The Transport Layer: De/Multiplexing, Reliability

CS 352, Lecture 6, Spring 2020

http://www.cs.rutgers.edu/~sn624/352

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Course announcements

- Quiz 2 will go online later today
  - Submit on Sakai by Tuesday 10 pm
  - 30-minute time limit

- Project 1 will be released later today
  - Check the Sakai resources section for a tarfile with instructions

Review of concepts

- Application-layer protocols: DNS, HTTP, SMTP
- HTTP caching: why?
- Content distribution networks (CDNs):
  - Origin server, CDN DNS server, CDN cache servers, client
  - Use indirection
- Simple Mail Transfer Protocol (SMTP): req/resp protocol
  - User agents, sending mail server, receiving mail server
  - SMTP commands, mail headers, response codes
  - Mail access protocols: POP, IMAP, HTTP
    - Support pull rather than push

POP vs IMAP

- POP3
- Stateless server
- UA-heavy processing
- UA retrieves email from server, then typically deleted from server
- Latest changes are at the UA
- Simple protocol (list, retr, del within a POP session)

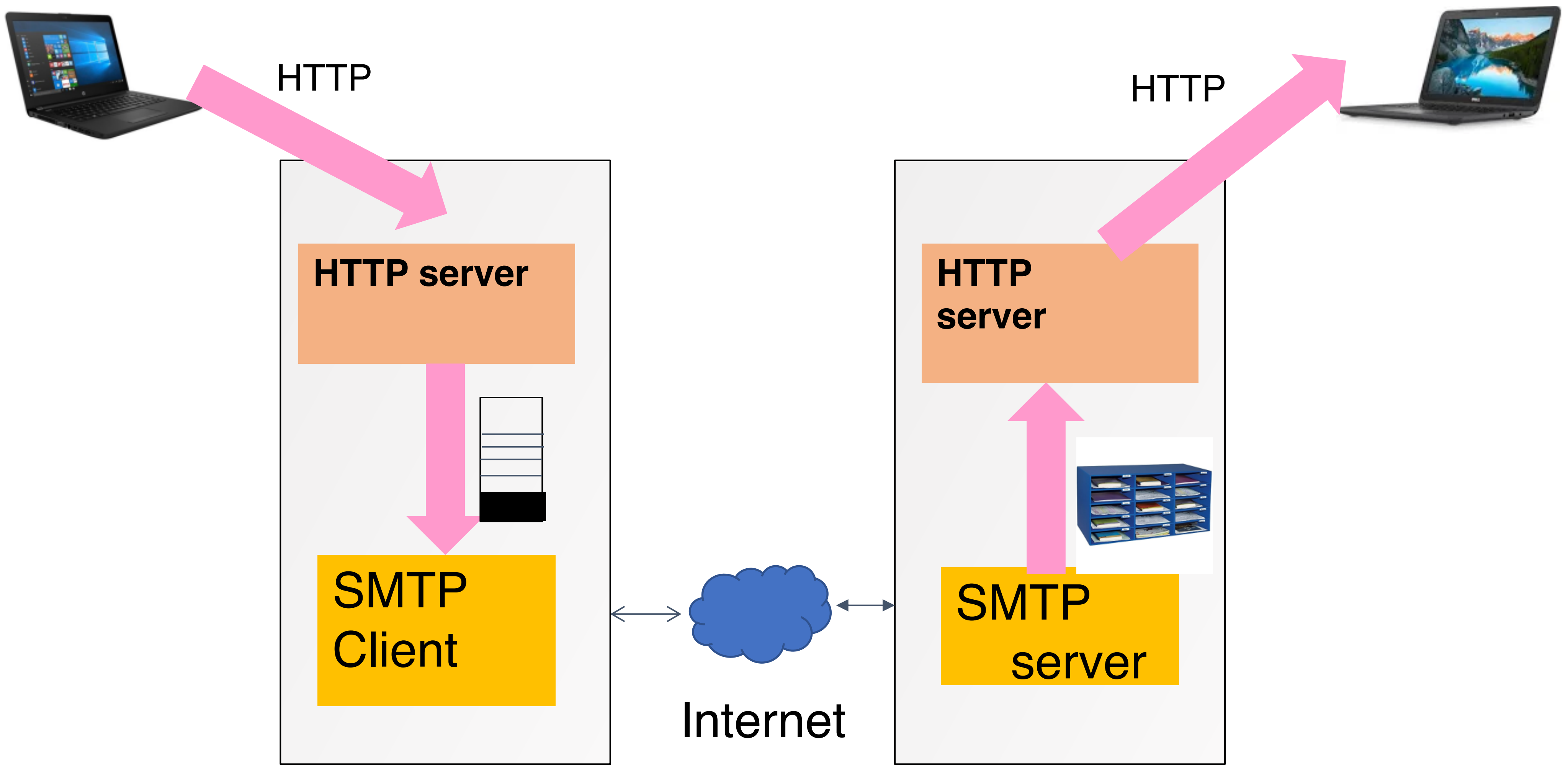
- IMAP4
- Stateful server
- UA and server processing
- Server sees folders, etc. which are visible to UAs
- Changes visible at the server
- Complex protocol

What about web-based email?

- Connect to mail servers via web browser
  - Ex: gmail, outlook, etc.

- Browsers speak HTTP
- Email servers speak SMTP
- Need a bridge to retrieve email using HTTP

Web based email

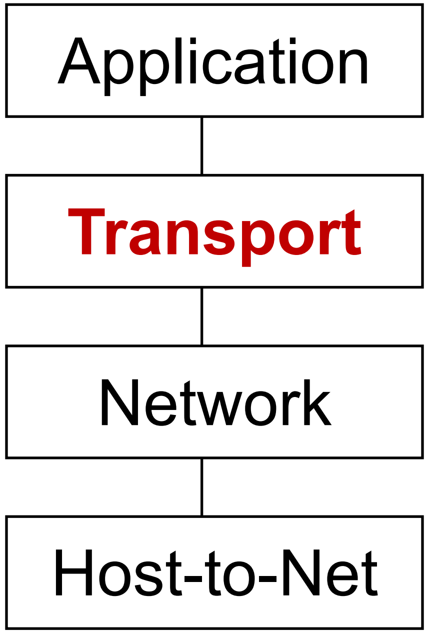


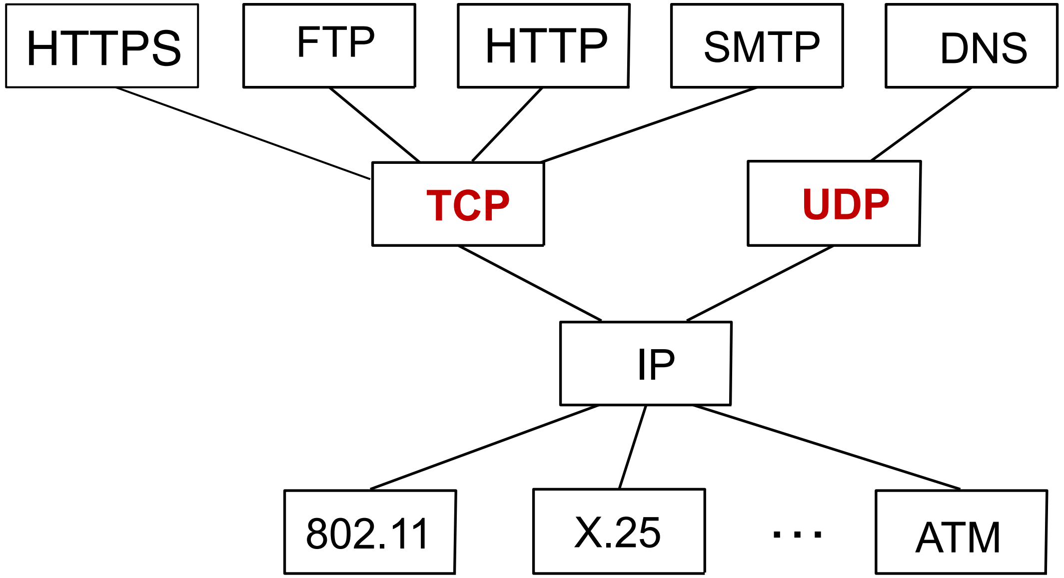
More themes from app-layer protocols

- Separation of concerns. Examples:
  - Content rendering for users (browser, UA) separate from protocol operations (mail server)
  - Reliable mail sending and receiving: mail UA doesn't need to be "always on" to send or receive email reliably
- In-band vs. out-of-band control:
  - In-band: headers determine the actions of all the parties of the protocol
  - There are protocols with out-of-band control, e.g., FTP
- Keep it simple until you really need complexity
  - ASCII-based design; stateless servers. Then introduce:
  - Cookies for HTTP state
  - IMAP for email organization
  - Security extensions
  - Different methods to set up and use underlying connections, etc.

The Transport Layer

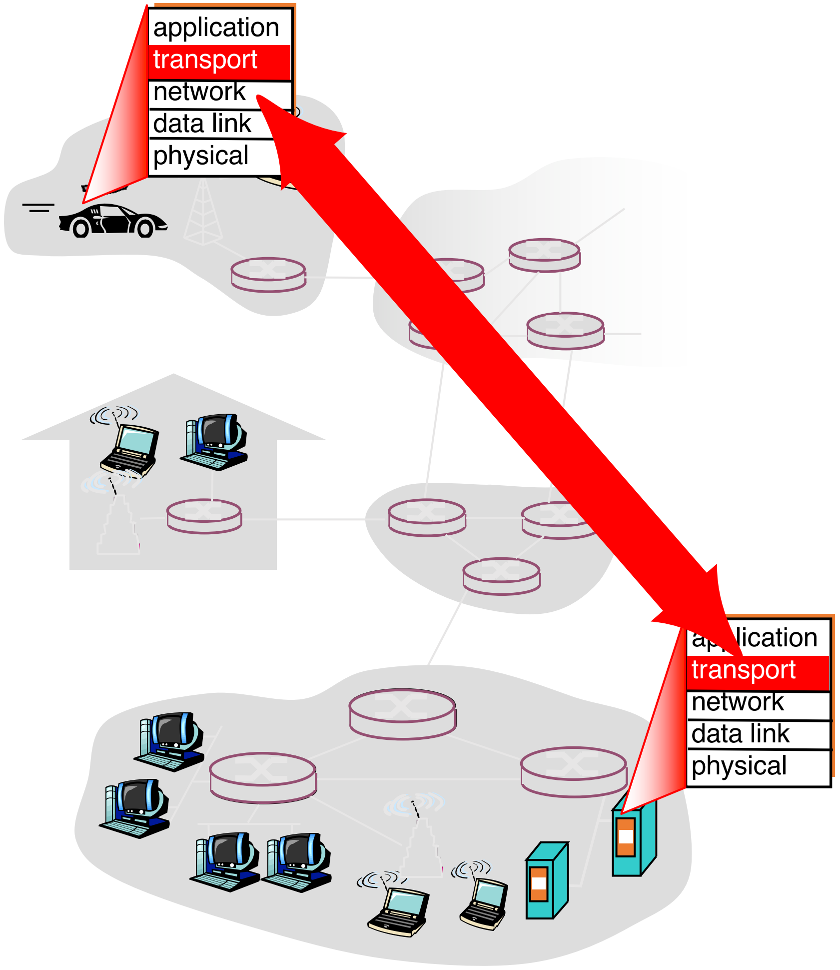
Transport


Application
Transport
Network
Host-to-Net


HTTPS
FTP
HTTP
SMTP
DNS
TCP
UDP
IP
802.11
X.25
...
ATM

Transport services and protocols

- Provide logical communication between app processes running on different hosts
- Transport protocols run @ hosts
  - send side: breaks app messages into segments, passes to network layer
  - recv 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 endpoints

- Transport layer: logical communication between processes
  - relies on and enhances network layer services

Household analogy:

12 kids sending letters to 12 kids

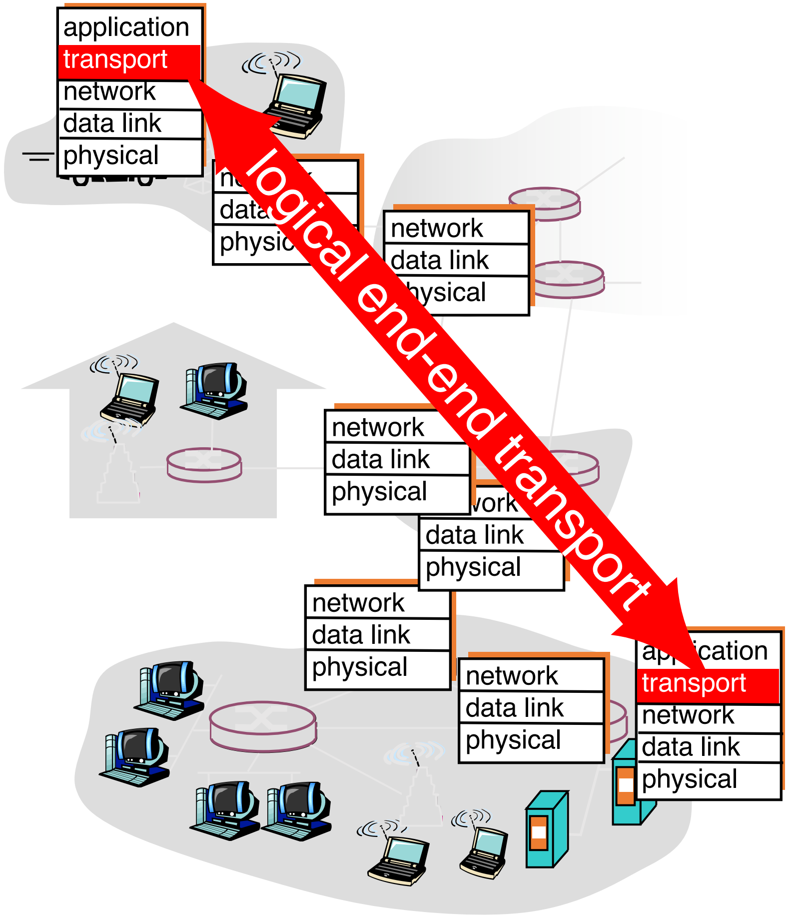
- processes = kids
- app messages = letters in envelopes
- endpoints = houses
- transport protocol = Alice and Bob who de/mux to in-house siblings
- network-layer protocol = postal service

What transport guarantees do you want?

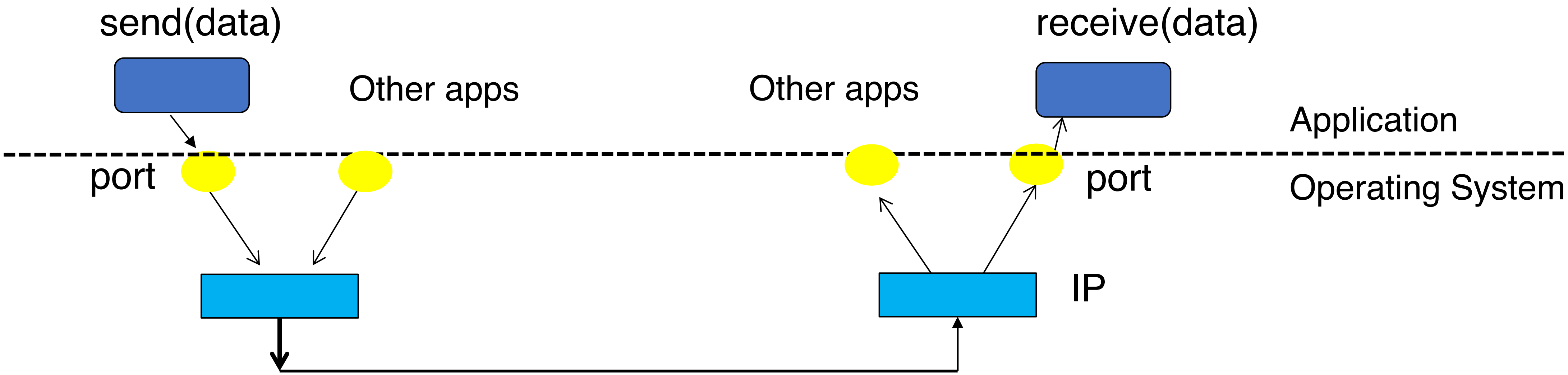
- Reliability
  - Don’t drop messages
  - Don’t corrupt messages
- Performance
  - Get messages to other side as soon as possible
  - As cheaply as possible
- Ordering
  - Don’t reorder messages

Internet transport-layer protocols

- Reliable, in-order delivery: Transmission Control Protocol (TCP)
  - congestion control, flow control, connection setup
- Unreliable, unordered delivery: User Datagram Protocol (UDP)
  - A small extension to underlying network layer protocol, IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



Layering: in terms of packets



Host-to-Net Addresses (S, D)	Src, Dst IP addr	Port #	Data

User Datagram Protocol

UDP: User Datagram Protocol [RFC 768]

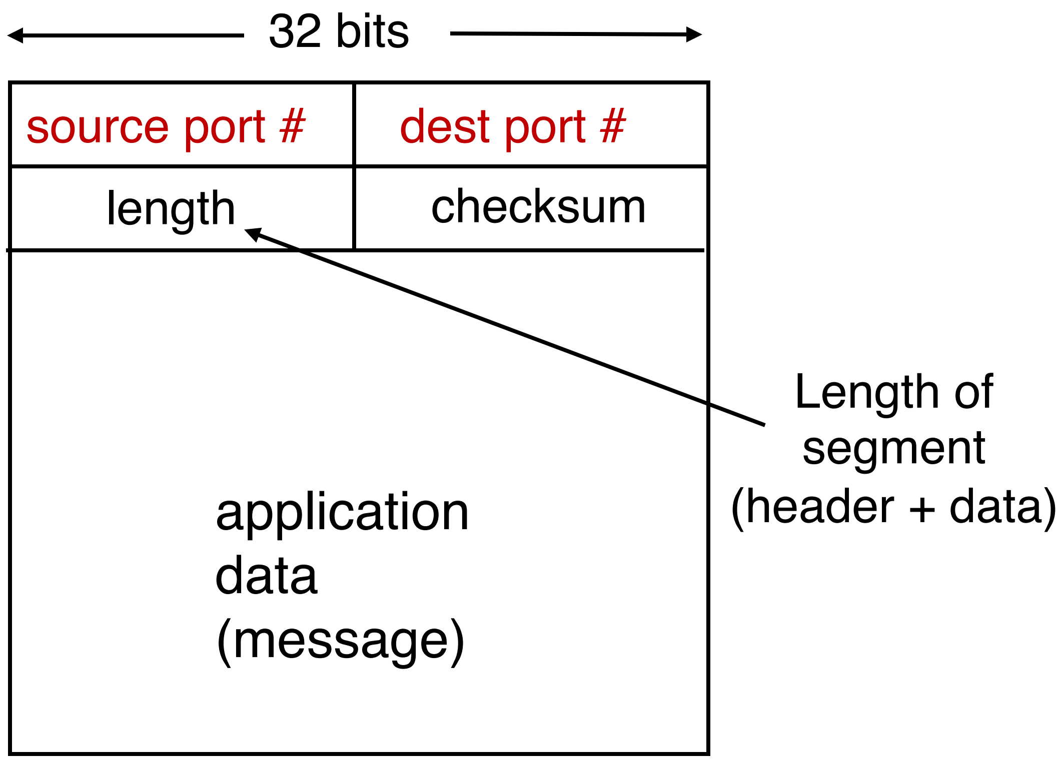
- Best effort service. UDP segments may be:
  - Lost
  - Delivered out of order to app
- UDP is connectionless
  - Each UDP segment handled independently of others (i.e. no "memory" across packets)
- DNS uses UDP
  - UDP suitable for one-off req/resp
  - Also for loss-tolerant delay-sensitive apps, e.g., video calling

Why are UDP's guarantees even okay?

- Simple & Low overhead
- No delays due to connection establishment
  - UDP can send data immediately
- No memory for connection state at sender & receiver
- Small segment header
- UDP can blast away data as fast as desired
  - UDP has no congestion control

How demultiplexing works

- Host receives IP datagrams
  - Datagram contains a transport-level segment
  - each segment has source IP address, destination IP address
  - each segment has source, destination port number
- Host uses IP addresses & port numbers to direct segment to appropriate app-level socket



UDP segment format

Time for an activity

Connectionless demultiplexing

- Create sockets with endpoint-local port numbers to receive data

```
// Example: Python UDP socket
sock = socket(AF_INET, SOCK_DGRAM);
// can bind sock to specific local port
```

- When creating data to send into a UDP socket, you must specify the remote IP address and port.

```
sock.sendto(msg, ("128.1.2.3", 4500));
```

- When endpoint receives UDP segment:
  - Lookup a table with the key (destination IP, destination port) to check if a corresponding socket exists
  - Direct UDP segment to socket
- Datagrams with different source IP addresses and/or source port numbers are directed to the same socket
  - Subtle difference from TCP demultiplexing.

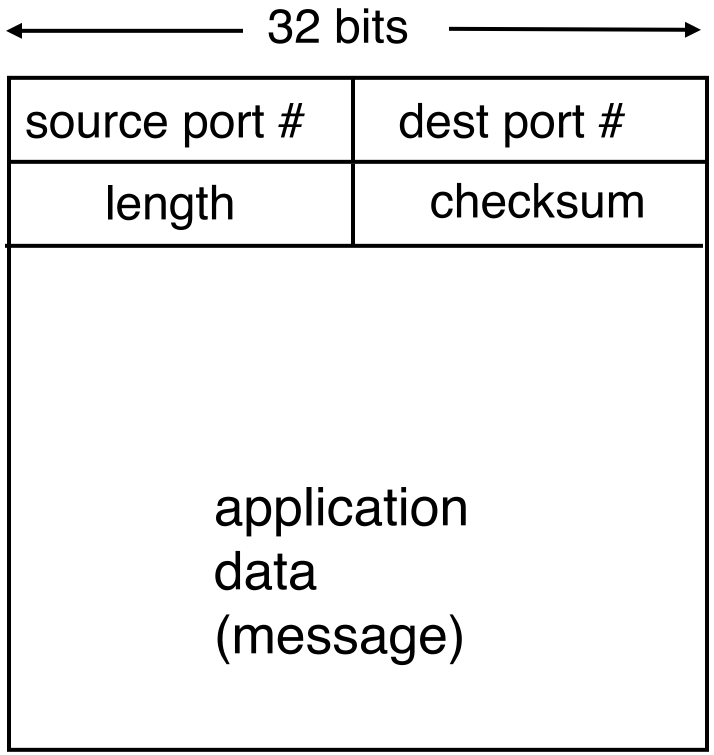
Error Detection

Necessary, but insufficient, for reliability

Data may get corrupted along the way…

- Bits flipped from 0→1 or 1→0
- Packet bits lost
- How to detect errors?

- Idea: compute a function over the data
  - Store the result along with the data
- Function must be easy to compute
- Function & stored data efficient to verify
- Ideas for functions?



UDP segment format

From the UDP specification (RFC 768)

- Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

- The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol, and the UDP length.

UDP Checksum

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.

UDP 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
```

User Datagram Protocol

- A thin shim around best-effort network delivery
  - Lightweight to send one-off request/response messages
  - Lightweight transport for loss-tolerant delay-sensitive applications

- Provides basic multiplexing/demultiplexing for applications

- No reliability, performance, or ordering guarantees
- Can do basic error detection (bit flips) using checksums
  - Error detection is a necessary condition for reliability
  - But need to do much more for reliability. Subject of the next lecture.