CSCE 463/612
Networks and Distributed Processing
Spring 2025

Transport Layer

Dmitri Loguinov
Texas A&M University

February 20, 2025

Chapter 3: Transport Layer

Our goals:

- Understand principles behind transport layer services:
  - Multiplexing/demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport

Application (5)
Transport (4)
Network (3)
Data-link (2)
Physical (1)

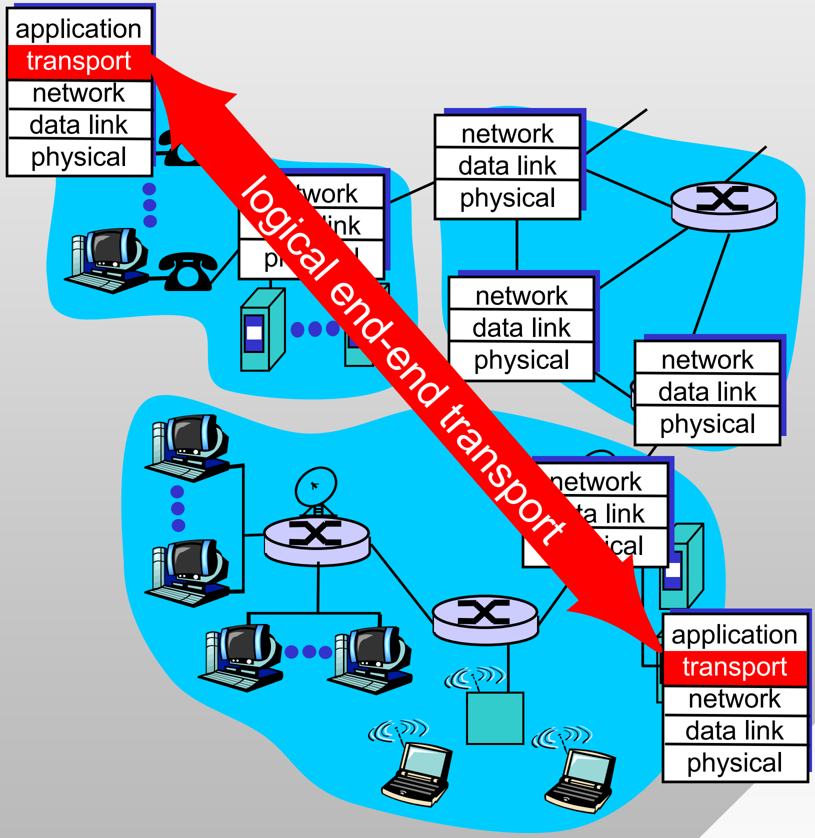
Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control
3.7 TCP congestion control

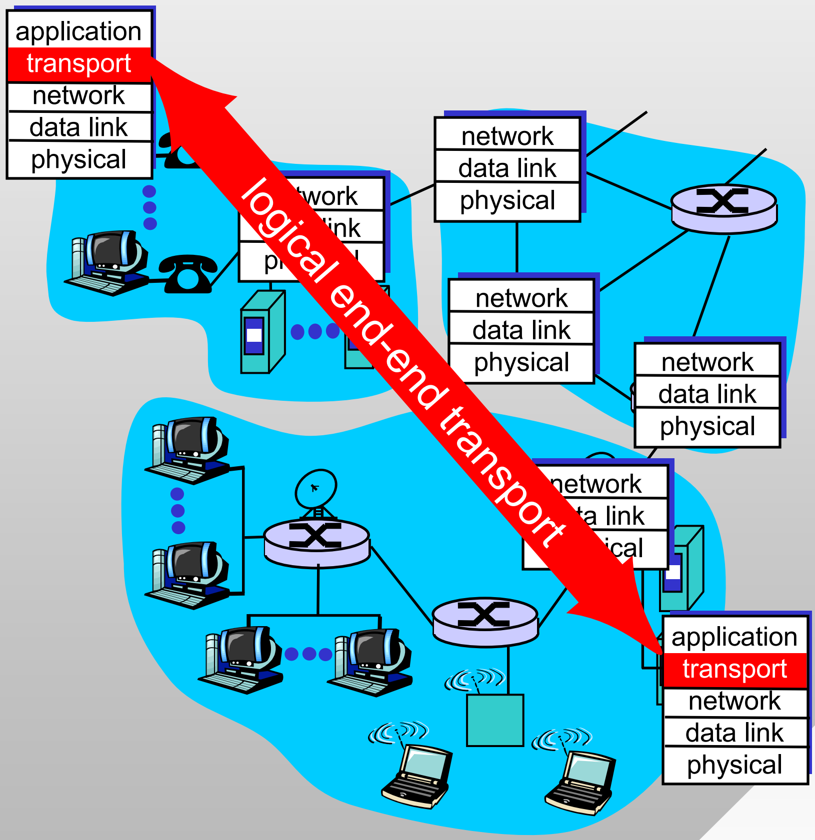
Transport Services and Protocols

- *Transport layer:* logical communication between processes on different hosts
  - Relies on and enhances network-layer services
- *Network layer:* logical communication between hosts
  - Consists of one protocol – IP



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 or loss guarantees
  - Bandwidth guarantees



Chapter 3: Roadmap

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Multiplexing/Demultiplexing

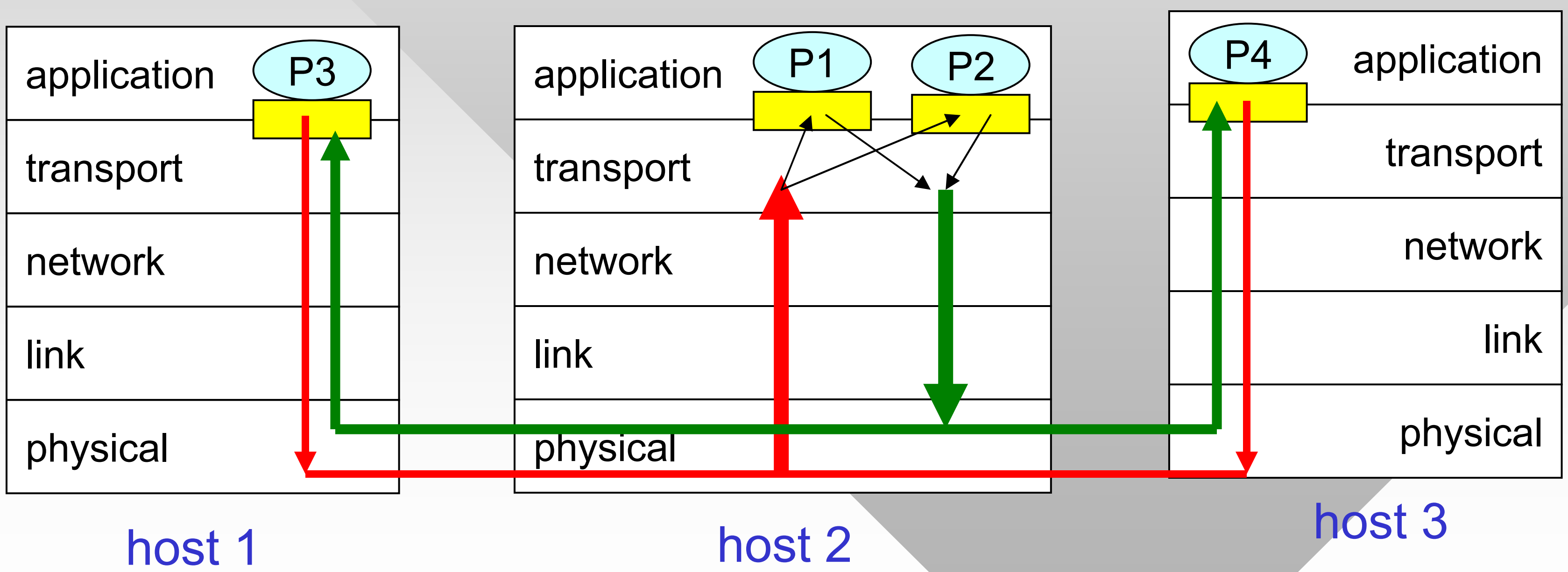
Demultiplexing at receiver host:

Delivering received segments to correct socket

Multiplexing at sender host:

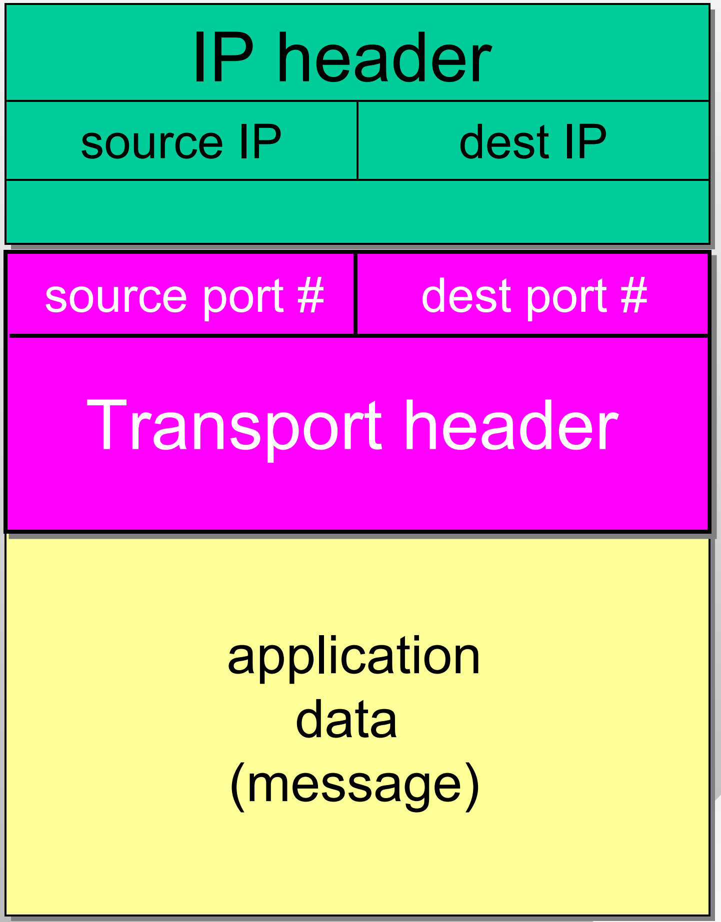
Gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket = process



How Demultiplexing Works

- Host receives IP datagrams
  - Each datagram has source IP address and destination IP address
- Each datagram carries one transport-layer header
  - Transport header starts with source and destination port numbers
- Kernel uses port numbers to direct packets to appropriate socket or reject the message
  - Each port # is a 16-bit unsigned integer (1-65535)



TCP/UDP segment format

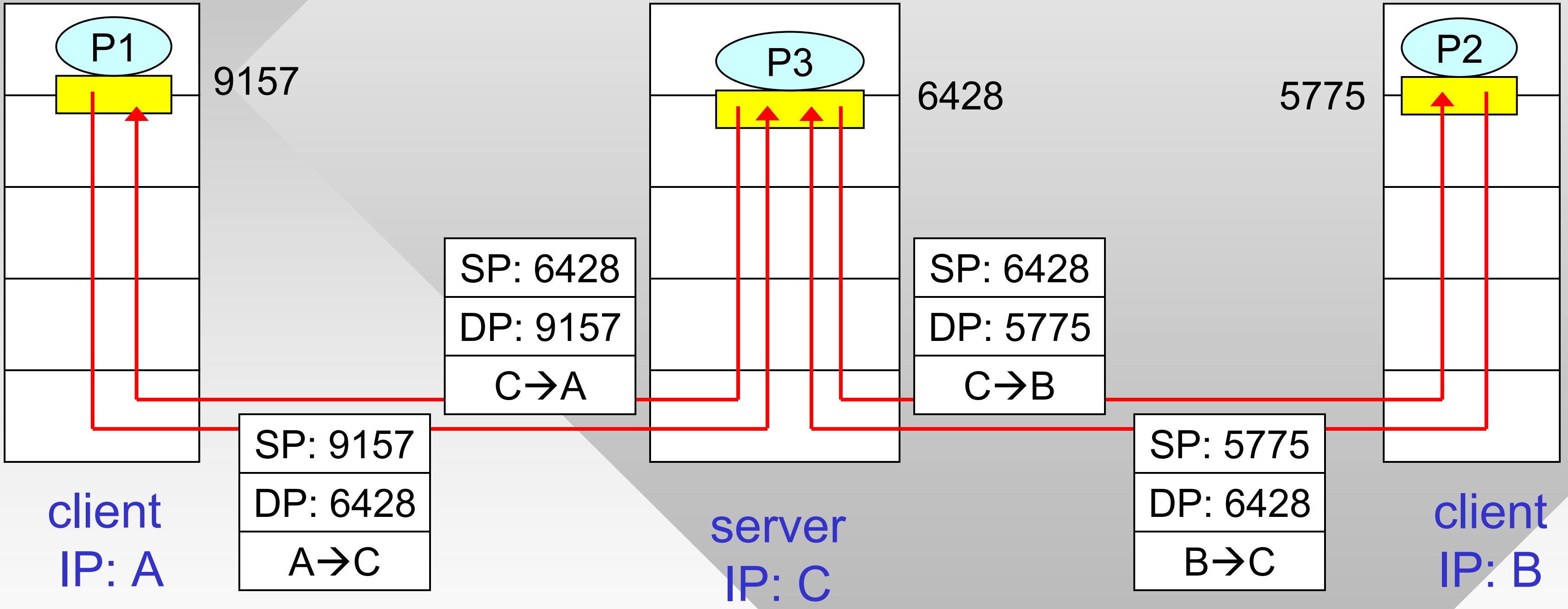
Connectionless Demultiplexing

- Create a SOCK_DGRAM socket
- Bind the socket
  - Server: specify a well-known port (e.g., 53 for DNS)
  - Client: bind to port 0 (OS assigns next available #)
- Use sendto(), recvfrom()
- Target UDP socket is identified by a 2-tuple: (dest IP address, dest port number)

- When host receives UDP segment:
  - OS checks destination port/IP in segment
  - Directs segment to the socket with a matching combination if socket is open; rejects otherwise
- IP datagrams with different source IP addresses and/or source port numbers may be directed to the same socket!

Connectionless Demultiplexing (Cont)

SP = source port, DP = destination port



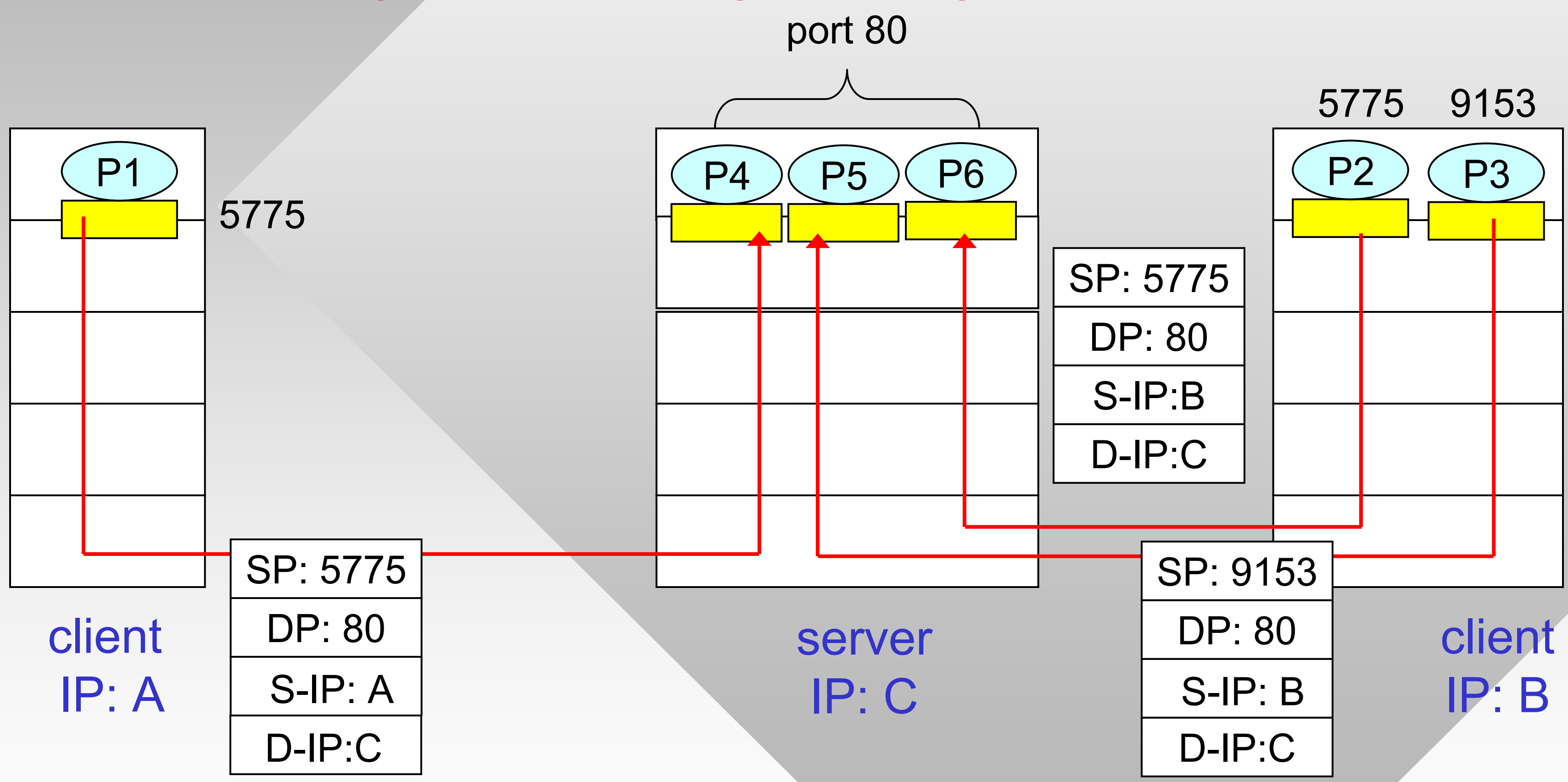
SP provides "return address"

Connection-Oriented Demultiplexing

- TCP socket identified by a 4-tuple:
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number
- Receiver host uses all four values to find appropriate socket
- Clients: each socket must have unique port
- Servers: possible to have multiple TCP sockets with same port number:
  - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - All are on port 80
  - Non-persistent HTTP may have different socket for each request

Connection-Oriented Demultiplexing (Cont)

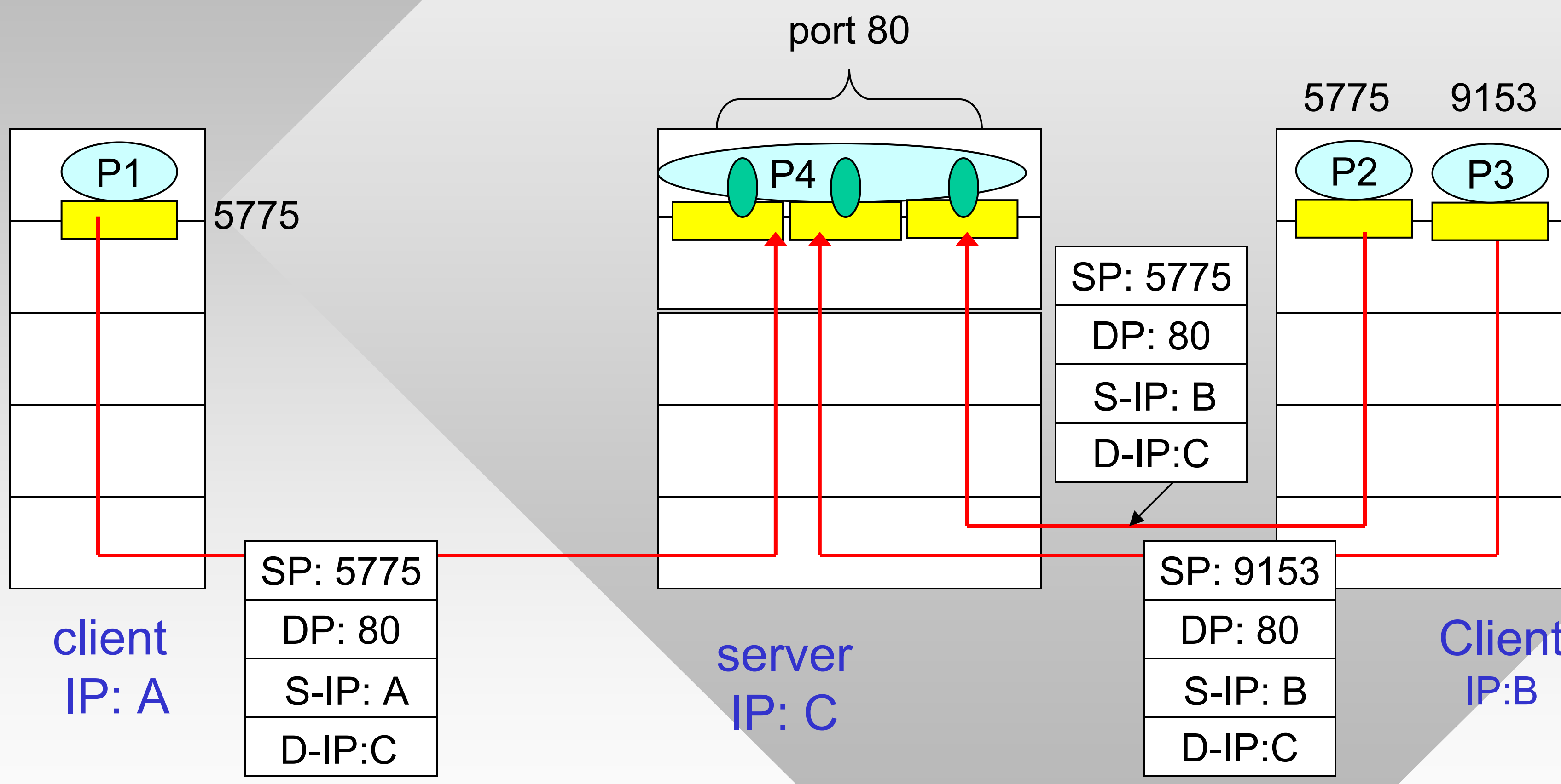
Web server spawns a new process per connection



SP = source port, DP = destination port;
S-IP = source IP, D-IP = destination IP

Connection-Oriented Demultiplexing (Cont)

Web server spawns a new thread per connection



SP = source port, DP = destination port;
S-IP = source IP, D-IP = destination IP

Chapter 3: Roadmap

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control
3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

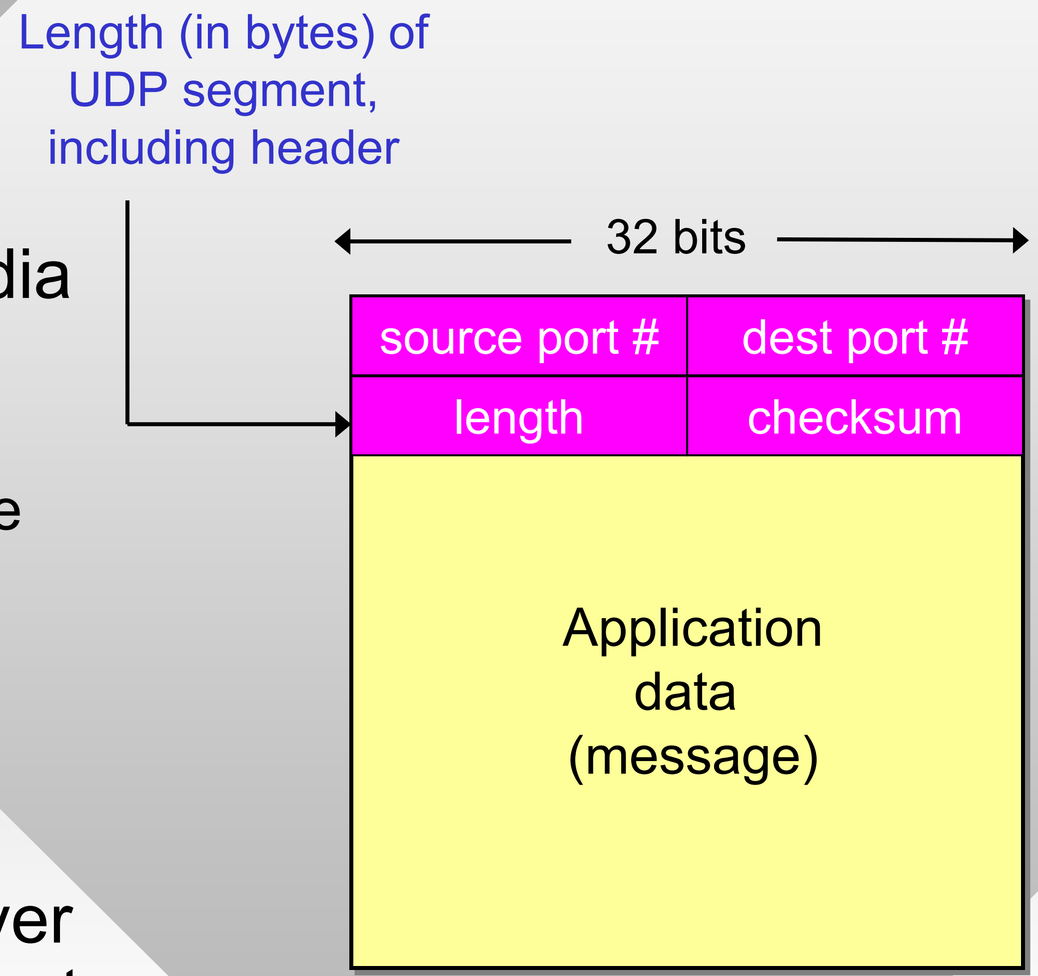
- Standardized in 1980
  - Hasn't changed since
- Best-effort service
- UDP segments may be:
  - Lost or corrupted
  - Delivered out of order to the application
- Connectionless:
  - No handshaking between UDP sender and receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- Low overhead: no connection establishment or retransmission
- Simplicity: no connection state at sender/receiver
- Small segment header
- No congestion control
  - For short transfers, this is completely unnecessary
  - In other cases, desirable to control rate directly from application

UDP: More

- Often used for streaming multimedia or online gaming
  - Loss tolerant
  - Rate/delay sensitive
- Other UDP uses
  - DNS
  - SNMP
  - NFSv2 (1989)
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recovery



UDP segment format

UDP Checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment (packet)

Sender (simplified):

- Set checksum = 0 in hdr
- Treat packet contents as a sequence of 16-bit integers (padded with 0s to 2-byte boundary)
- Checksum: add all integers, then XOR with 0xffff
- Sender puts checksum value into UDP checksum field

Receiver:

- Sum all 16-bit words in entire received segment (including the checksum field in the header)
- Check if result = 0xffff
  - NO - error detected
  - YES - no error detected
- Idea: (x XOR 0xffff) + x = 0xffff
- *Are undetected errors possible nonetheless?*

UDP Checksum Example

- Note on 1's complement addition:
  - 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
```

UDP Checksum (Cont)

- How many corrupted bits does UDP detect?
- Example of undetected single-bit corruption?
  - Not possible
- Example of undetected 2-bit corruption?
  - Two words (0, 5) result in sum = 5
  - Suppose 0 is corrupted to become 1 and 5 is corrupted to become 4, then the checksum is the same
- Example of undetected 3-bit corruption w/two words?
  - Two words (1, 1) → (0, 2)
- What if the transmitted words are 0 and 12?
  - Can two-bit corruption produce the same checksum?
  - If yes, how many ways can (0,12) be affected by 2-bit corruption so as to avoid detection?

Wrap-up

- Is there a pair of integers (x,y) that allow the UDP checksum to detect any 2-bit corruption?
- Data-link and physical layers are often assumed to have their own checksums and error correction
  - Why is transport-level checksum important then?
- Reasons:

1) Lower layers do not always run error checking
   - Even then, implementation bugs may affect the result
2) Corruption may occur in router RAM or faulty hardware, outside the control of data-link protocols