



19CSE301 COMPUTER NETWORKS

Amrita Vishwa Vidyapeetham
Amritapuri Campus



TRANSPORT LAYER



Chapter 3: Transport Layer

All material copyright 1996-2016

J.F Kurose and K.W. Ross, All Rights Reserved

Chapter 3: Transport Layer

Transport-layer protocols are implemented in the end systems but not in network routers.

On the sending side, the transport layer converts the application layer messages it receives from a sending application process into transport-layer packets, known as transport-layer segments in Internet terminology. This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment. The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a data gram) and sent to the destination

Transport Layer

our goals:

- understand principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- learn about Internet transport layer protocols:
 - UDP: connectionless transport
 - TCP: connection-oriented reliable transport
 - TCP congestion control

Chapter 3 outline

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

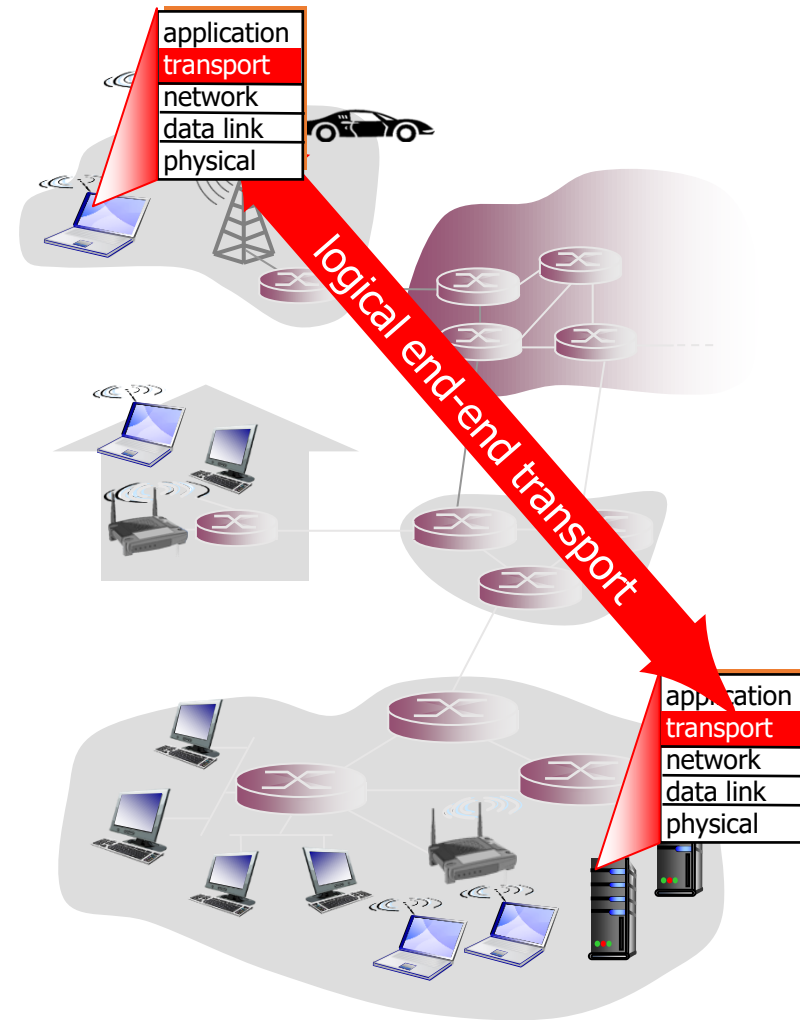
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion
control

3.7 TCP congestion control

Transport services and protocols

- 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

If the network-layer protocol cannot provide delay or bandwidth guarantees for transport layer segments sent between hosts, then the transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes.

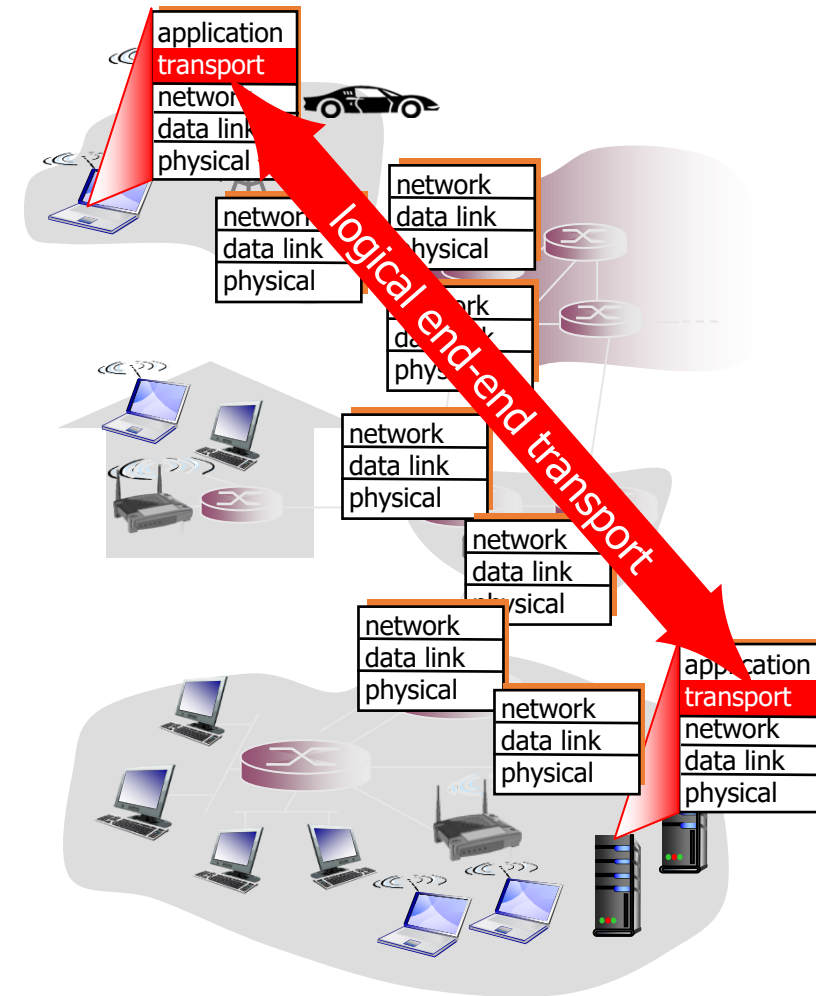
12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

A transport protocol can offer reliable data transfer service to an application even when the underlying network protocol is unreliable, eg: transport protocol can use encryption to guarantee that application messages are not read by intruders, even when the network layer cannot guarantee the confidentiality of transport-layer segments.

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



Chapter 3 outline

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-layer Multiplexing/demultiplexing

The most fundamental responsibility of UDP and TCP is to extend IP's delivery service between two end systems to a delivery service between two processes running on the end systems. Extending host-to-host delivery to process-to-process delivery is called transport-layer multiplexing and demultiplexing.

At the receiving end, the transport layer examines the fields in the segment corresponding to appropriate sockets to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called **demultiplexing**. The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called **multiplexing**.

Port Address

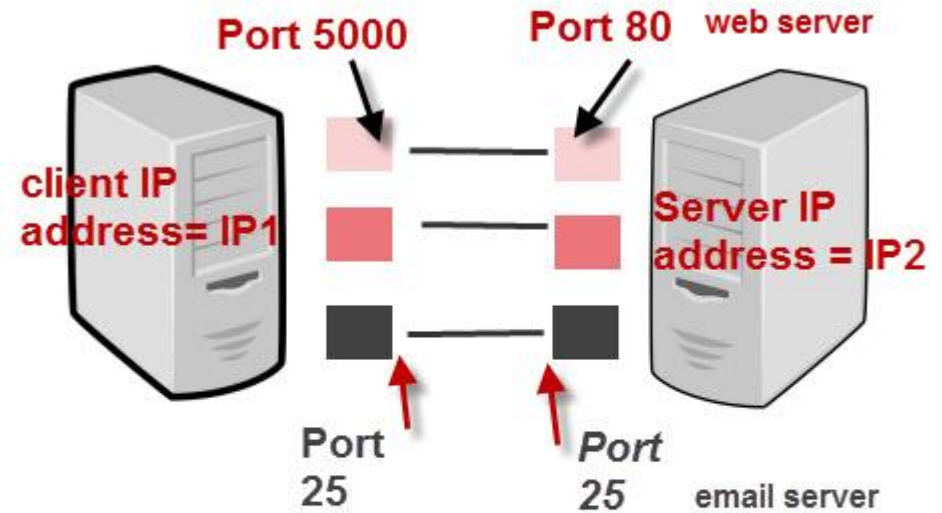
- The IP address identifies the device e.g. computer.
- However an IP address alone is not sufficient for running network applications, as a computer can run multiple applications and/or services.
- Just as the IP address identifies the computer, The network port identifies the application or service running on the computer.
- The use of ports allow computers/devices to run multiple services/applications.

Client port numbers are dynamically assigned, and can be reused once the session is closed.

Ports and Sockets

0 - 65535

- **Port Number Ranges and Well Known Ports:** A port number uses 16 bits and so can therefore have a value from **0** to **65535** decimal
- **Sockets** :A connection between two computers uses a socket.
 - A socket is a **software interface** between the application layer and the transport layer, Using a metaphor, the process is analogous to a house and a **socket** to its **door**.
 - A socket is the combination of IP address plus port



IP Address + Port number = Socket

TCP/IP Ports And Sockets

Analogy

If you use a house or apartment block analogy the IP address corresponds to the street address.

All of the apartments share the same street address.

However each apartment also has an apartment number which corresponds to the Port number.

Types of Port numbers

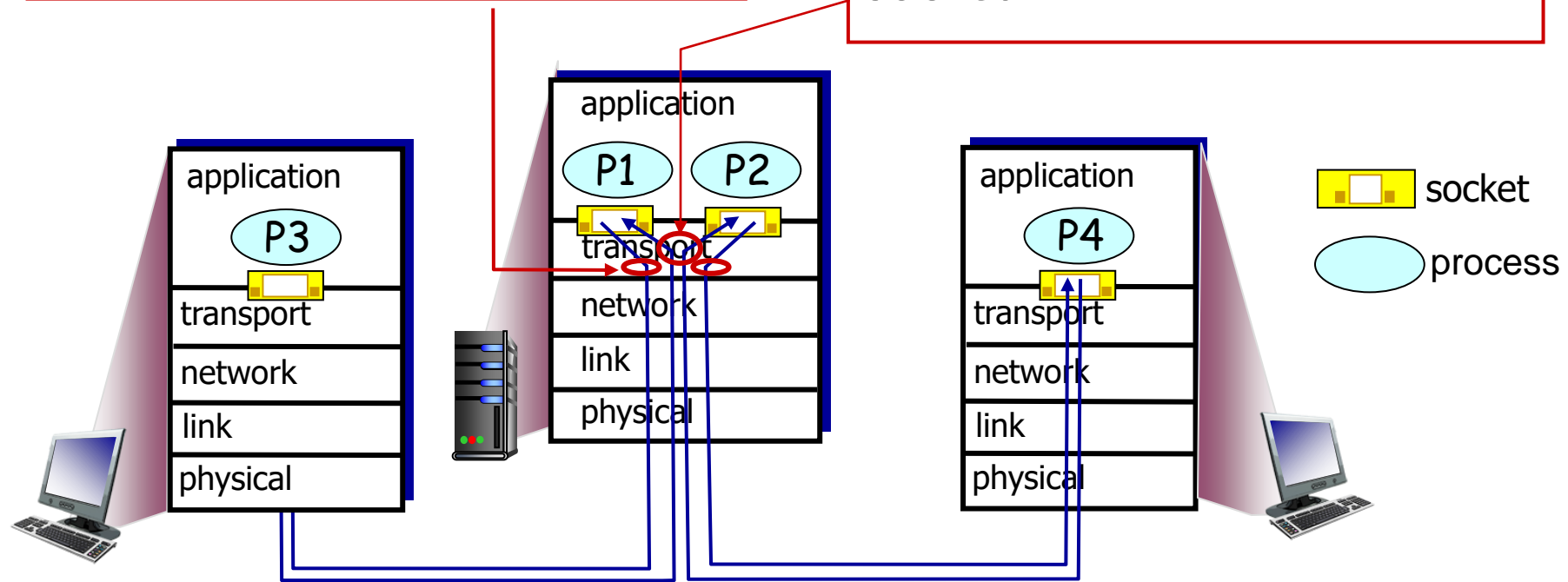


- Port numbers are divided into ranges as follows:
- **Port numbers 0-1023 – Well known ports.** These are allocated to **server services** by the **Internet Assigned Numbers Authority (IANA)**. e.g Web servers normally use **port 80** and SMTP servers use **port 25** (see diagram above).
- **Ports 1024-49151- Registered Port** -These can be registered for services with the **IANA** and should be treated as **semi-reserved**. Registered port numbers are non-well-known ports that are used by vendors for their own server applications.
- **Ports 49152-65535**– These are used by **client programs** and you are free to use these in client programs. When a Web browser connects to a web server the browser will allocate itself a port in this range. Also known as **ephemeral ports**.

Multiplexing/demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

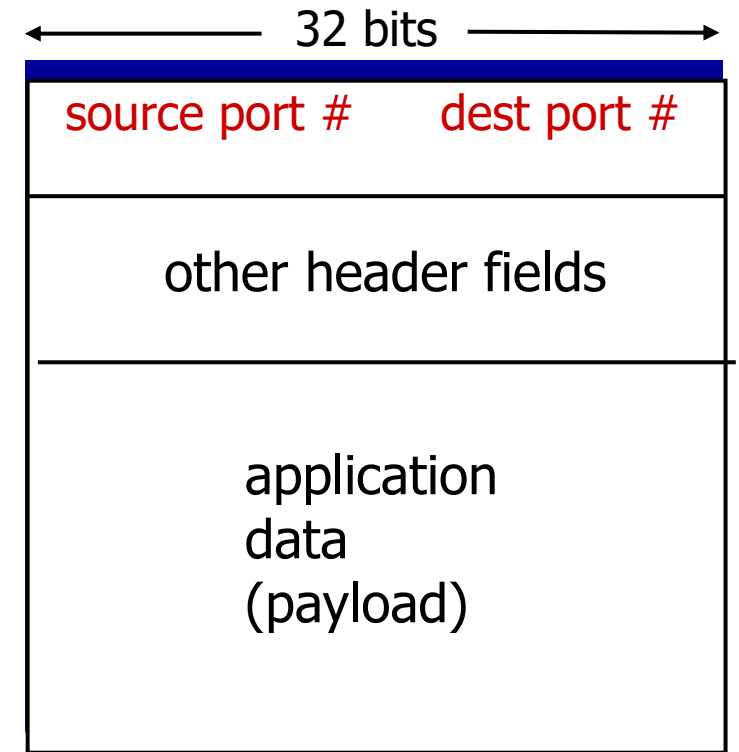
demultiplexing at receiver:
use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
 - each datagram has source IP address, destination IP address
 - each datagram carries one transport-layer segment
 - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket

Transport-layer multiplexing requires (1) that sockets have unique identifiers, and (2) that each segment have special fields that indicate the socket to which the segment is to be delivered. These special fields are the **source port number field** and the **destination port number field**.



TCP/UDP segment format

Connectionless demultiplexing

- *recall*: created socket has host-local port #:

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

- *recall*: when creating datagram to send into UDP socket, must specify

- destination IP address
- destination port #

-
- when host receives UDP segment:

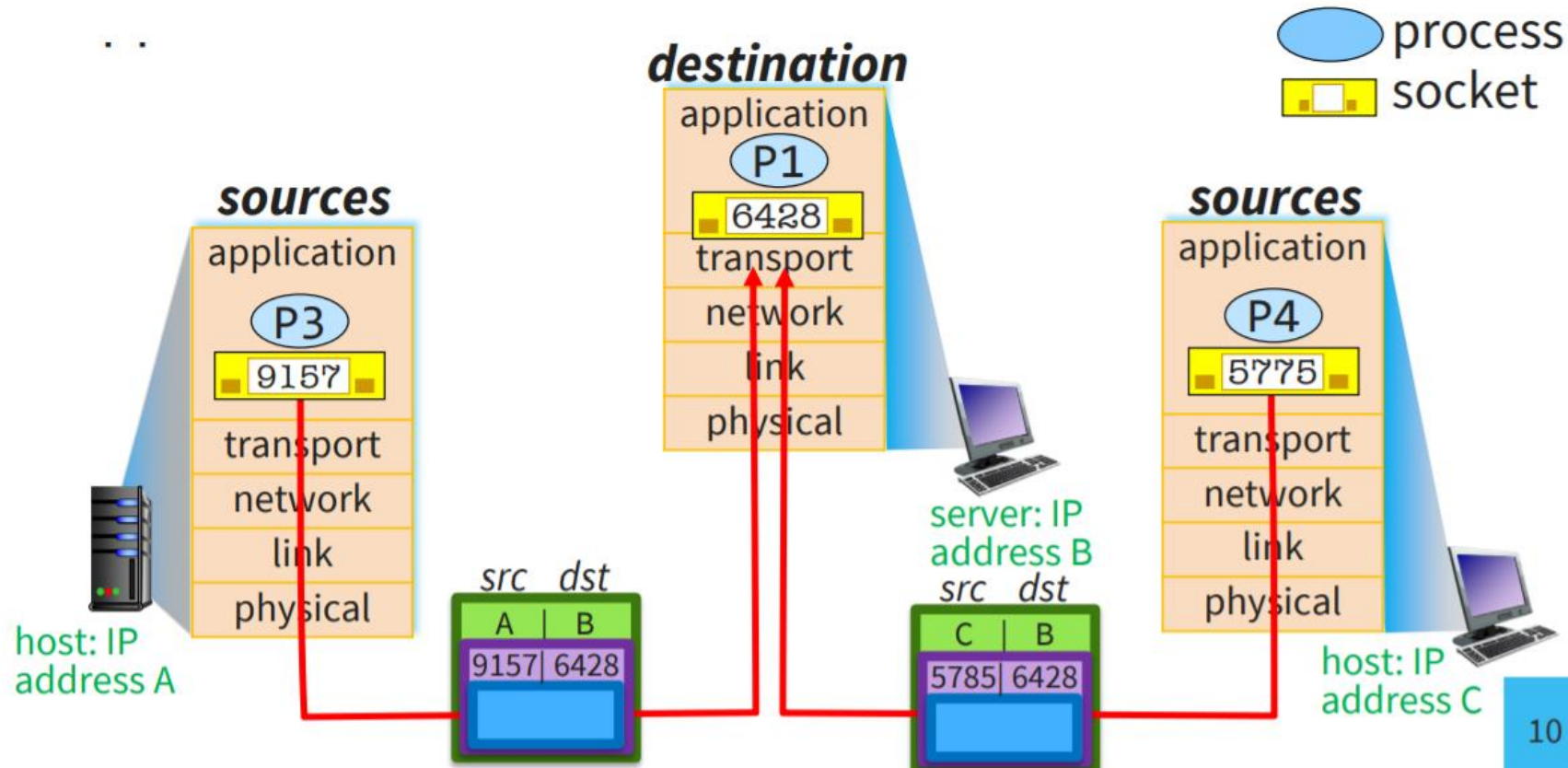
- checks destination port # in segment
- directs UDP segment to socket with that port #



IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

- Host receives 2 UDP segments: • checks dst port, directs segment to socket w/that port • different src IP or port but same dst port → same socket • application must sort it out



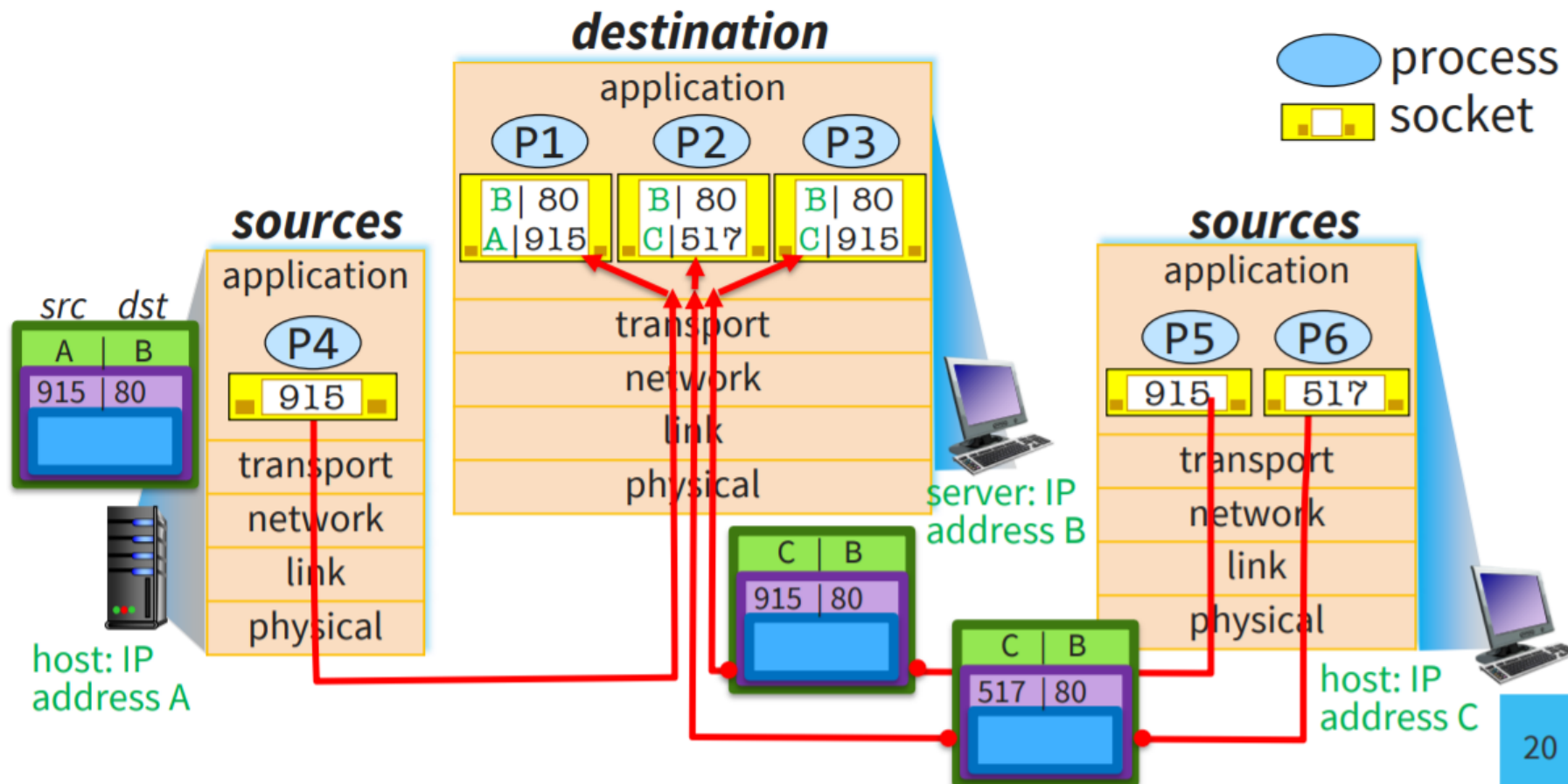
Connection-oriented demux

- TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- demux: receiver 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: example

- Host receives 3 TCP segments: • all destined to IP addr B, port 80 • demuxed to different sockets with socket's 4-tuple



Chapter 3 outline

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]

If the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP of network layer. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process.

UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be **connectionless**.

DNS is an example of an application-layer protocol that typically uses UDP. When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to UDP. Without performing any handshaking with the UDP entity running on the destination end system, the host-side UDP adds header fields to the message and passes the resulting segment to the network layer.

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
- UDP use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
 - reliable transfer over UDP:
 - add reliability at application layer
 - application-specific error recovery!

Isn't TCP always preferable, since TCP provides a reliable data transfer service, while UDP does not?

- The answer is no, as many applications are better suited for UDP for the following reasons:
- *Finer application-level control over what data is sent, and when* : Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs. (EG: streaming multimedia)
- *No connection establishment*. TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text
- *No connection state*. TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.
- *Small packet header overhead*. The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

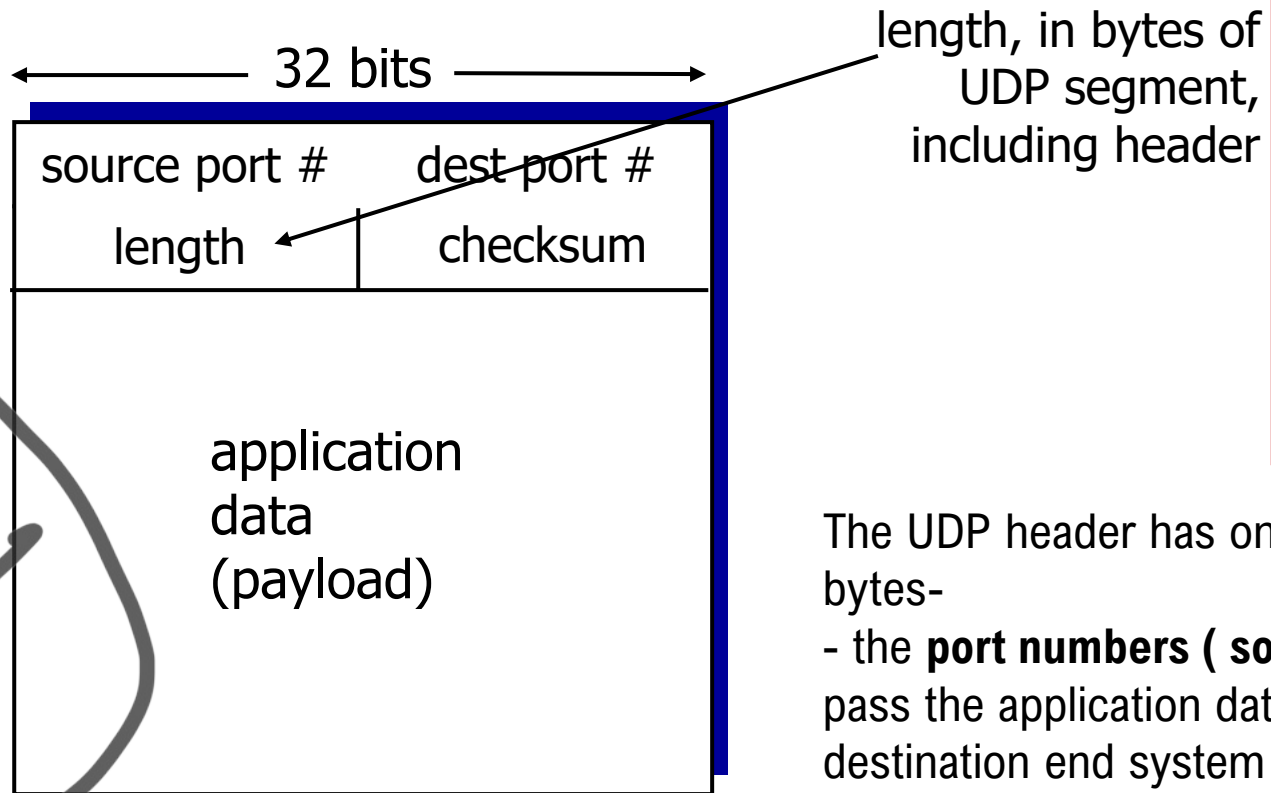
| Application | Application-Layer Protocol | Underlying Transport Protocol |
|------------------------|----------------------------|-------------------------------|
| Electronic mail | SMTP | TCP |
| Remote terminal access | Telnet | TCP |
| Web | HTTP | TCP |
| File transfer | FTP | TCP |
| Remote file 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 |

Figure 3.6 ♦ Popular Internet applications and their underlying transport protocols

UDP: segment header

why is there a UDP?

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



UDP segment format

The UDP header has only four fields, each consisting of two bytes-

- the **port numbers (source, destn)** allow the destination host to pass the application data to the correct process running on the destination end system
- The **length field** specifies the number of bytes in the UDP segment (header plus data)
- **checksum** is used by the receiving host to check whether errors have been introduced into the segment

UDP checksum

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field


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

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

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 |
| <hr/> | | | | | | | | | | | | | | | | |
| wraparound | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| |  | | | | | | | | | | | | | | | |
| checksum | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

At the receiver, all four 16-bit words are added, including the checksum.

If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111.

If one of the bits is a 0, then we know that errors have been introduced into the packet.

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Chapter 3 outline

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