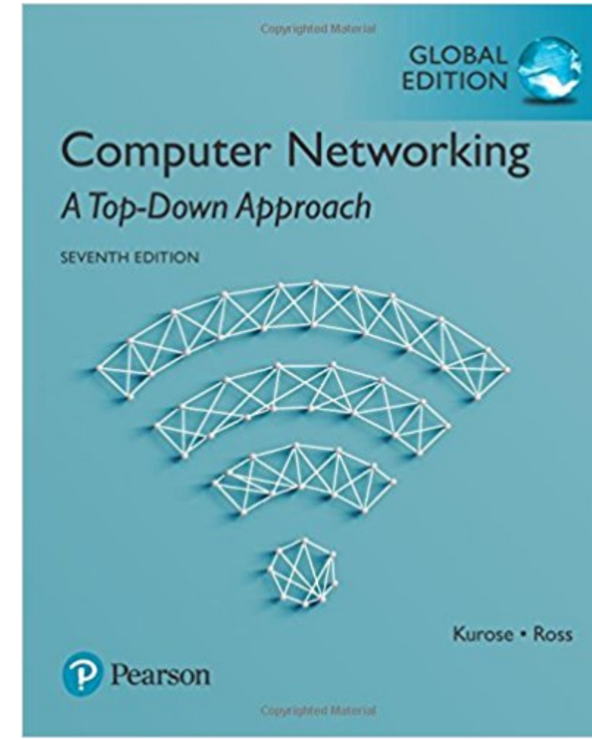
Chapter 3
Transport Layer part 1

School of Computing
Gachon Univ.
Joohyung Lee

Most of slides from J.F Kurose and K.W. Ross. And, some slides from Prof. Joon Yoo



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson, 2017

Chapter 3: 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

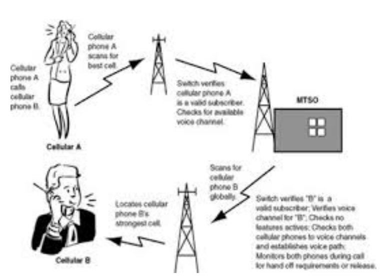
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

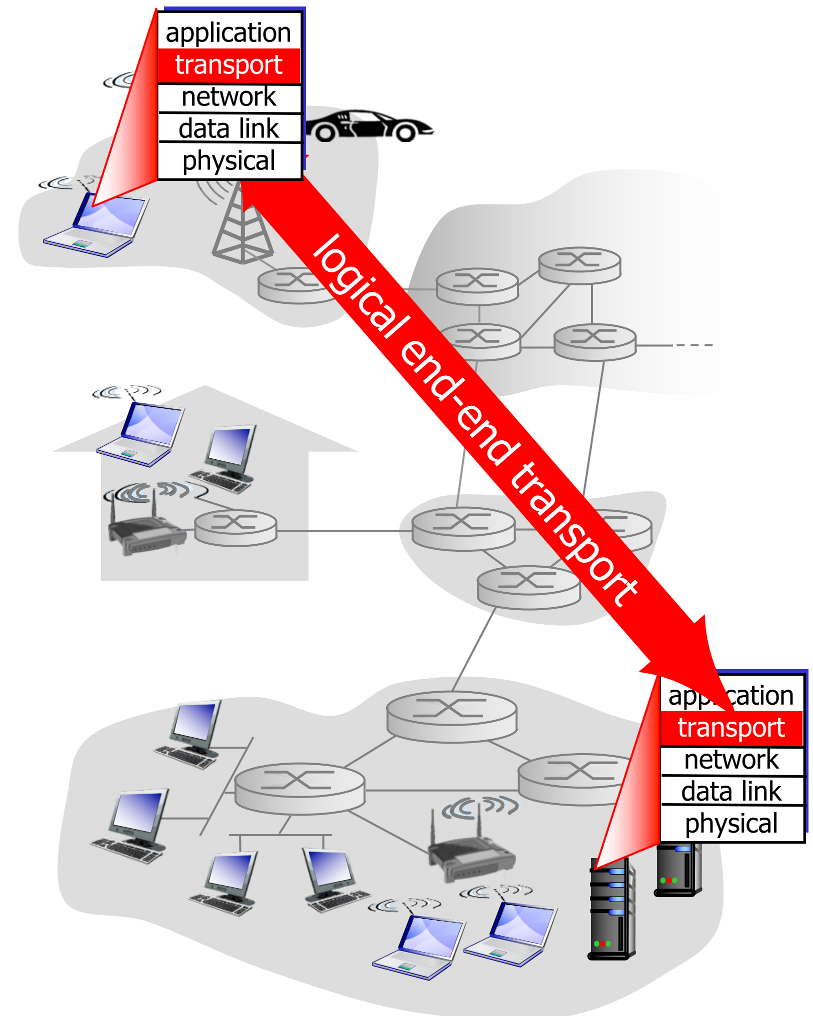
3.6 principles of congestion control

3.7 TCP congestion control

Transport services and protocols

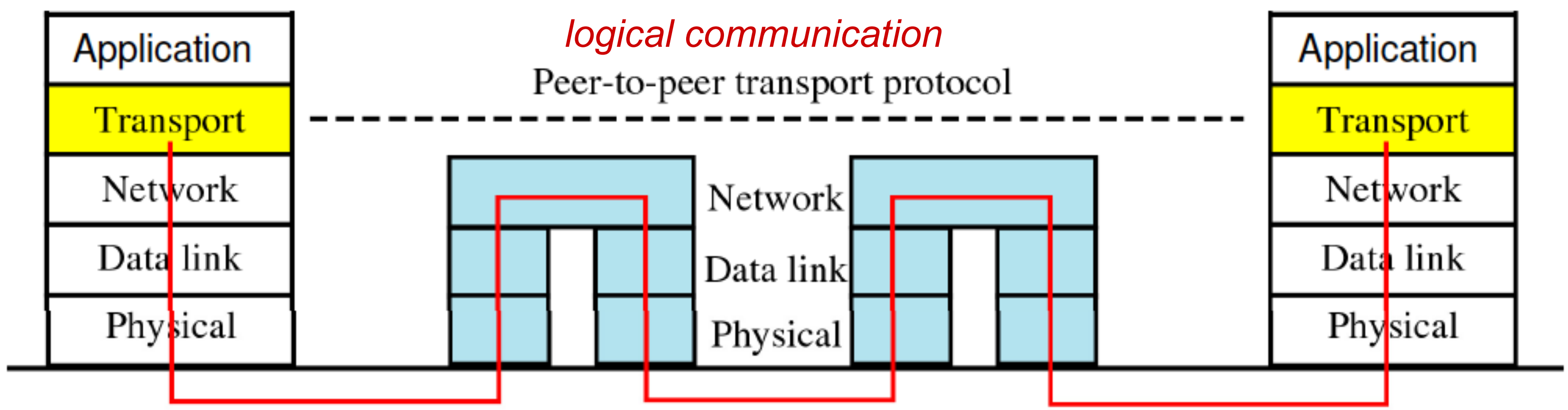
- provide *logical communication* between app processes running on different hosts
  - "logical" communication?
    - The communicating processes are not necessarily physically and directly connected to each other – there are many routers/links in-between
    - But, from the processes' viewpoint, it is as if they were physically connected.





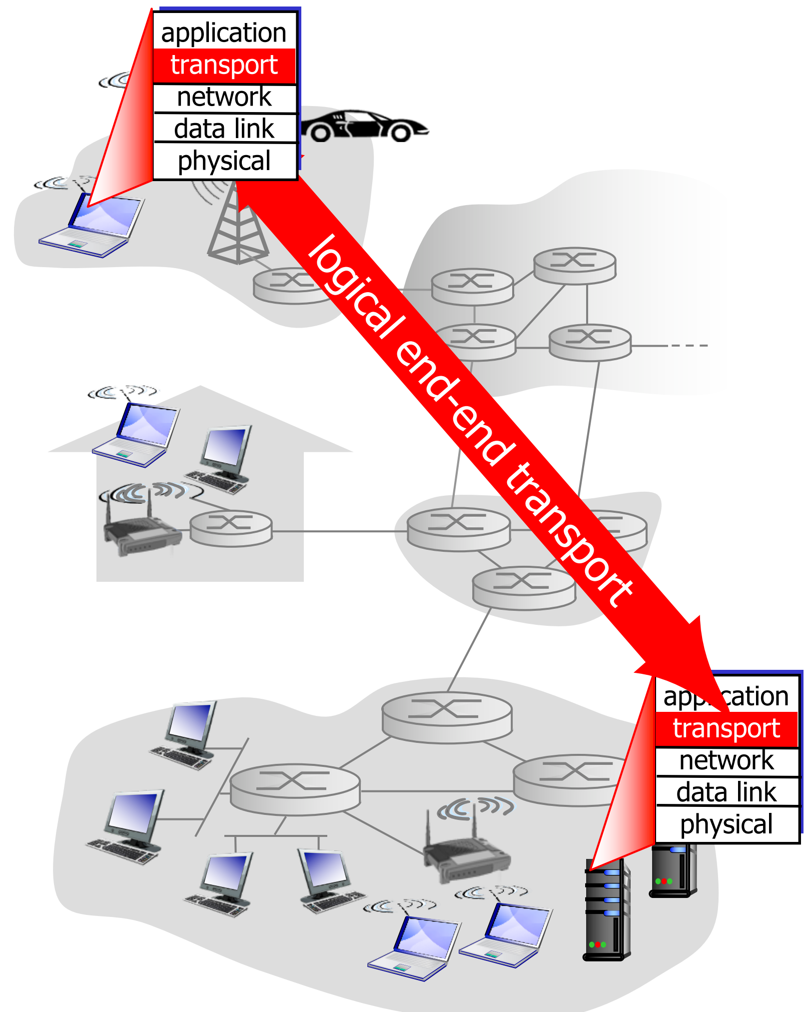
Transport vs. network layer

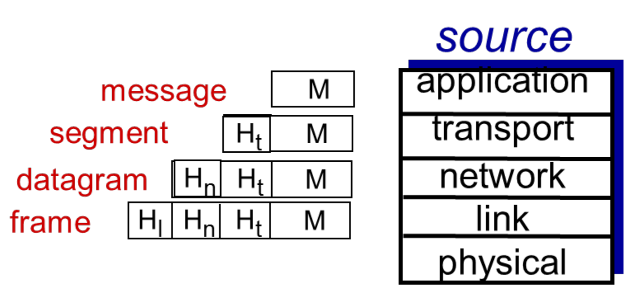
- Transport : end-to-end layer communication – not implemented in the intermediate routers



Transport services and protocols

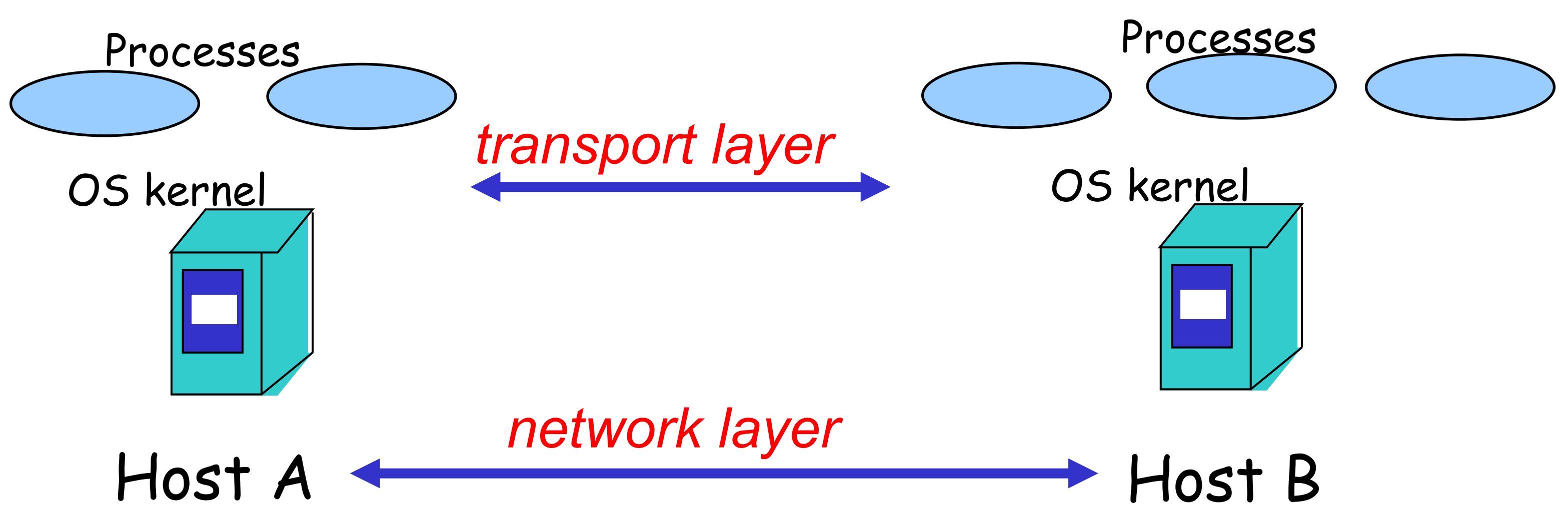
- Transport protocols run in end systems
  - sender side: breaks app messages into ***segments***, passes to network layer
  - receiver side: reassembles ***segments*** into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP





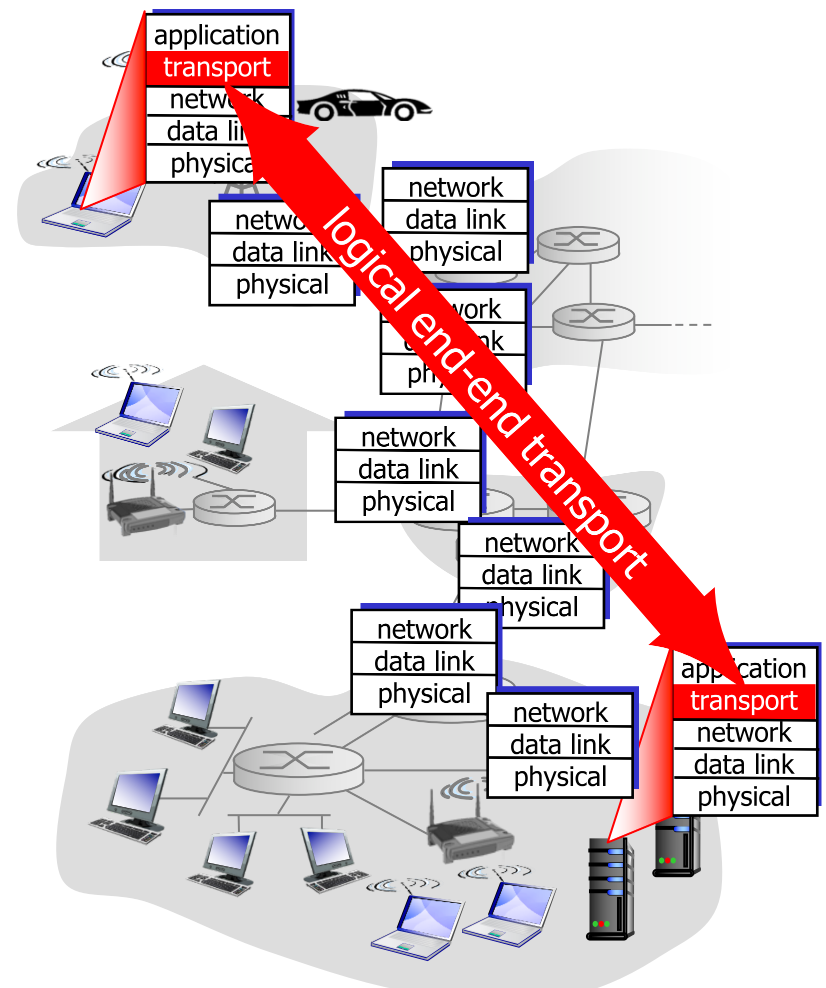
Transport vs. network layer

- *transport layer:* logical communication between **processes**
  - Move message from application processes to another using **Port numbers**
  - TCP provides **reliability**
- *network layer:* logical communication between **hosts**
  - Move message within the network core using **IP address**
  - **Best-effort delivery**: no guarantees on integrity, order



Internet transport-layer protocols

- Transport layer
  - Extend from end-system (host) delivery (Network layer; IP) to process-to-process delivery
    - *Multiplexing / Demultiplexing*
- TCP
  - reliable, in-order delivery
  - congestion control, flow control
  - connection setup
- UDP
  - unreliable, unordered delivery
  - no connection, congestion, flow control



Chapter 3 outline

UDP: User Datagram Protocol [RFC 768]

- **UDP**
  - unreliable, unordered delivery
  - Add/use **port numbers**
  - Light **error checking** (is there an error?)
  - *connectionless:*
    - no handshaking between UDP sender, receiver
    - each UDP segment handled independently of others

udp : 에러가 있는지만 알아본다. 있어도 후속 조치가 없다.

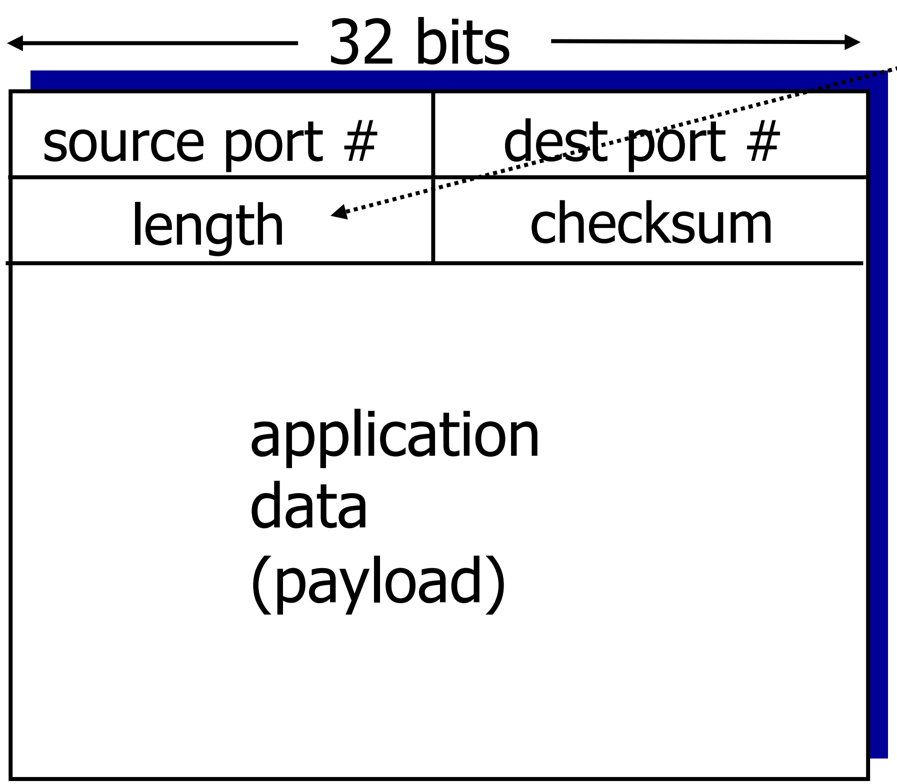
- UDP usage:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS

- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

transport에서 오류가 나면 다시 보냄(reliable하게).
application 딴에서 reliable하게 전송하는 것.

udp는 스트리밍에 적합하다는 공식이 깨져서 요즘에는 tcp 기반으로 운영된다.

UDP: segment header



length, in bytes of UDP segment, including header

UDP segment format

why is there a UDP?

- faster: no connection establishment (which can add delay)
  - e.g., DNS
- simple: no connection state at sender, receiver
- Lightweight: small header size, no congestion/flow control
  - e.g., Real-time applications

UDP for DNS

- DNS query / DNS reply sent via **UDP**
  - But UDP is *unreliable*! **Why not** use TCP instead?
  - DNS is used often, delay should be minimal – DNS uses fast UDP
    - For Web pages with text, reliability is critical - HTTP uses TCP
  - DNS *application* itself provides reliability
    - If an DNS reply does not arrive, resend another DNS query

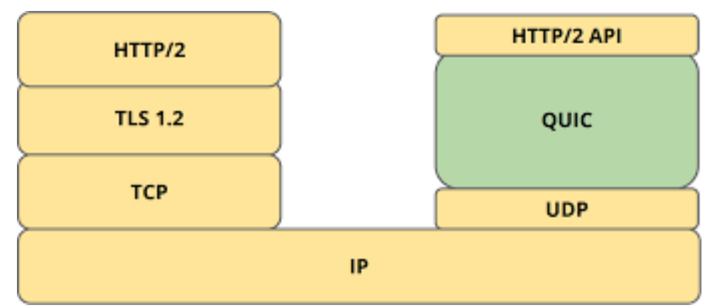
왜 reliable한 전송을 중복해서 할까?
dms 메세지를 seg로 쪼갠다하면 재전송할 때 부하가 줄어듦

UDP for QUIC

- Google QUIC in chrome browser
  - based on UDP
  - Application-layer protocol: In-between HTTP and UDP
  - Used in chrome browser for Google services

Making the internet faster with...

Quick
UDP
Internet
Connections



How to disable QUIC protocol in Google Chrome

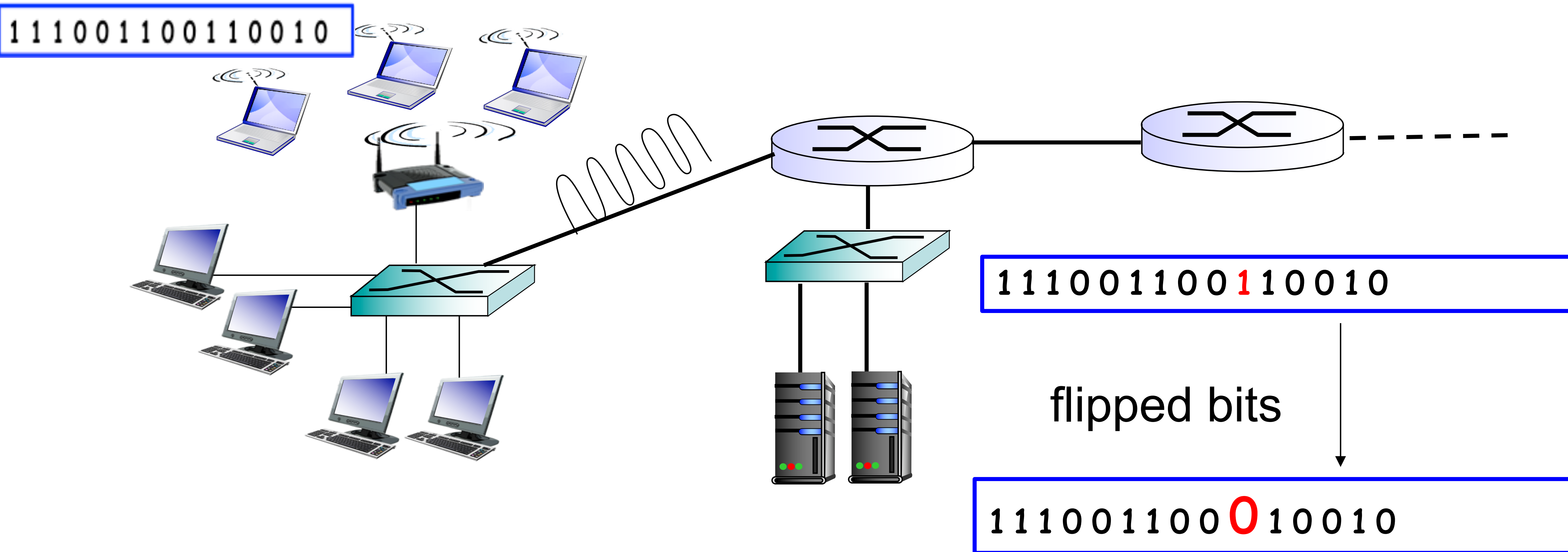
Bitdefender GravityZone provides full visibility into organizations' overall security posture, global security threats, and control over its security services that protect virtual or physical desktops, servers and mobile devices. All Bitdefender's Enterprise Security solutions are managed within the GravityZone through a single console, Control Center, that provides control, reporting, and alerting services for various roles within the organization.

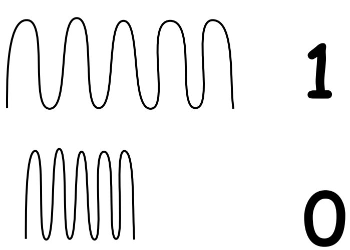
Overview

크롬, 유튜브 등 웹서핑시 느려질 때 해결 방법 QUIC 설정 해제

2017.07.16 00:00

Errors may occur in delivering





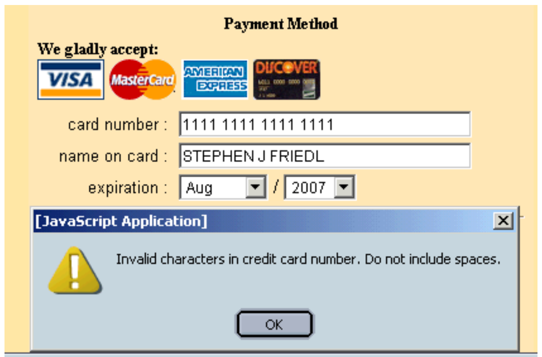
link layer에서도 재전송하는데
access network(기지국)에서 loss가 많이난다. l
ink layer에서는 재전송 기능이 없기에
server에서 재전송 해줘야 한다.
-> 반응 속도도 느려지고 traffic이 늘어남.

Also, there are other types of errors: e.g. bits may be lost

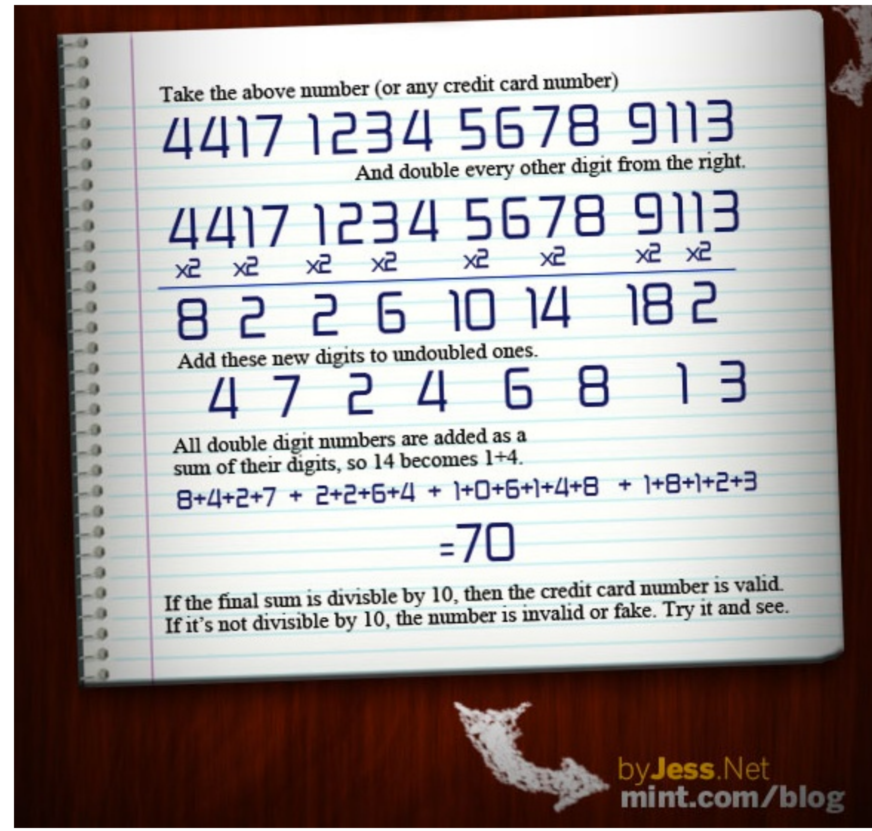
Checksum?

Credit Cards

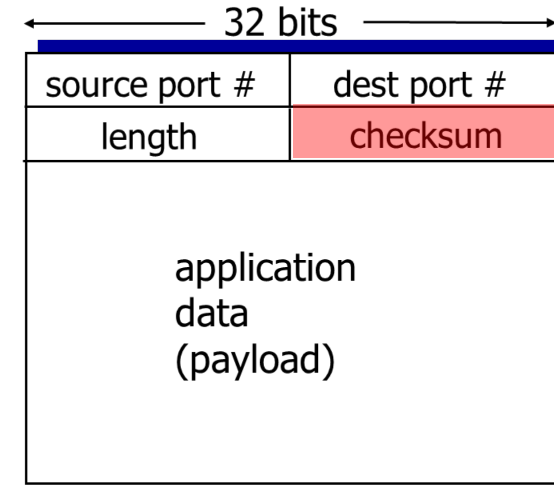
checksum?



- How did they know this?



UDP checksum



UDP segment format

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 ….

udp 가 필요한 것 : 자신과 destination

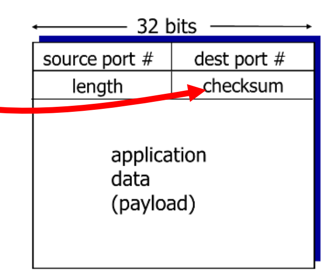
Internet checksum: example

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
```

1s complement



UDP segment format

- Receiver checks for errors:
  - adds the sum and checksum – if no error result is **111…111**
- Can UDP recover from errors? How about TCP?

3-17 : 에러가 있다고 재전송은 x

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

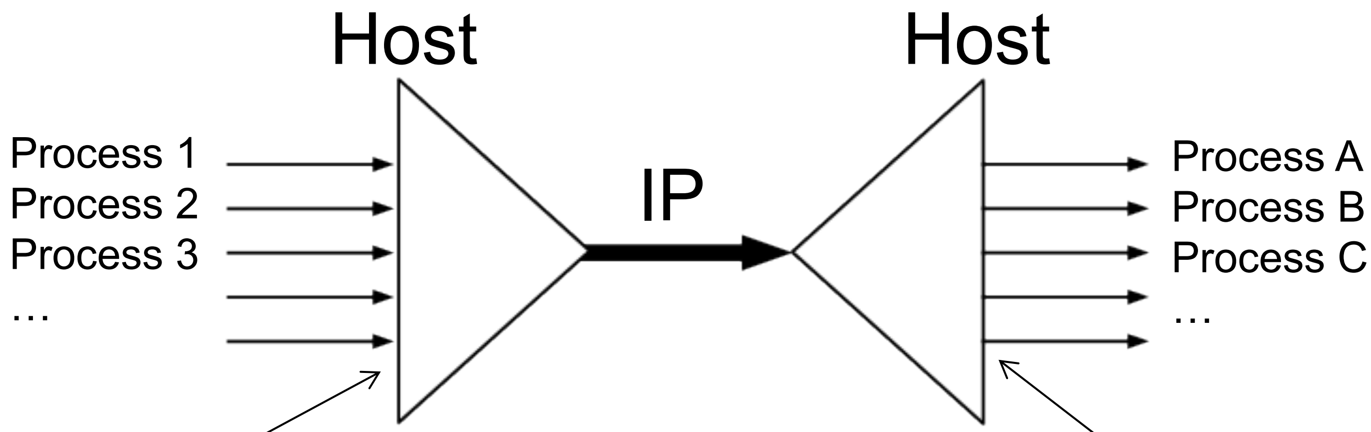
3.4 principles of reliable data transfer

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

Multiplexing / Demultiplexing



Multiplexing: a method by which multiple data streams are combined into one stream/signal over a shared logical/physical communication link

Demultiplexing: performs the reverse process

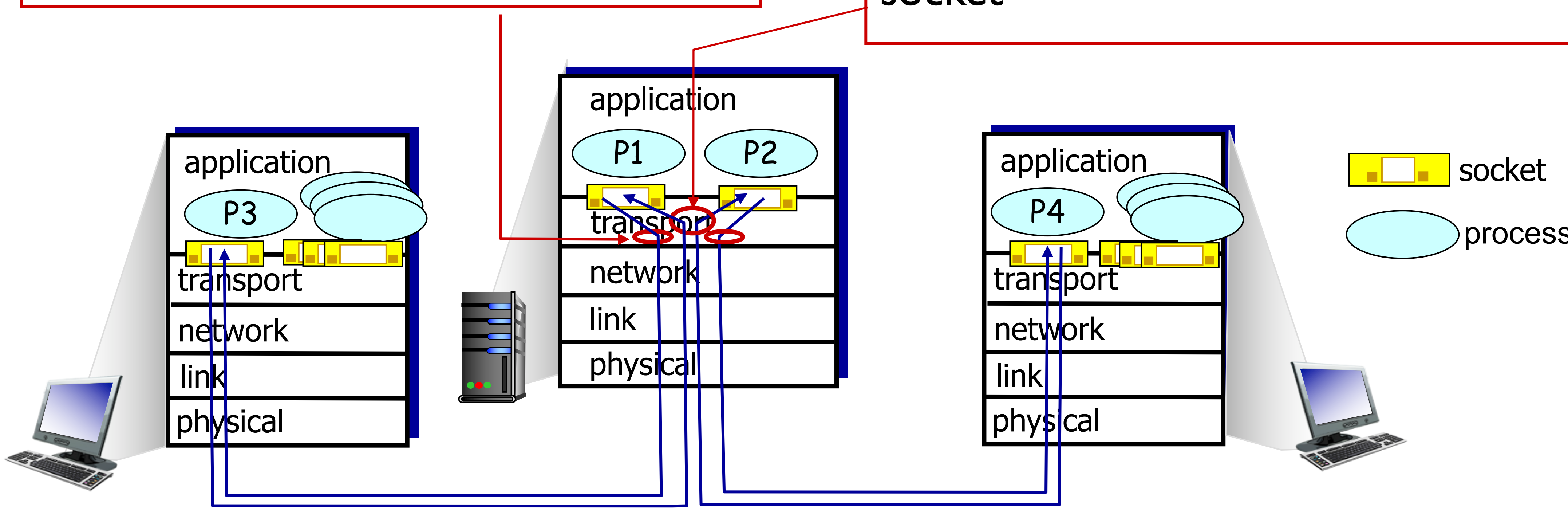
Multiplexing/demultiplexing

multiplexing at sender:

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

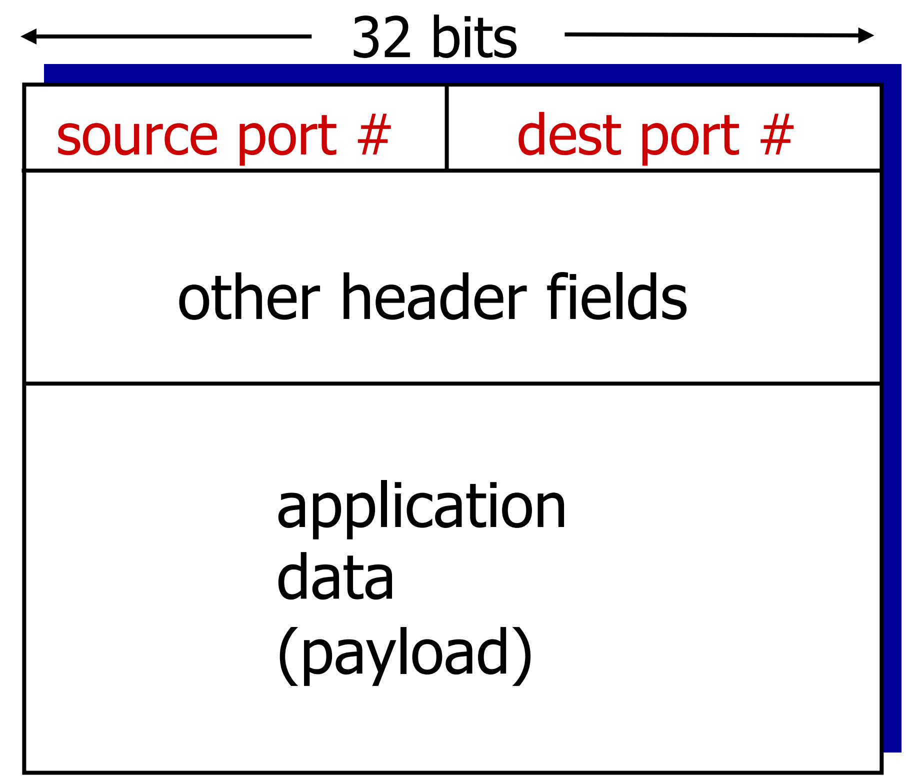
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - each datagram has *source / 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

Connectionless (UDP) demultiplexing

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

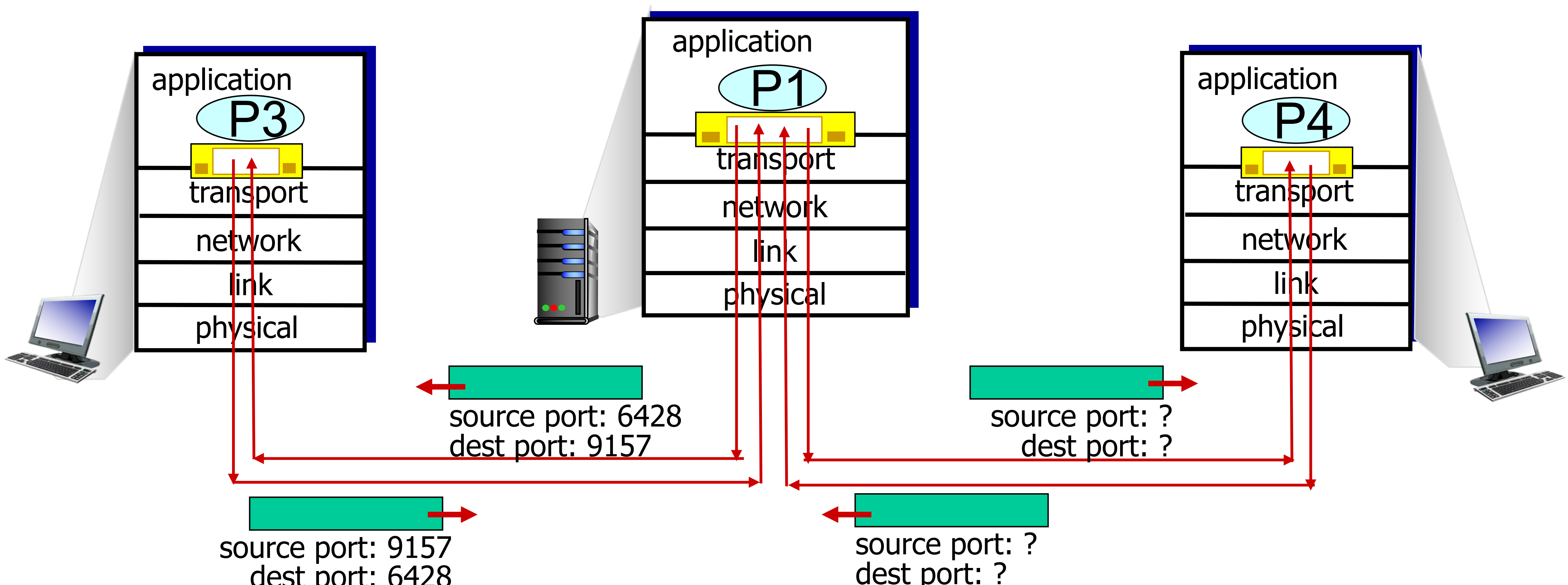
IP datagrams with *same dest. IP address/port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless (UDP) demux: example

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

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

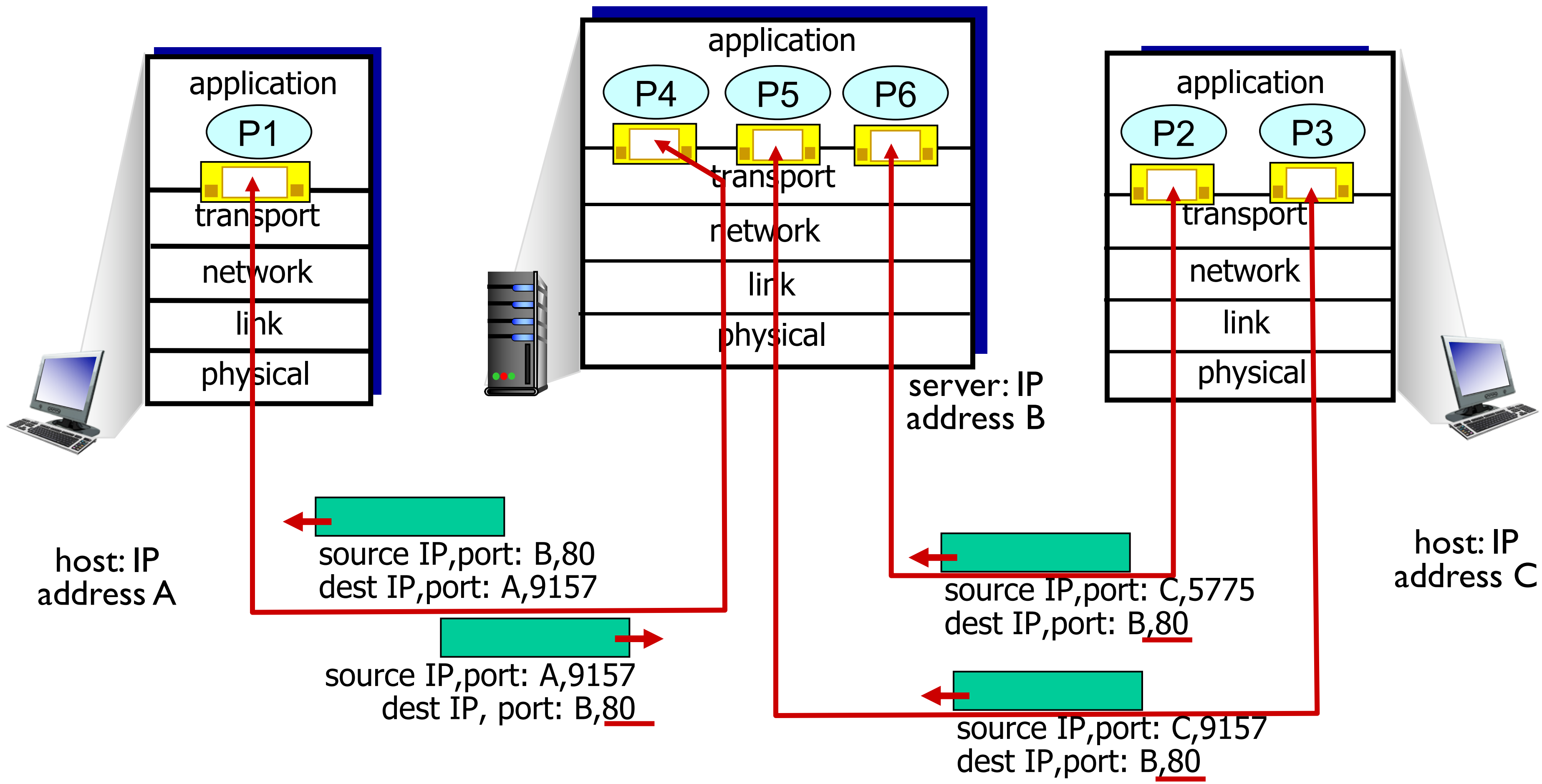
```
DatagramSocket mySocket1
 = new DatagramSocket
 (5775);
```



Connection-oriented (TCP) demux

- **TCP socket** (≈connection) identified by **<u>4-tuple</u>**:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses **<u>all four values</u>** 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 (TCP) demux: example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets