Transport Layer (contd..)

Lec 14

Multiplexing and demultiplexing

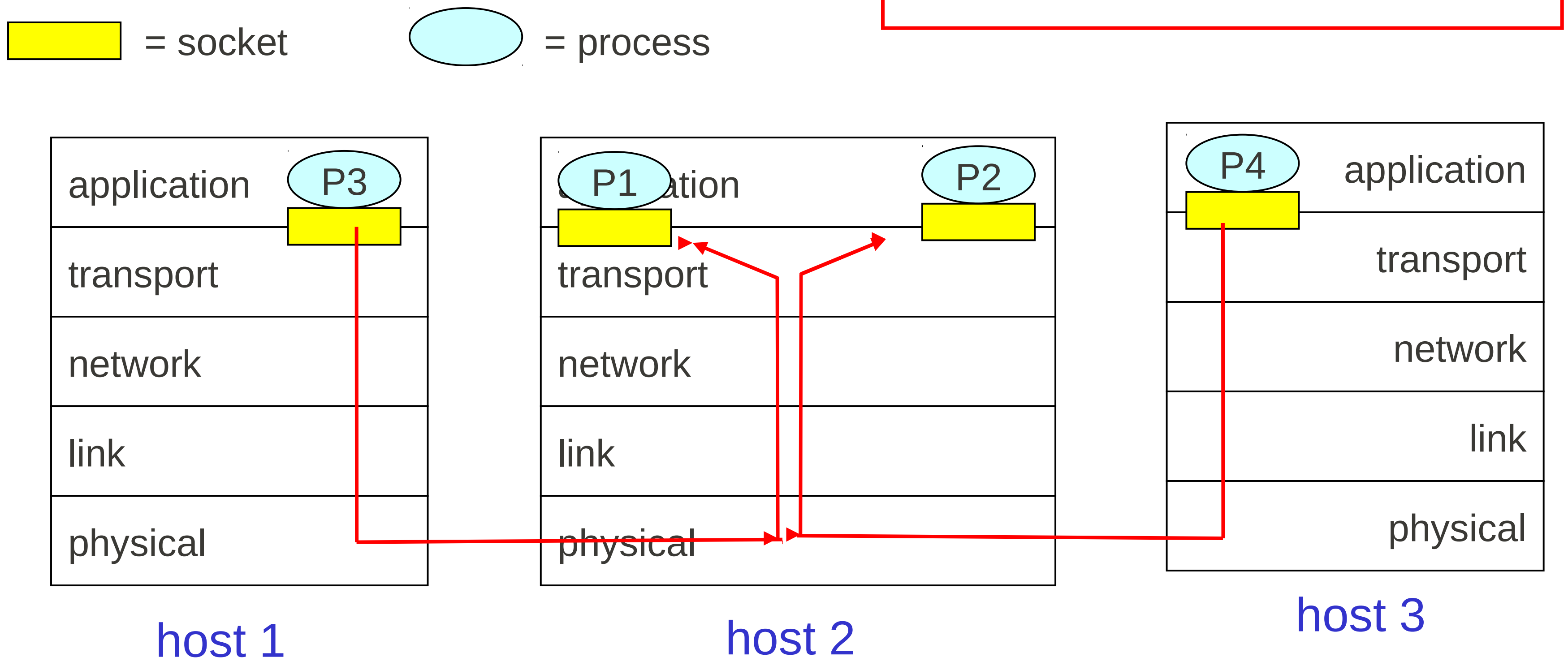
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

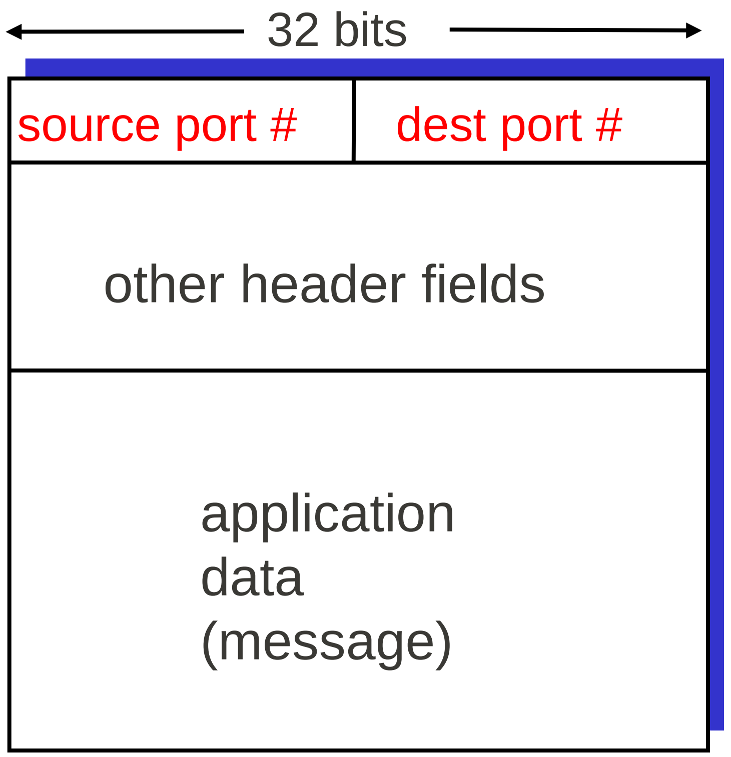
Multiplexing at send host:

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



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);
```

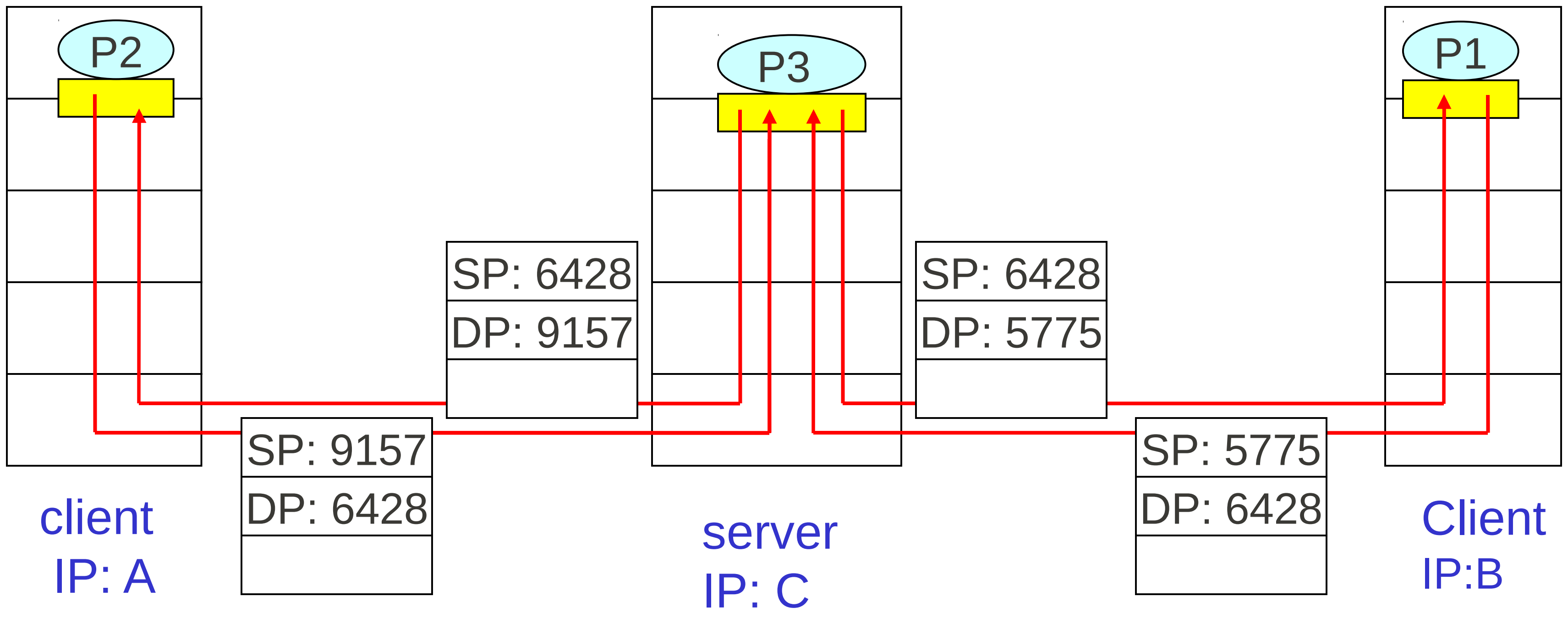
- UDP socket identified by two-tuple:

(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);
```



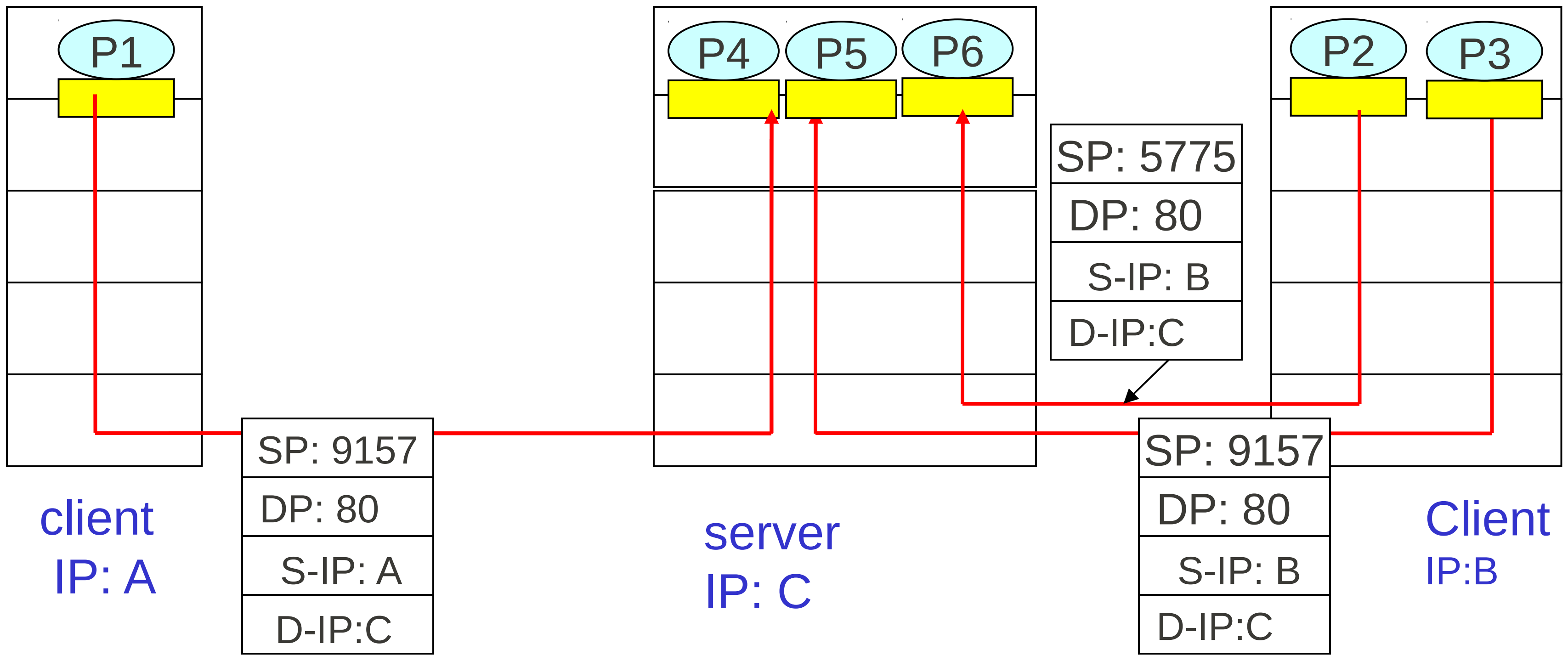
SP provides "return address"

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

Connection-oriented demux (cont)



Connectionless transport: UDP

UDP: User Datagram Protocol [RFC 768]

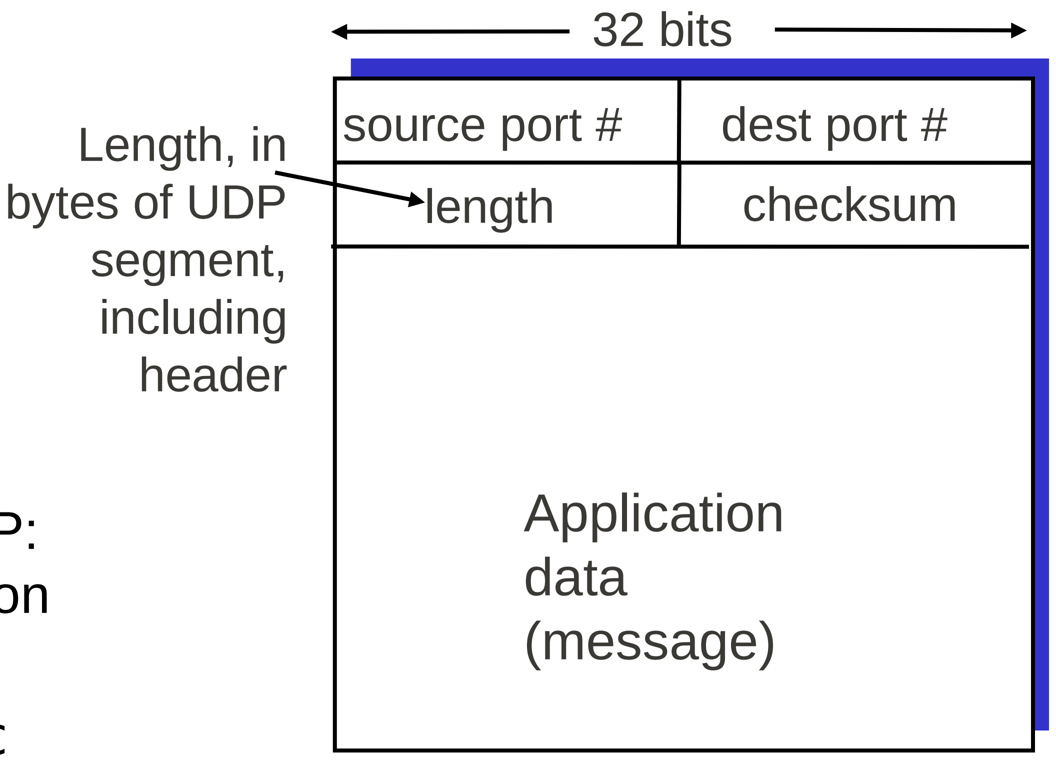
- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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!



32 bits
Length, in bytes of UDP segment, including header
source port #
dest port #
length
checksum
Application data (message)

UDP segment format

UDP checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet Checksum Example

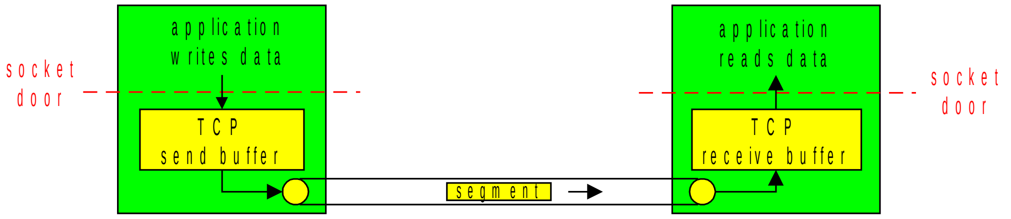
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

```
               1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
               1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
             ------------------------------------
wraparound   1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
             ------------------------------------
       sum     1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
  checksum     0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - 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