

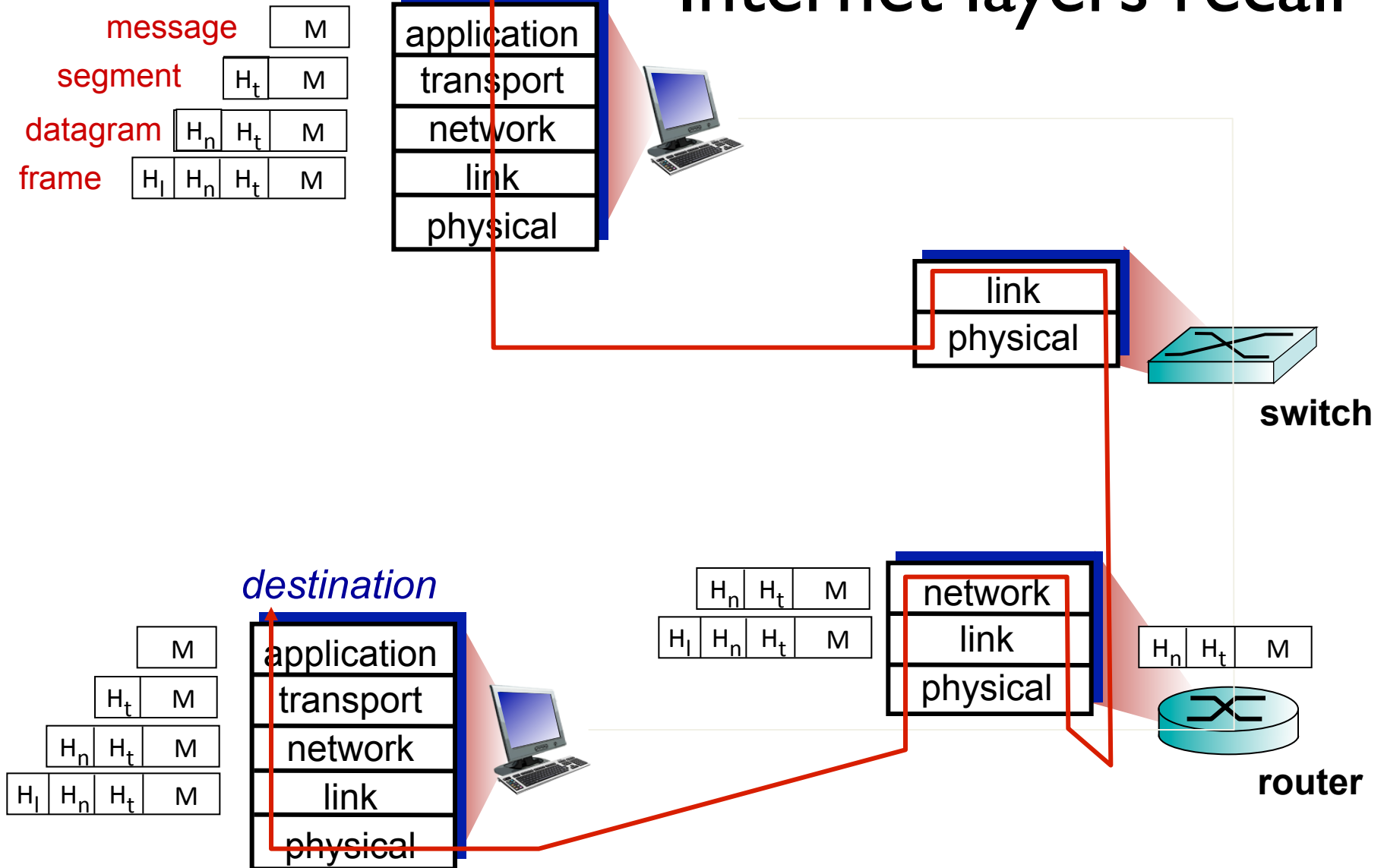
CS450 – Introduction to Networking

Lecture 11 – Transport Layer and UDP

Phu Phung

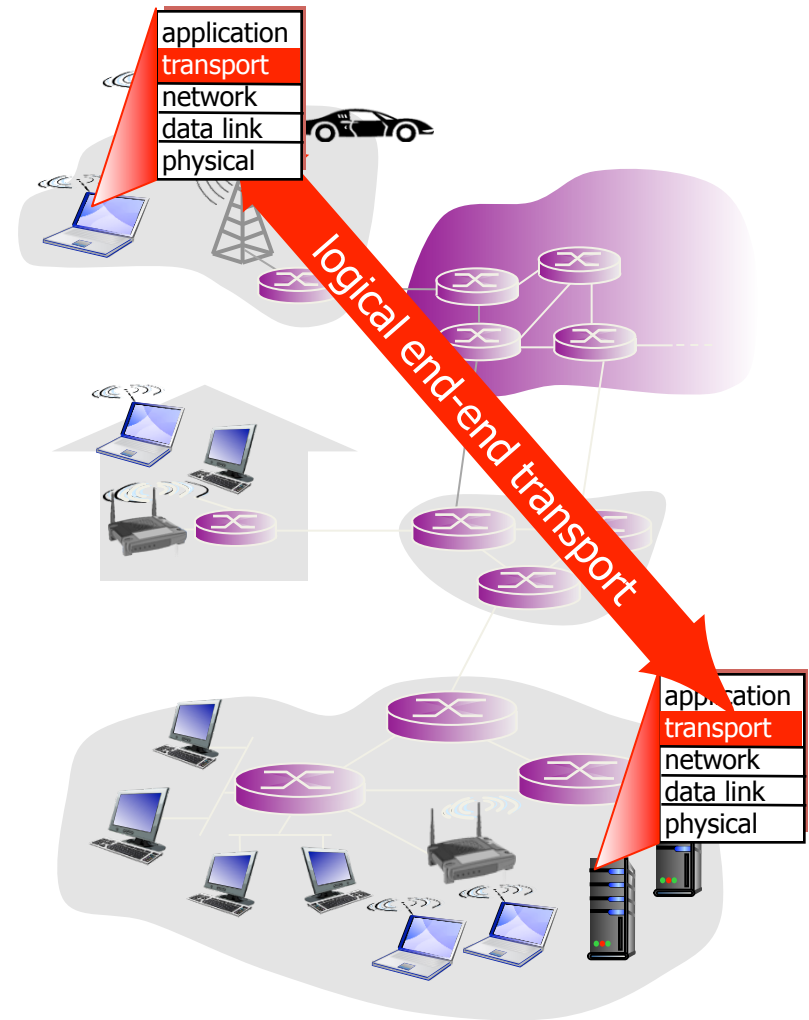
Feb 6, 2015

Internet layers recall



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

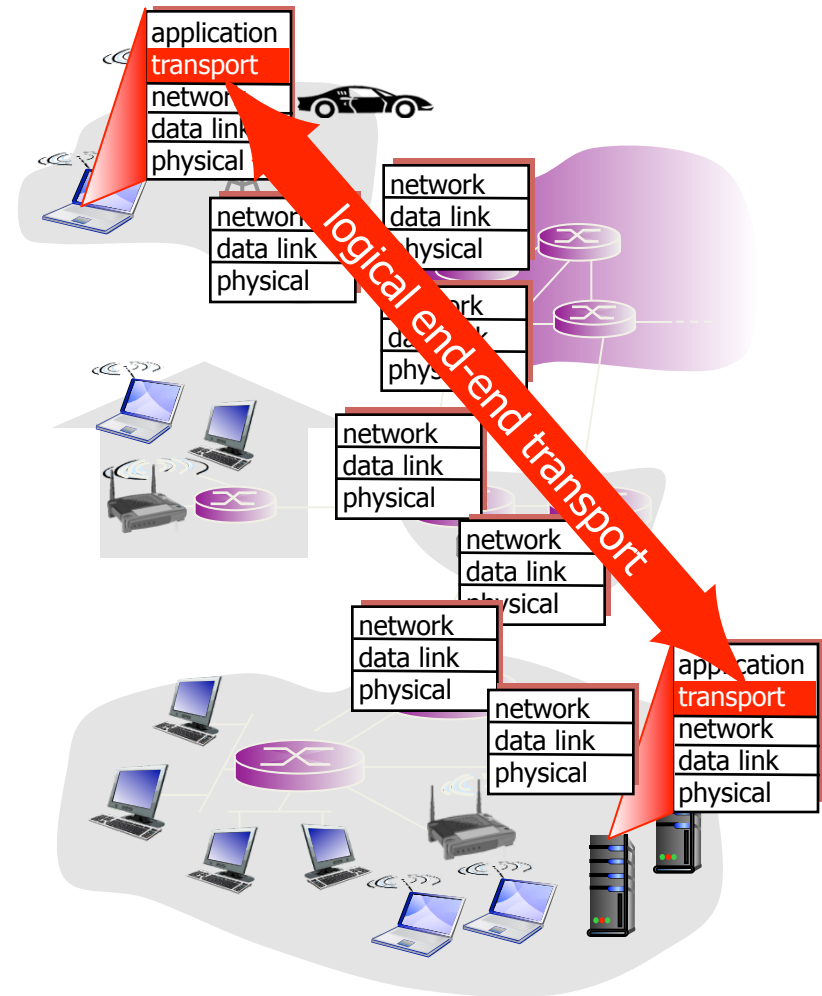
household analogy:

12 kids 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

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



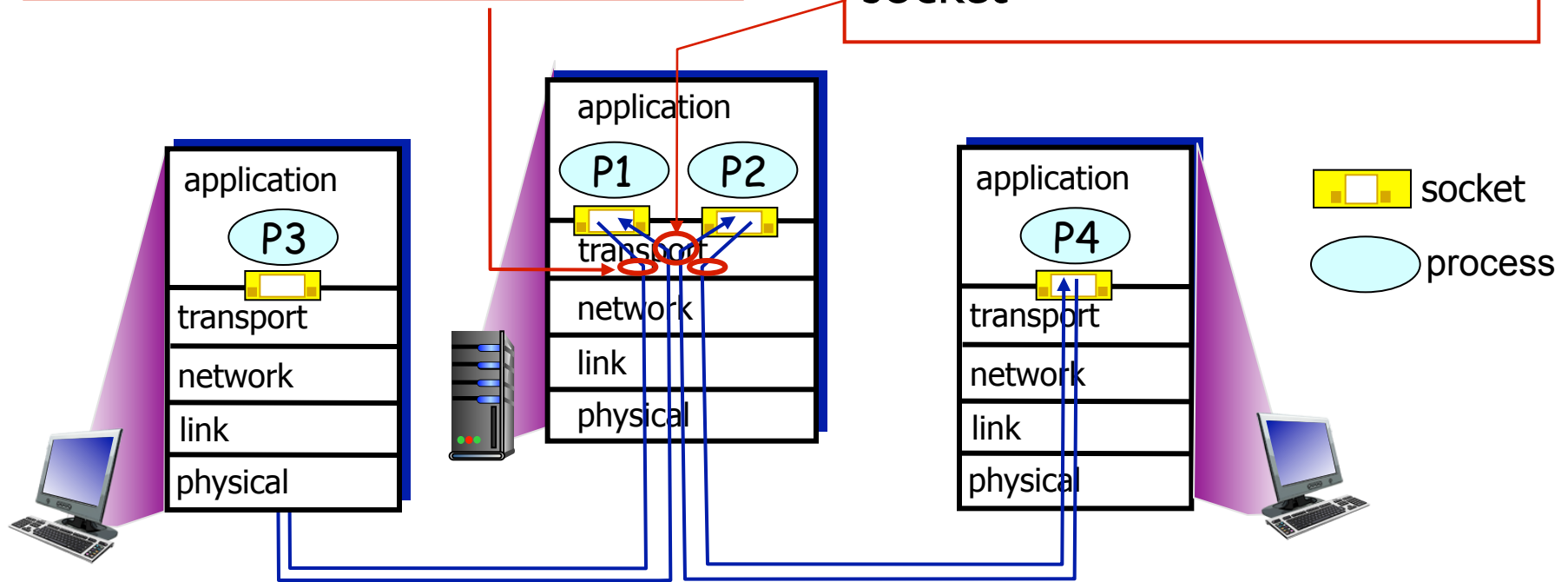
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

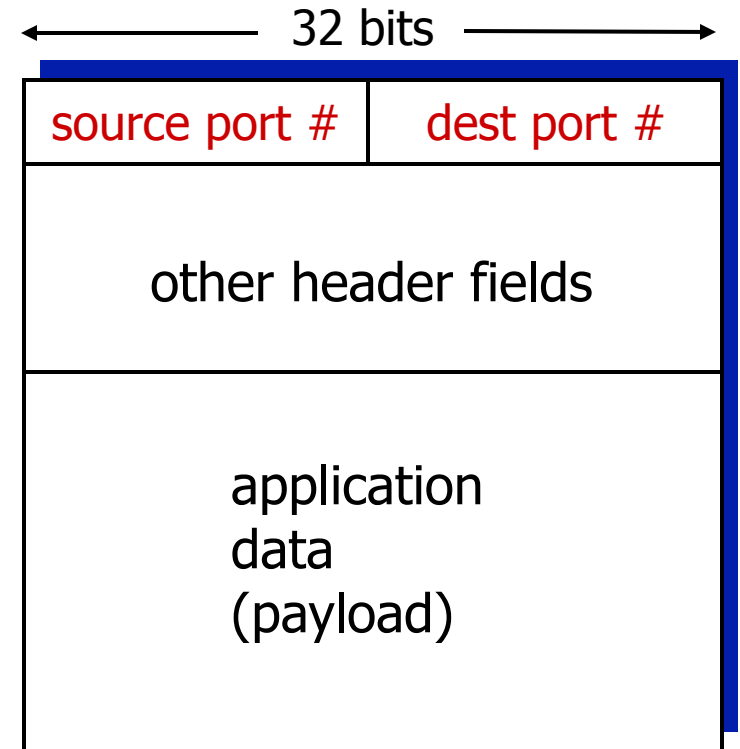
demultiplexing at receiver:

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



TCP/UDP segment format

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;
```

```
class UDPClient {
    public static void main(String args[]) throws Exception
    {
```

create
input stream

```
    → BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));
```

create
client socket

```
    → DatagramSocket clientSocket = new DatagramSocket();
```

translate
hostname to IP
addr using DNS

```
    → InetAddress IPAddress = InetAddress.getByName("hostname");
```

```
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];
```

```
    String sentence = inFromUser.readLine();
```

```
    sendData = sentence.getBytes();
```


Example: Java client (UDP)

create datagram with
data-to-send,
length, IP addr, port

→ `DatagramPacket sendPacket =
 new DatagramPacket(sendData, sendData.length,
 IPAddress, 9876);`

send datagram
to server

→ `clientSocket.send(sendPacket);`

`DatagramPacket receivePacket =
 new DatagramPacket(receiveData, receiveData.length);`

read datagram
from server

→ `clientSocket.receive(receivePacket);`

`String modifiedSentence =
 new String(receivePacket.getData());`

`System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}`

`}`

Example: Java server (UDP)

```
import java.io.*;  
import java.net.*;
```

```
class UDPServer {  
    public static void main(String args[]) throws Exception  
    {
```

create
datagram socket
at port 9876



```
DatagramSocket serverSocket = new DatagramSocket(9876);
```

```
byte[] receiveData = new byte[1024];  
byte[] sendData = new byte[1024];
```

```
while(true)  
{
```

create space for
received datagram



```
DatagramPacket receivePacket =  
    new DatagramPacket(receiveData, receiveData.length);
```

receive
datagram



```
serverSocket.receive(receivePacket);
```

Example: Java server (UDP)

```
String sentence = new String(receivePacket.getData());
```

get IP addr
port #, of
sender

```
→ InetAddress IPAddress = receivePacket.getAddress();  
→ int port = receivePacket.getPort();
```

```
String capitalizedSentence = sentence.toUpperCase();
```

```
sendData = capitalizedSentence.getBytes();
```

create datagram
to send to client


```
→ DatagramPacket sendPacket =  
    new DatagramPacket(sendData, sendData.length, IPAddress,  
        port);
```

write out
datagram
to socket


```
→ serverSocket.send(sendPacket);  
    }  
    }  
}
```

end of while loop,
loop back and wait for
another datagram

Connectionless demultiplexing

- *recall*: created socket has host  *recall*: when creating datagram to send into UDP socket, must specify local port #:

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

 - destination IP address
 - destination port #
 - when host receives UDP segment:
 - checks destination port # 
 - 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

```
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157);
```

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

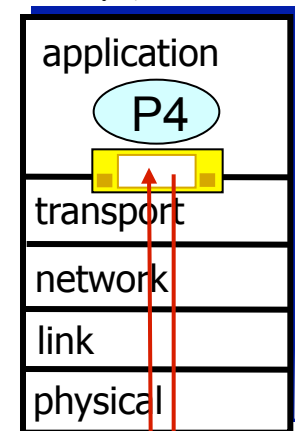
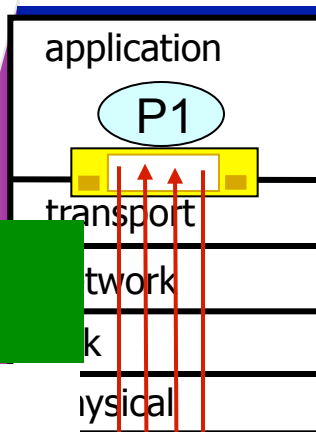
```
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775);
```

SP, DP =

A. 6428, 5775

B. 5775, 6428

source port: 9157
dest port: 6428



source port: SP?
dest port: DP?

source port: ?
dest port: ?

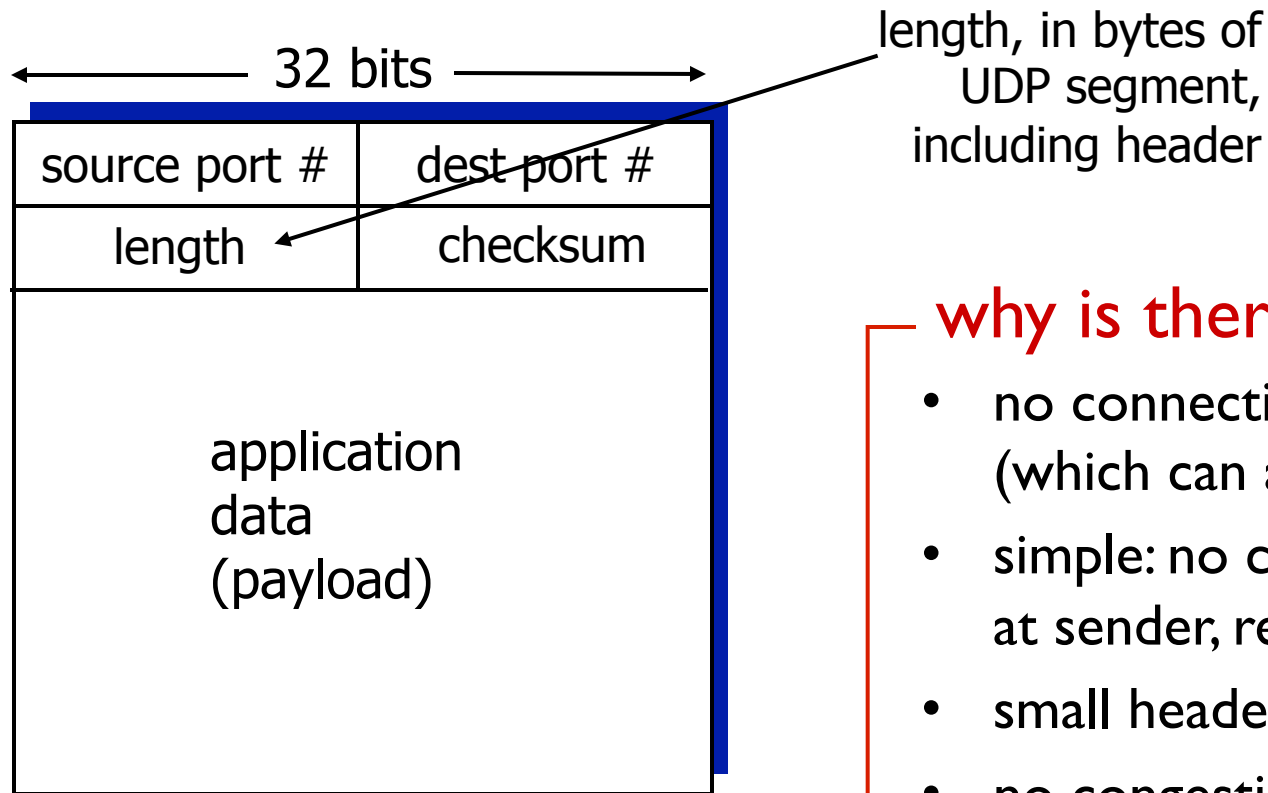
Select a wrong statement

- A. Both TCP and UDP need connection setup
- B. Delivery of UDP packets is unordered
- C. TCP provides reliable transport protocol
- D. TCP provides congestion control while UDP does not
- E. Neither TCP nor UDP has delay guarantees

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!

UDP: segment header



UDP segment format

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 checksum

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

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

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
<hr/>																
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

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

Let's assume a sender sent a UDP segment with (checksum, data) = (0110, 0101); and the receiver received (0100, 0101). The receiver can conclude

- A. The segment is corrupted
- B. The checksum is wrong
- C. The data is correct
- D. B and C

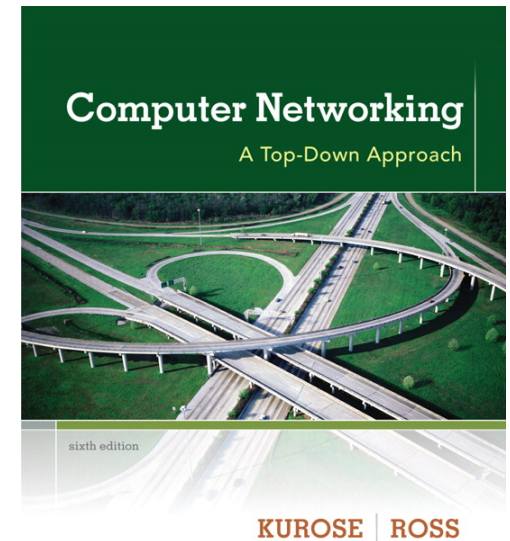
Next lecture

- Reliable Data Transfer
 - Readings 3.4
- Midterm exam in 4 weeks
 - In class: 1 PM Friday, March 6th

Copy right notice:

These slides are adapted from J.F Kurose and K.W. Ross's ones

© All material copyright 1996-2012
J.F Kurose and K.W. Ross, All Rights Reserved



Computer Networking: A Top Down Approach

6th edition

Jim Kurose, Keith Ross

Addison-Wesley

March 2012