP's use of sequence numbers reflects this view.

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

- byte stream
"number" of first
byte in segment's
data

ACKs:

- seq # of next byte
expected from
other side
- cumulative ACK

Q: how receiver handles
out-of-order segments

- A: TCP spec doesn't
say, - up to
implementor

