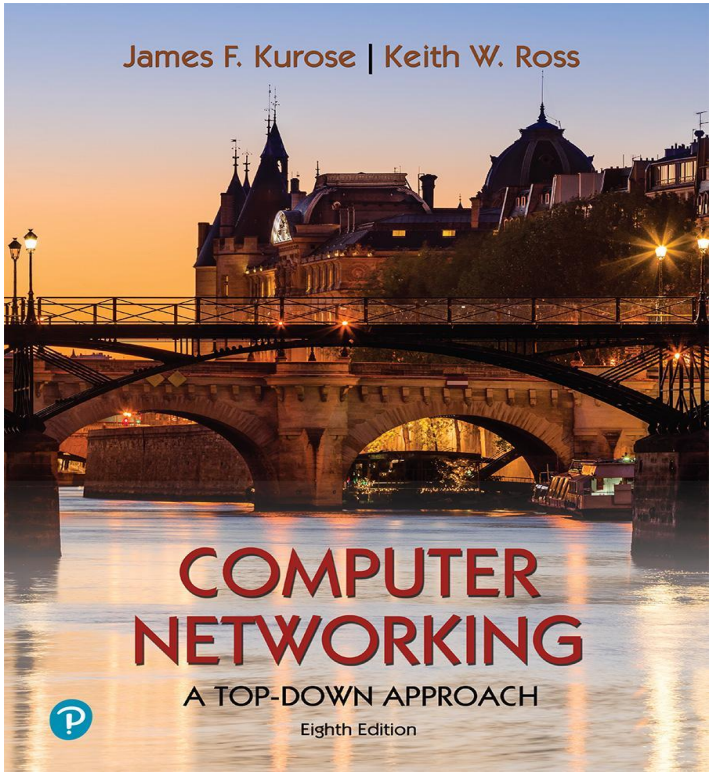
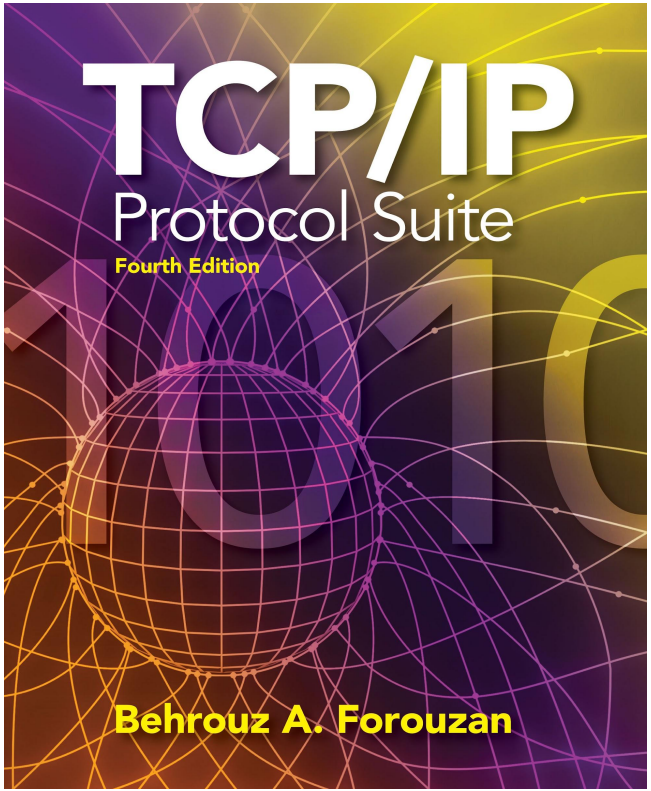
Transport Layer

Sources

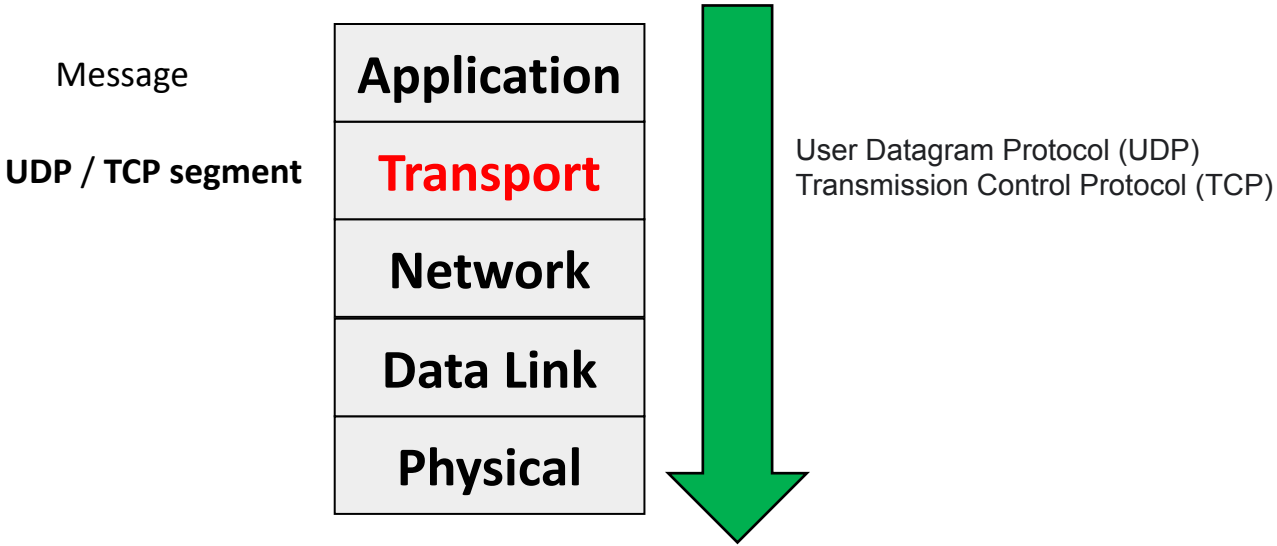


Computer Networking: A Top-Down Approach

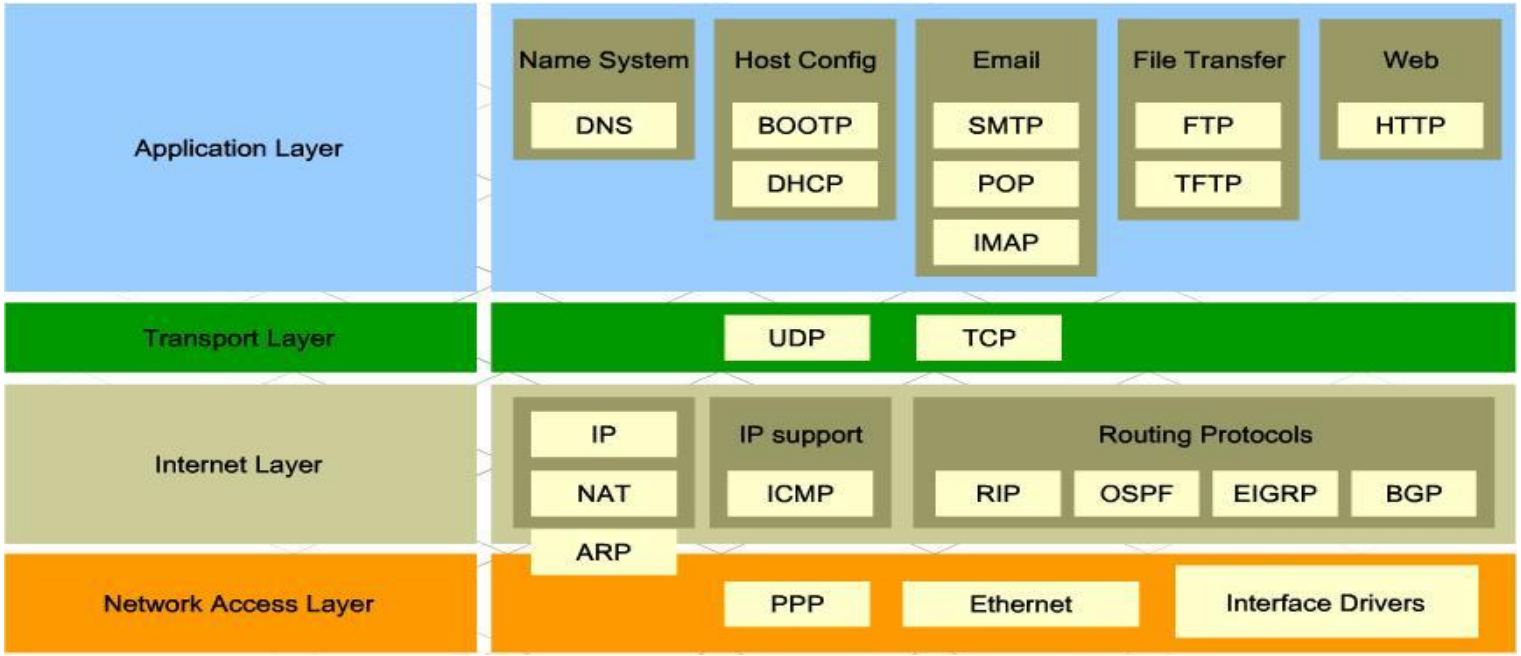


TCP/IP Protocol Suite

Top Down Approach

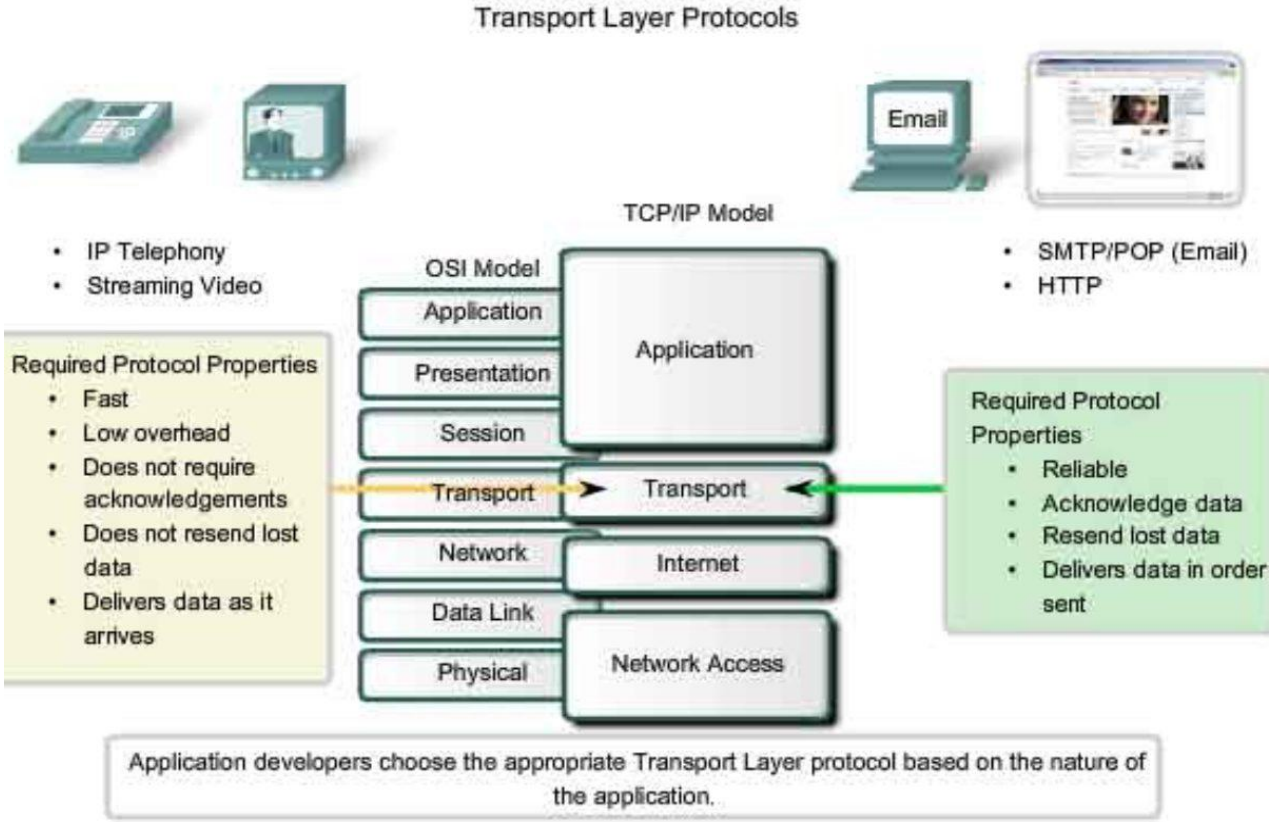


Protocols @ Different layers

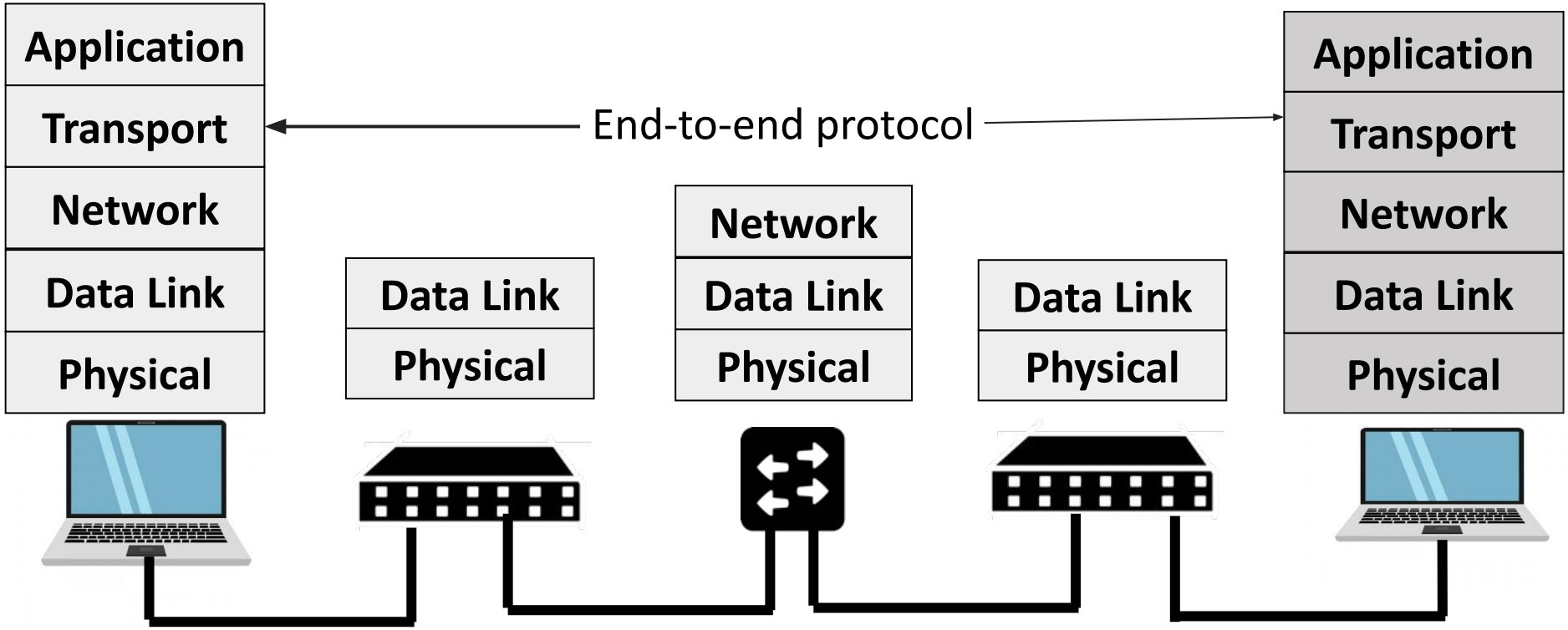


Source: http://walkwidnetwork.blogspot.com/2013/04/application-layer-internet-protocol.html

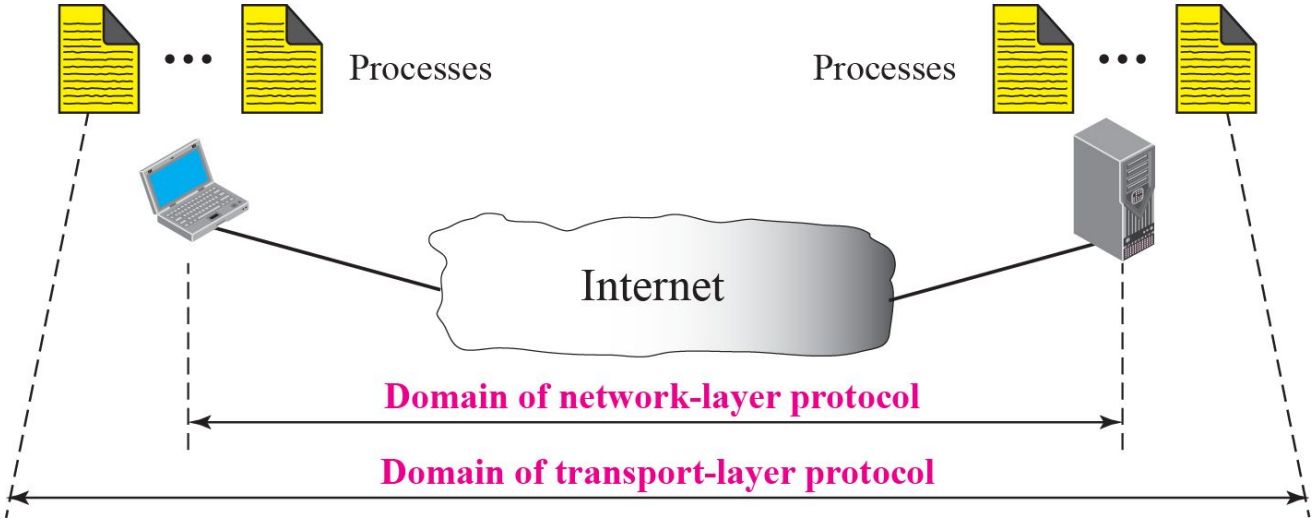
Transport Layer Protocols



Communication between two remote Machine

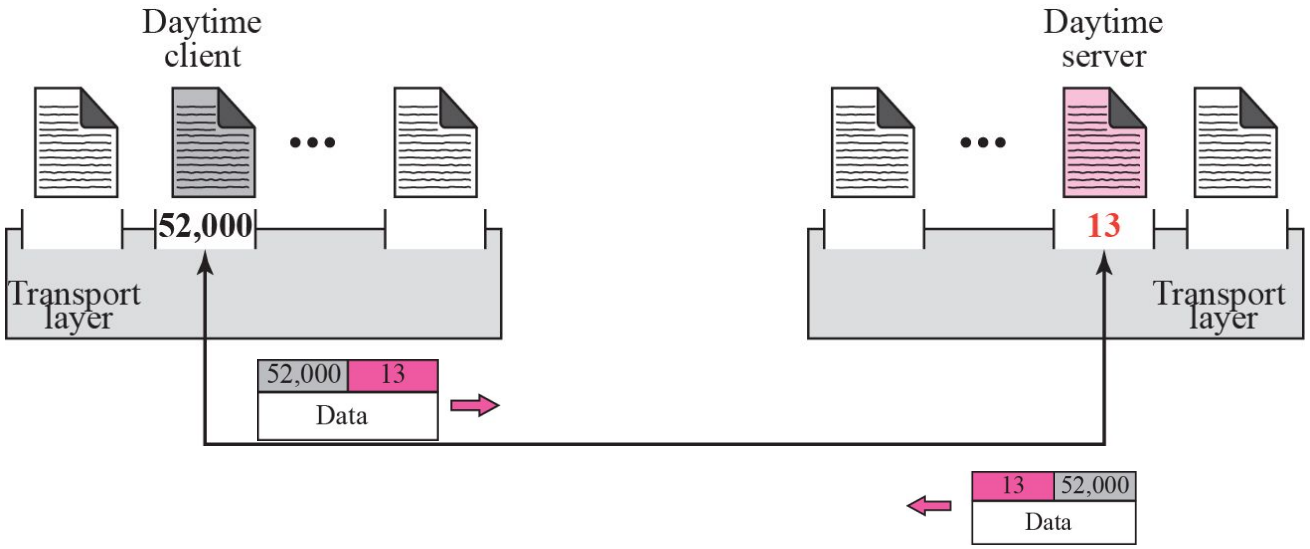


Network layer versus transport layer



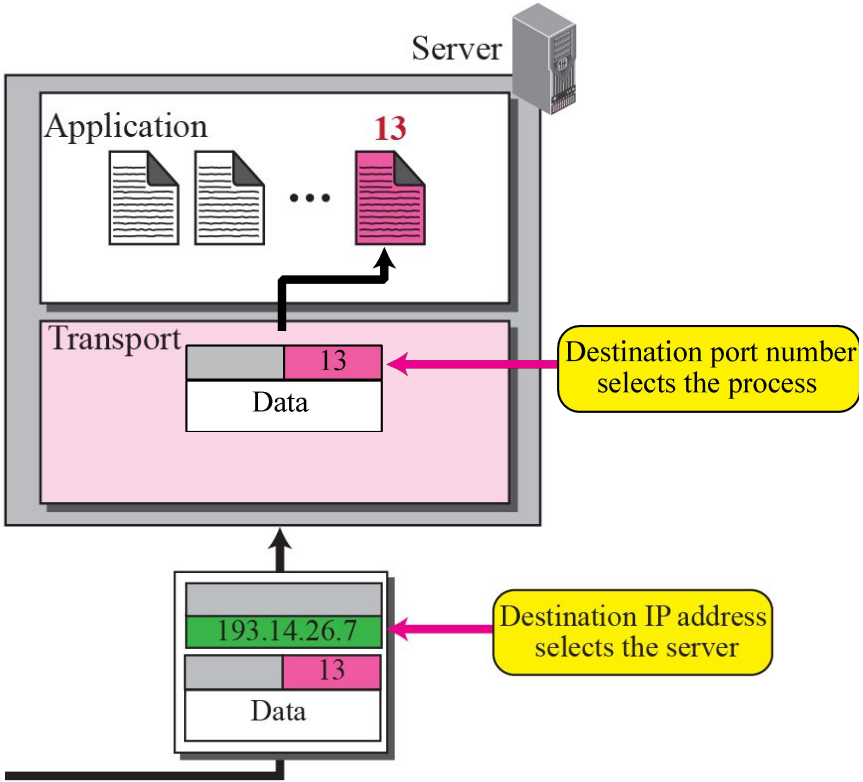
TCP/IP Protocol Suite

Port numbers

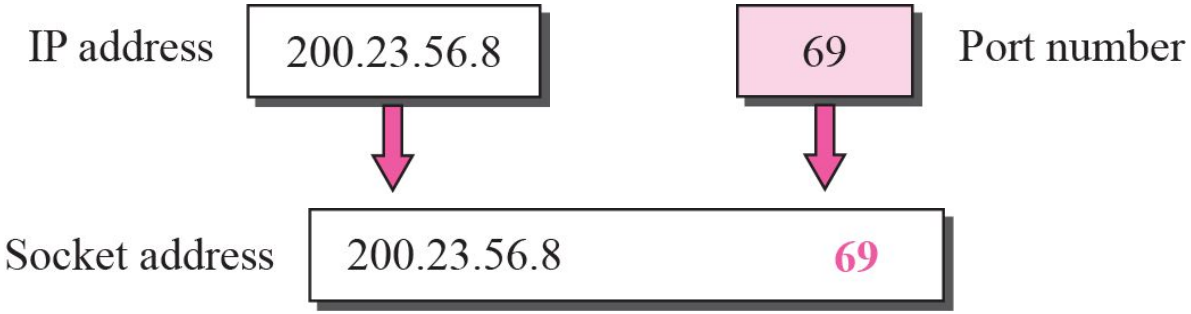


TCP/IP Protocol Suite

IP addresses versus port numbers

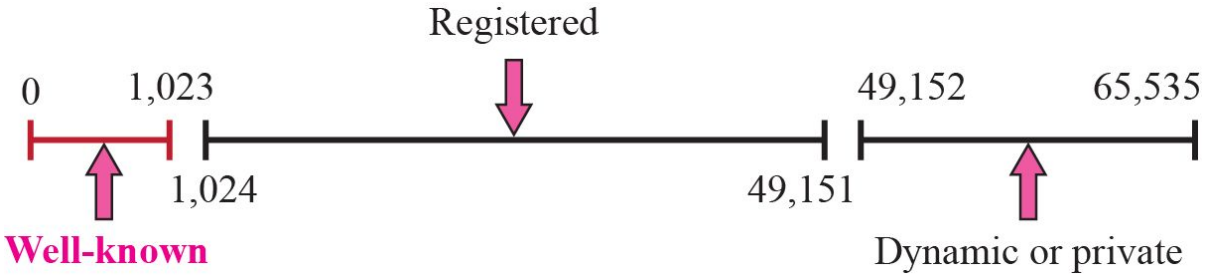


Socket address



TCP/IP Protocol Suite

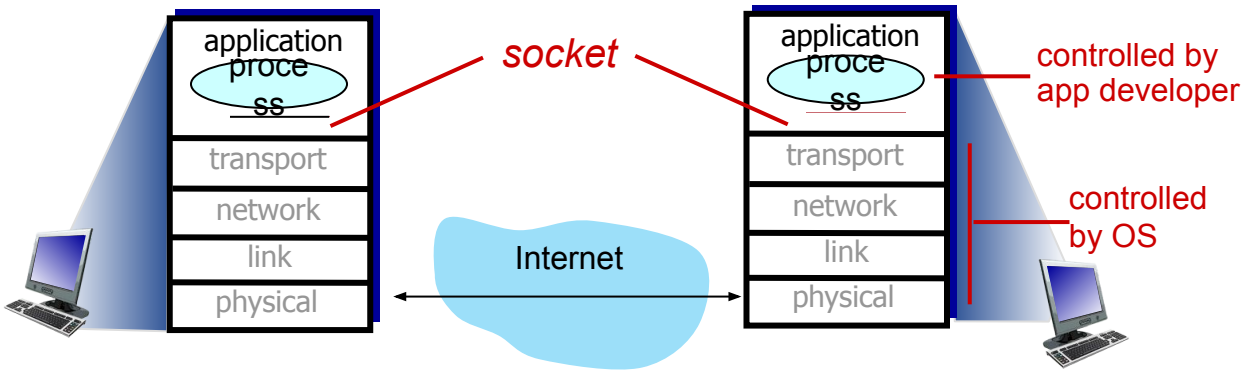
ICANN (Internet Corporation for Assigned Names and Numbers) Ranges

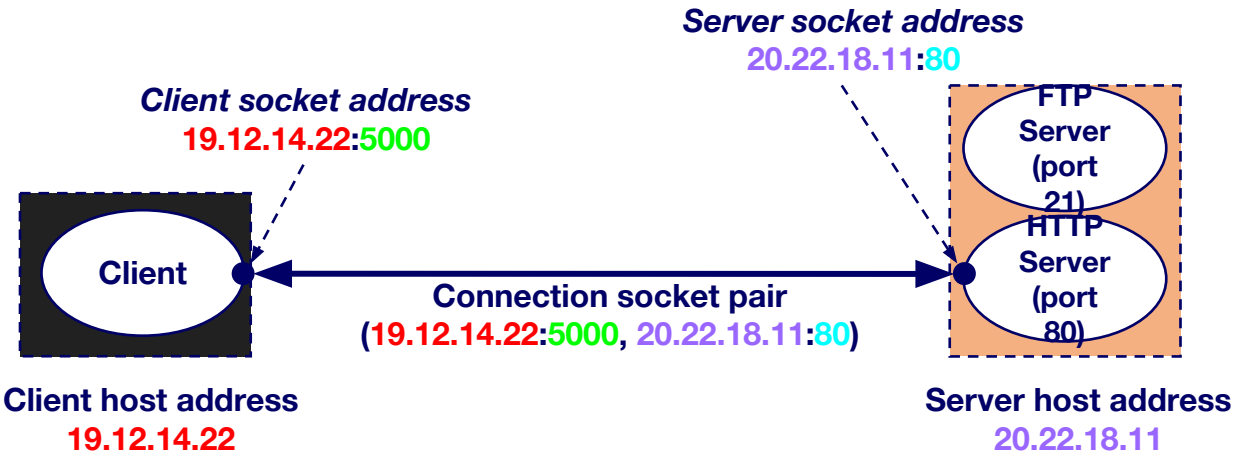


- *The well-known port numbers are less than 1,024. These are used by processes that provide widely used types of network services.*
- **Registered Port Numbers:** They are assigned by IANA (**Internet Assigned Numbers Authority, Owner ICANN**) for specific service upon application by a requesting entity.
- **Dynamic Port Numbers:** This range is used for private or customized services, for temporary purposes, and for automatic allocation

https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports

Socket Pair


application
process
transport
network
link
physical
socket
Internet
application
process
transport
network
link
physical
controlled by
app developer
controlled
by OS


Server socket address
20.22.18.11:80
Client socket address
19.12.14.22:5000
FTP
Server
(port
21)
HTTP
Server
(port
80)
Client
Connection socket pair
(19.12.14.22:5000, 20.22.18.11:80)
Client host address
19.12.14.22
Server host address
20.22.18.11

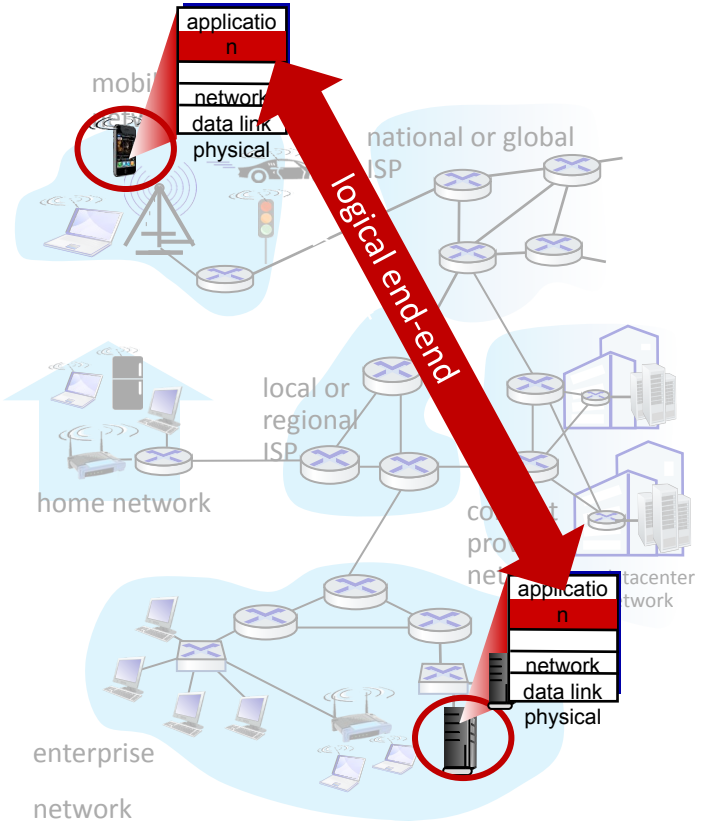
Transport layer: overview

Our goal:

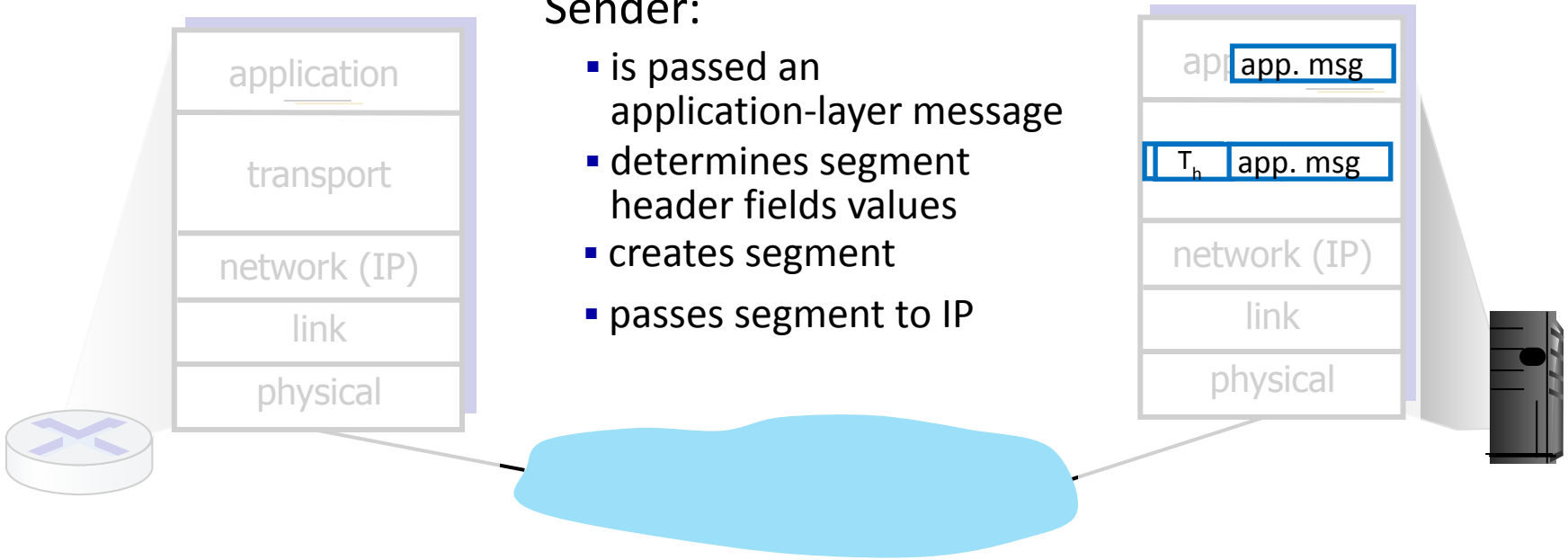
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion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

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



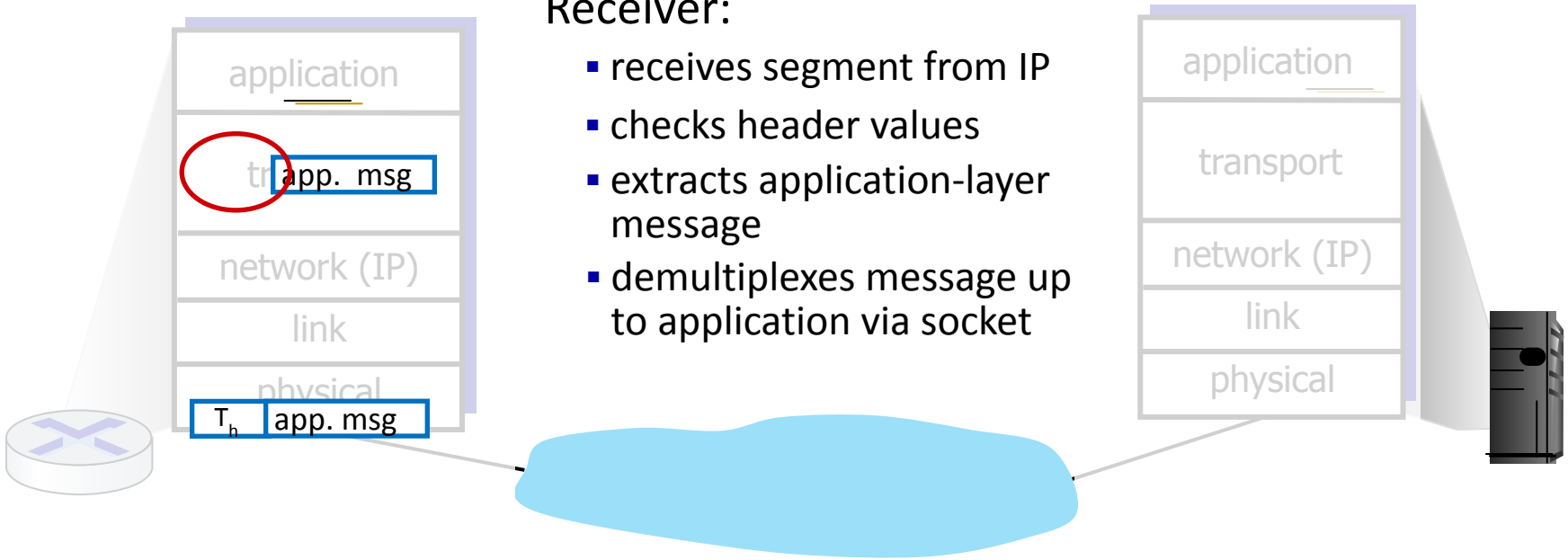
Transport Layer Actions



Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

Transport Layer Actions

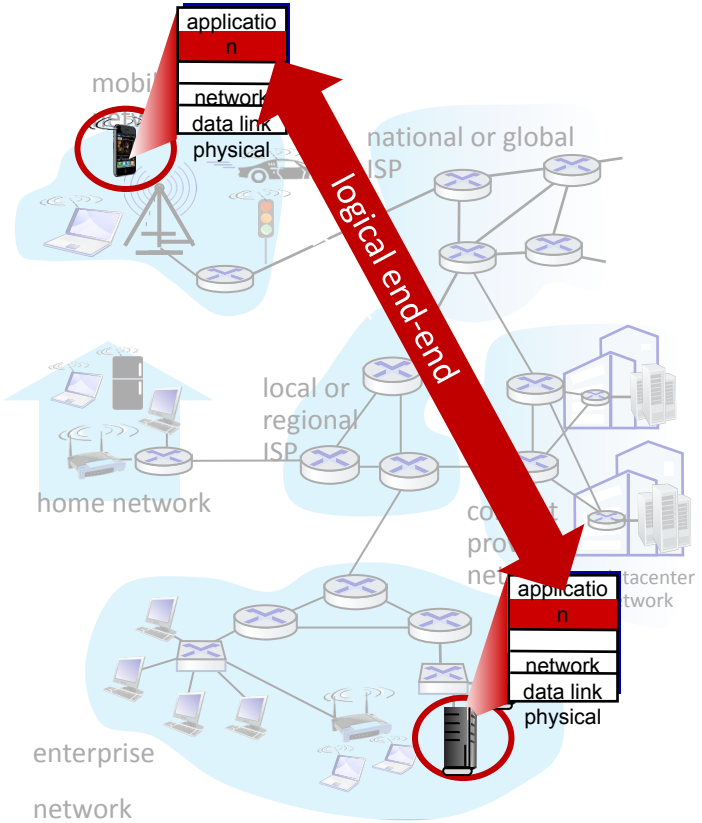


Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

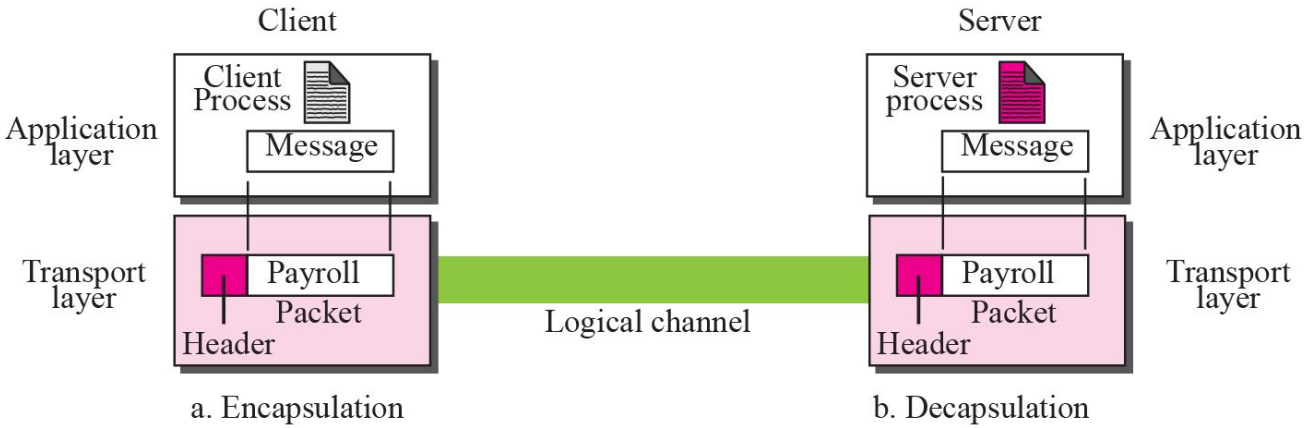


Transport Layer Services

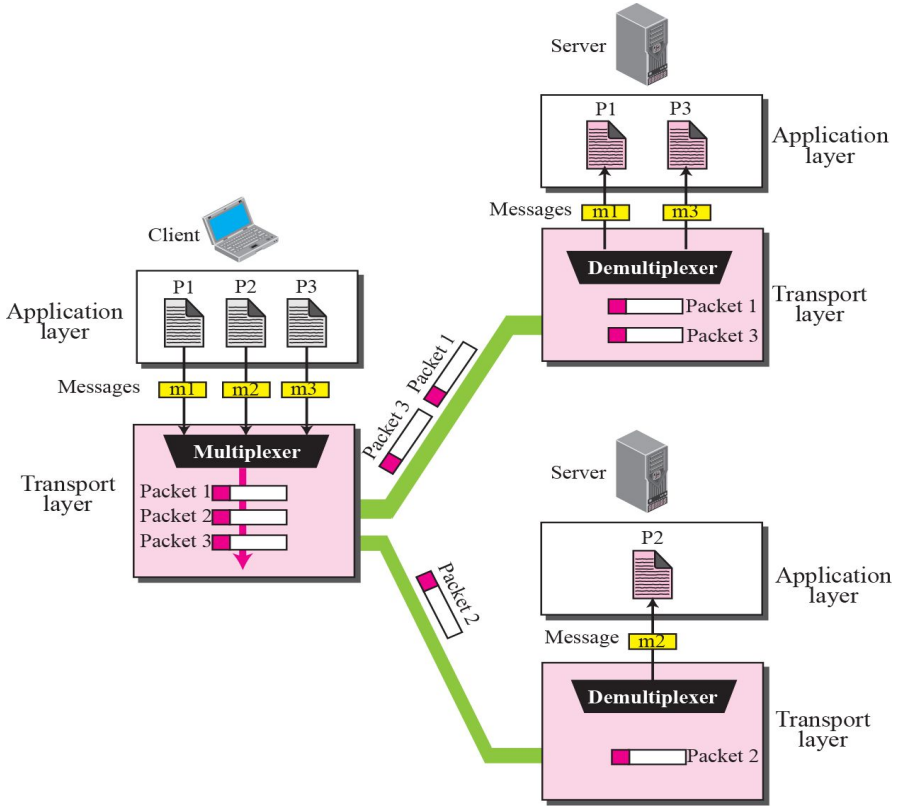
- ✔ Encapsulation and Decapsulation
- ✔ Multiplexing and Demultiplexing
- ✔ Flow Control
- ✔ Error Control
- ✔ Congestion Control
- ✔ Connectionless and Connection-Oriented Services

TCP/IP Protocol Suite

Encapsulation and Decapsulation



Multiplexing and Demultiplexing



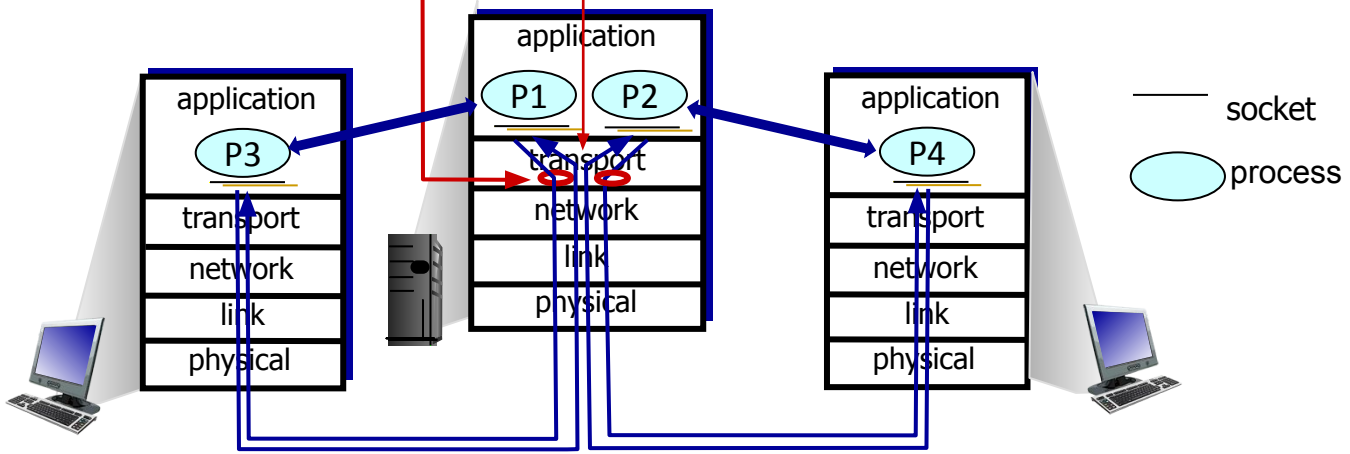
Multiplexing/demultiplexing

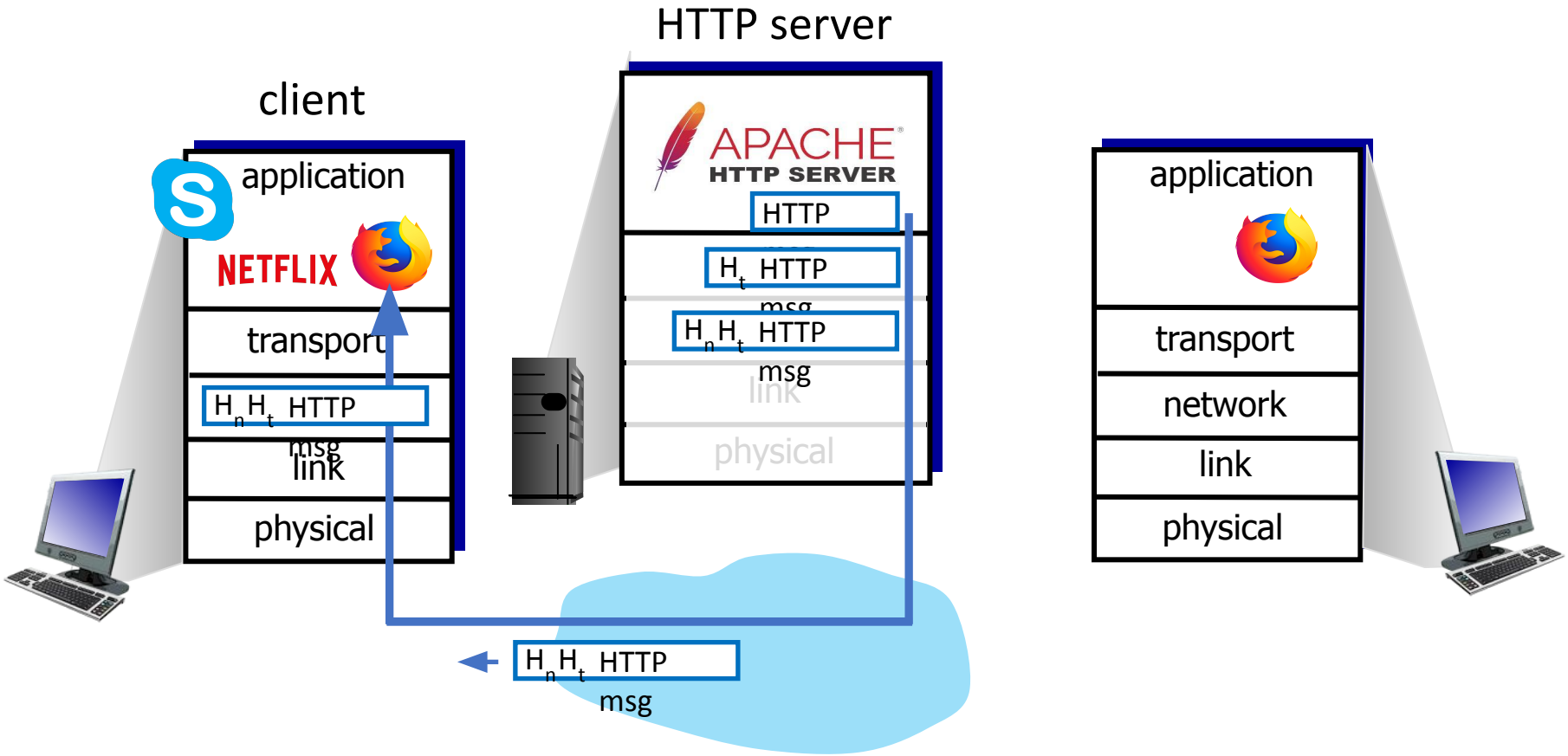
multiplexing as sender:

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

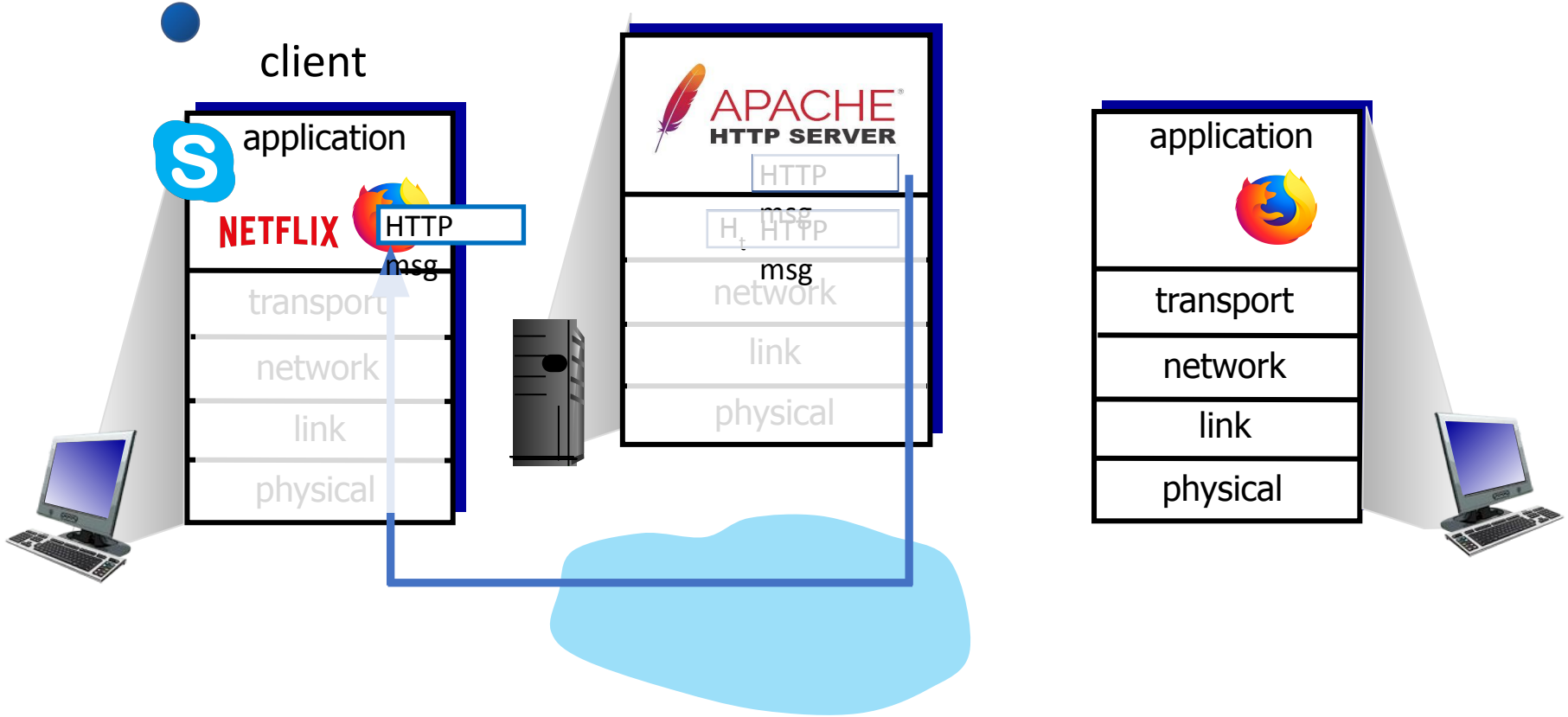
demultiplexing as receiver:

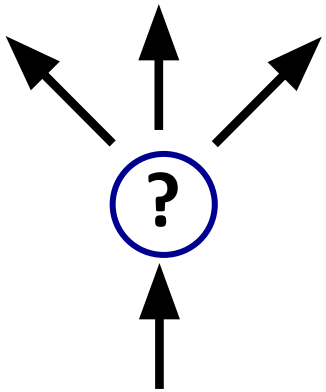
use header info to deliver received segments to correct socket



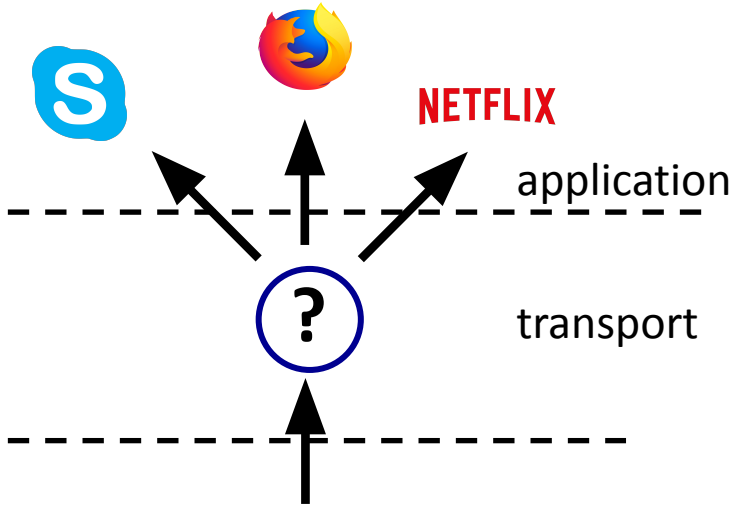

HTTP server
client
application
NETFLIX
transport
$H_n H_t$ HTTP msg
link
physical
APACHE
HTTP SERVER
HTTP
H_t HTTP msg
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg

Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?



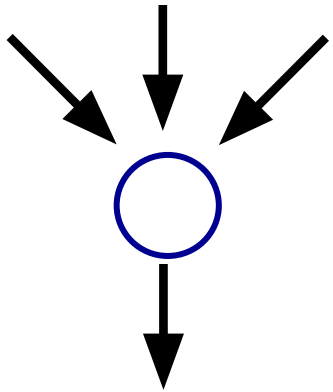


de-multiplexing

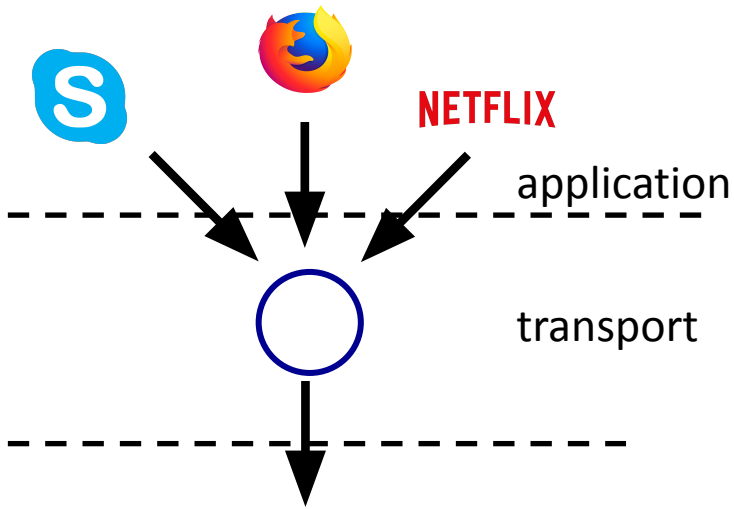


de-multiplexing

Demultiplexing



multiplexing

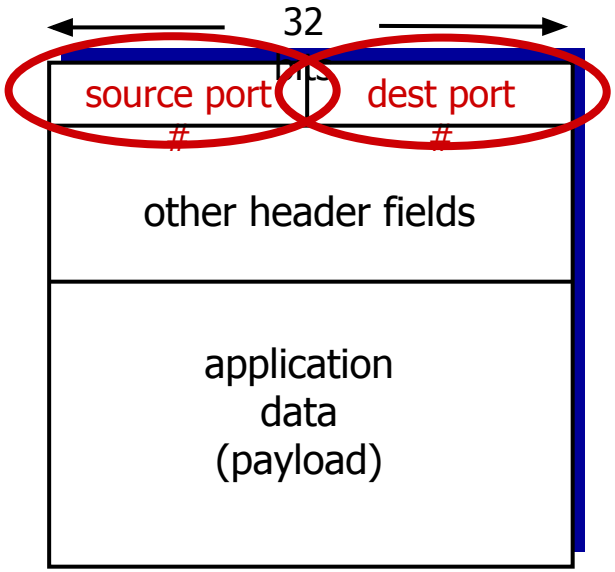


multiplexing


Multiplexing

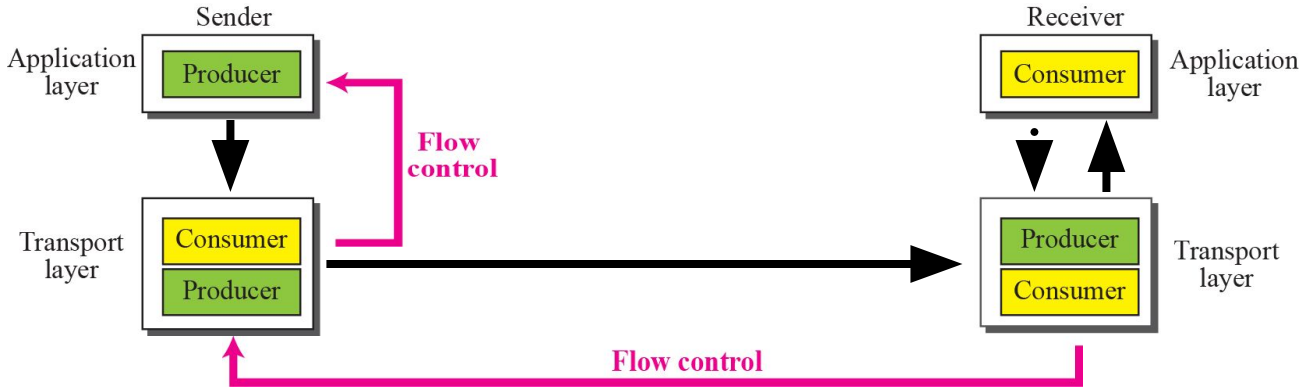
How demultiplexing works

- 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

Flow control



Error Control

1. **Checksum**
2. **Acknowledgement**
3. **Retransmission**

Error Control Service is responsible for:

1. Detecting and discarding corrupted packets.
2. Keeping track of lost and discarded packets and resending them
3. Recognizing duplicate packets and discarding them.
4. Buffering Out-of-Order packets until the missing packets arrive.

Flow Control and Error Control Protocol

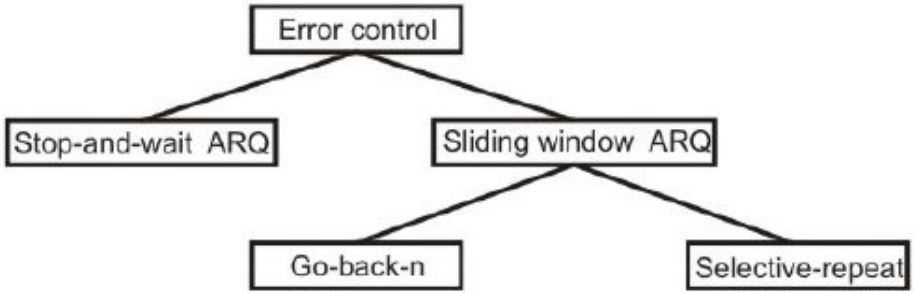
- ✔ Simple Protocol
- ✔ Stop-and-Wait Protocol
- ✔ Go-Back-*N* Protocol
- ✔ Selective-Repeat Protocol

Flow Control and Error Control Protocol Cont..

Flow control: It coordinates amount of data that can be sent before receiving an ack.

- Stop and Wait
- Sliding window

Error Control: It is refer to the methods of error detection and retransmission. The most popular retransmission scheme is known as Automatic-Repeat-Request (ARQ). Three popular ARQ techniques



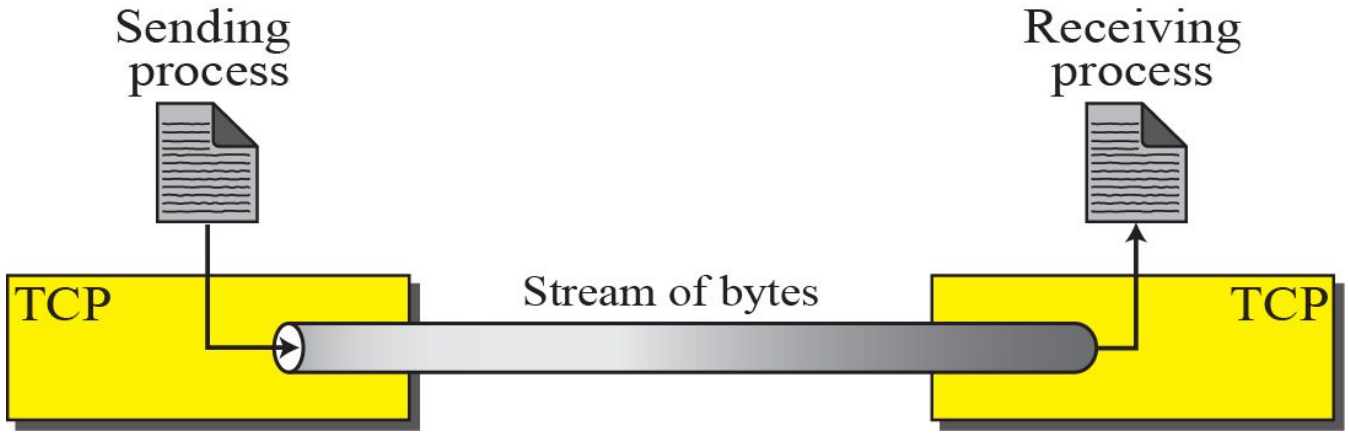
Chapter 3: roadmap

- Transport-layer services
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Connectionless transport: UDP
- Principles of congestion control
- TCP congestion control

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

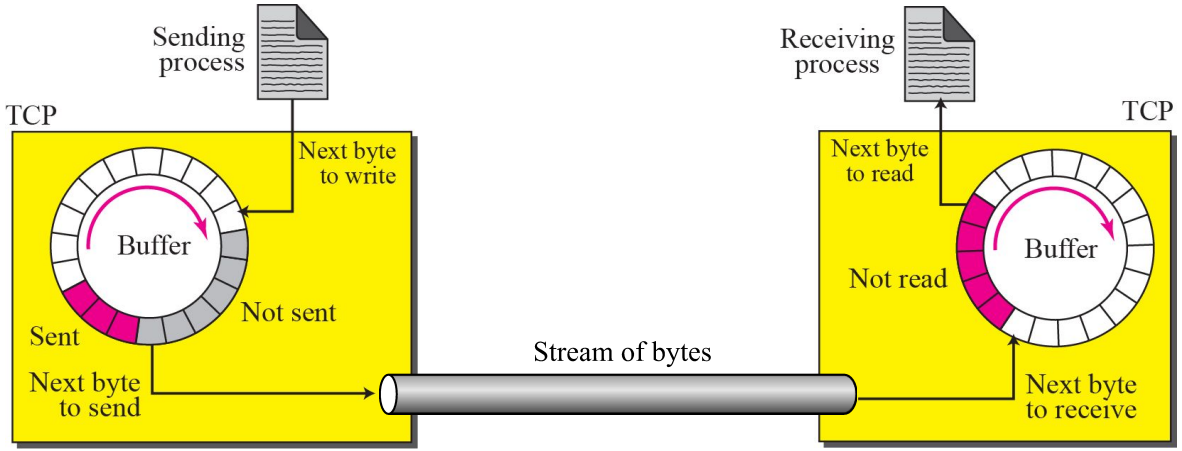
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

Stream delivery

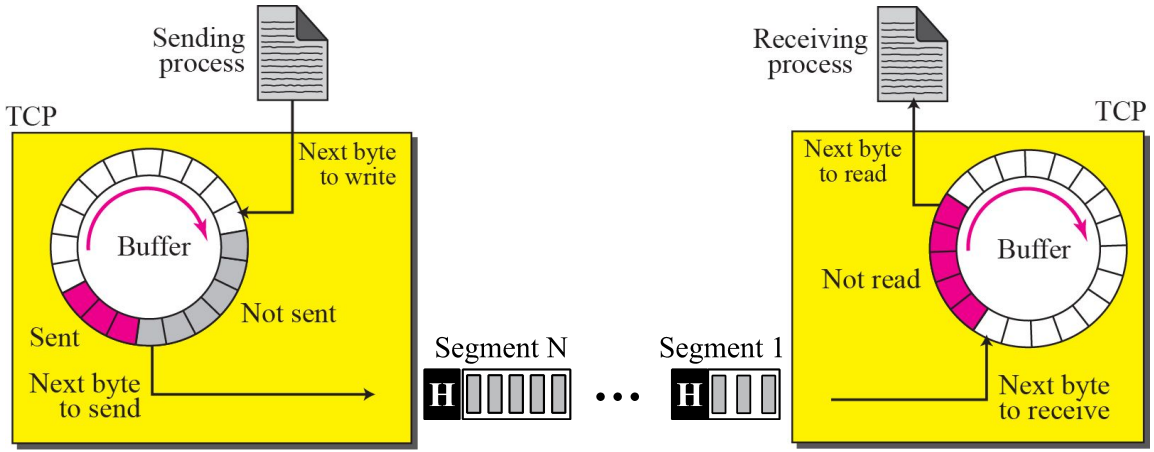


- TCP is byte stream protocol. Every byte that is being sent is actually cunted.

Sending and receiving buffers



TCP segments



TCP FEATURES

- TCP has several features

✔Numbering System

✔ Flow Control

✔ Error Control

✔ Congestion Control

TCP sequence numbers, ACKs

Sequence numbers:

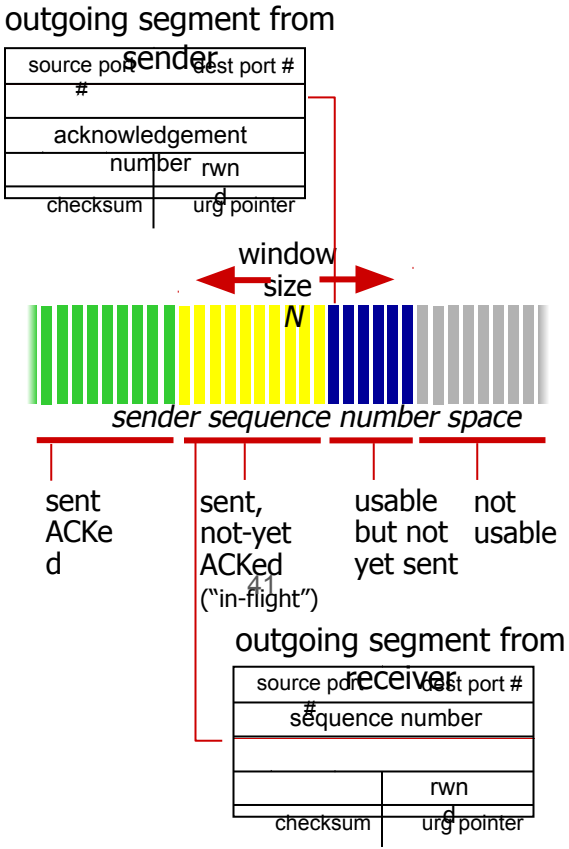
- byte stream "number" of first byte in segment's data

Acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor

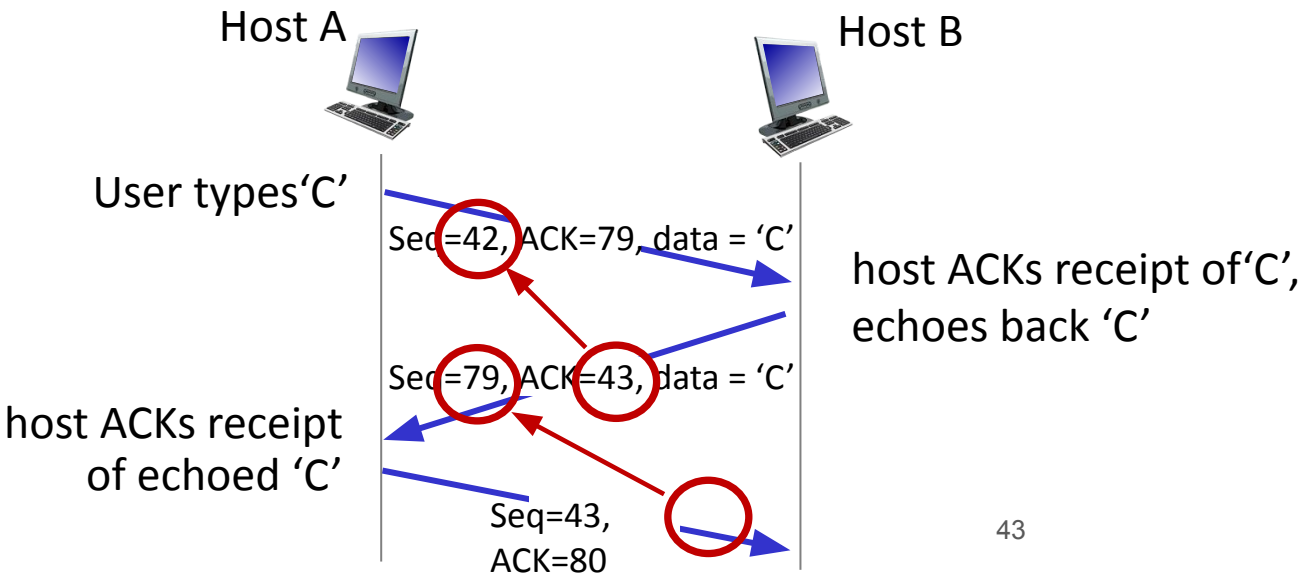


TCP sequence numbers

The bytes of data being transferred in each connection are numbered by TCP.
The numbering starts with a randomly generated number.

- Both sides of TCP connection randomly choose an initial sequence number.
- This is done to minimize the possibility that a segment that is still present in the network from an earlier, already terminated connection between two hosts is mistake for a valid segment in a later connection between these same two hosts.

TCP sequence numbers, ACKs



simple telnet scenario

Numbering System

- The bytes of data being transferred in each connection are numbered by TCP.
- The numbering starts with an arbitrarily generated number.
- Suppose a TCP connection is transferring a file of 5,000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1,000 bytes?

Solution

The following shows the sequence number for each segment:

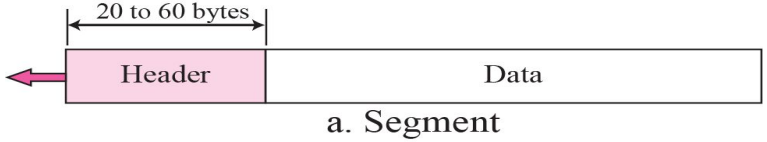
Segment 1	→	**Sequence Number:**	**10,001**	**Range:**	**10,001**	to	**11,000**
Segment 2	→	**Sequence Number:**	**11,001**	**Range:**	**11,001**	to	**12,000**
Segment 3	→	**Sequence Number:**	**12,001**	**Range:**	**12,001**	to	**13,000**
Segment 4	→	**Sequence Number:**	**13,001**	**Range:**	**13,001**	to	**14,000**
Segment 5	→	**Sequence Number:**	**14,001**	**Range:**	**14,001**	to	**15,000**

Cont..

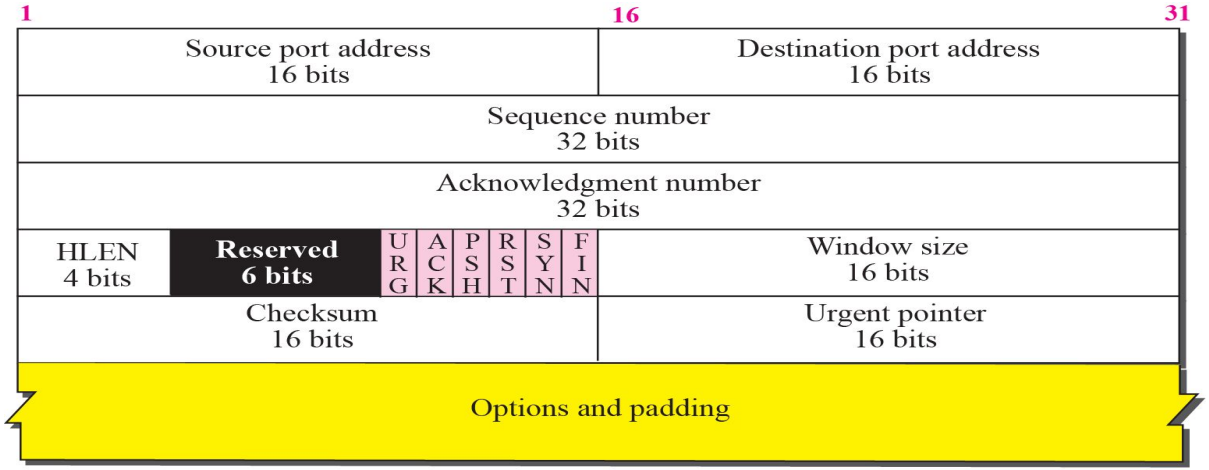
- The value in the sequence number field of a segment defines the number assigned to the **first data byte contained in that segment**.
- The value of the acknowledgment field in a segment defines the **number of the next byte** a party expects to receive.
- The acknowledgment number is **cumulative**.

TCP segment format

- Before discussing TCP in more detail, let us discuss the TCP packets themselves. A packet in TCP is called a segment.

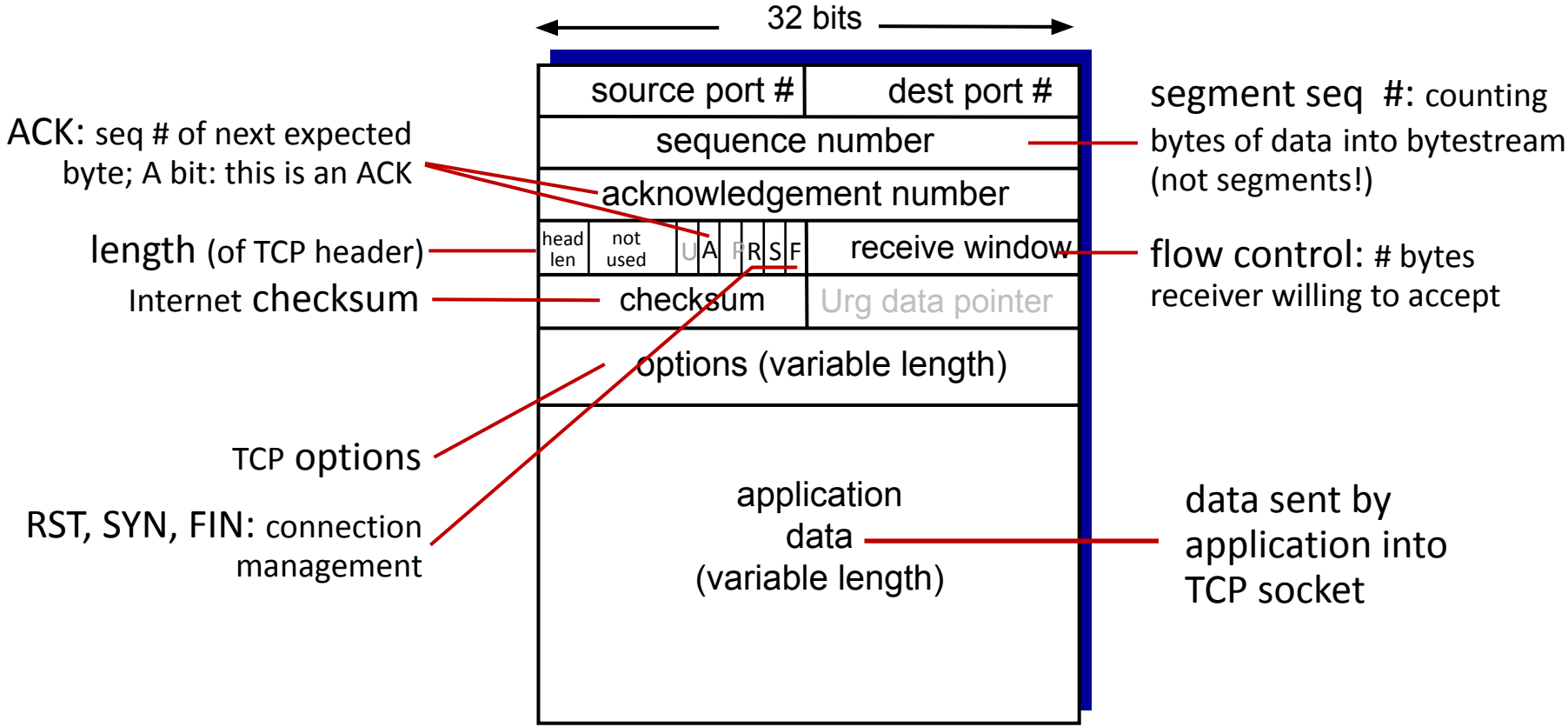


a. Segment



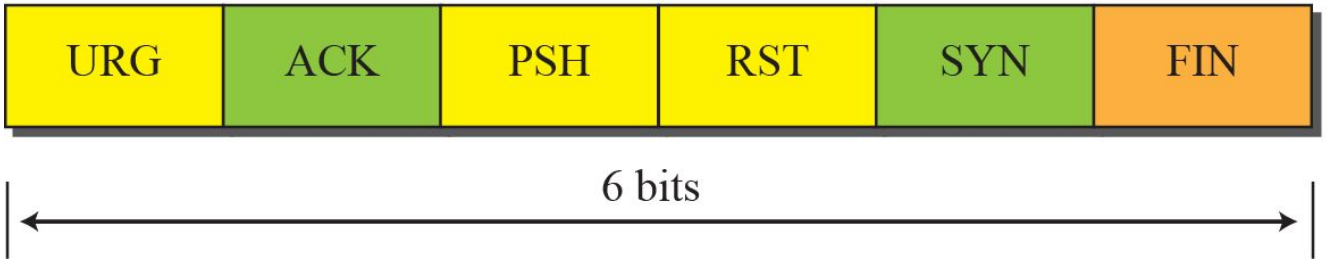
b. Header

TCP segment structure



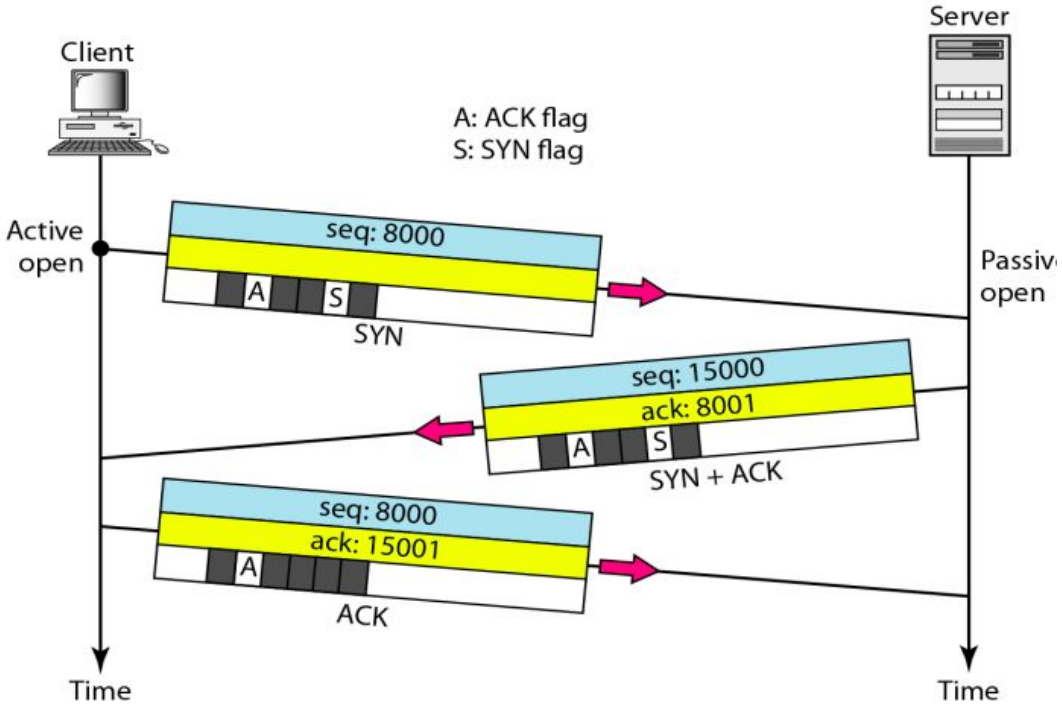
TCP Flag Bits

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

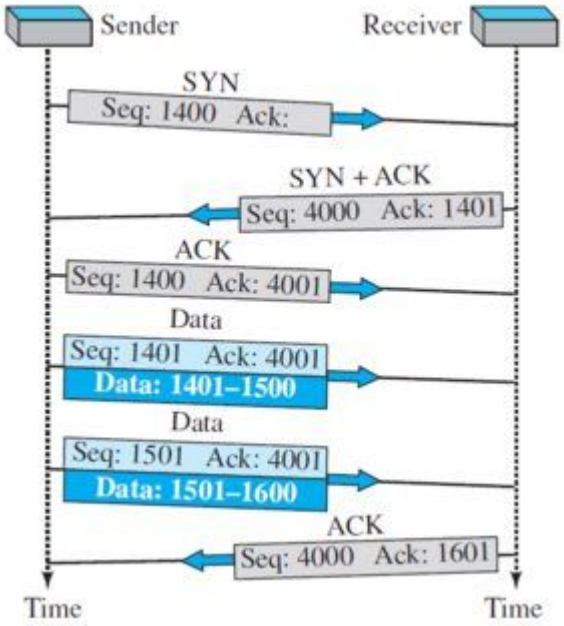


In practice, the PSH, URG, and the urgent pointer are not used.

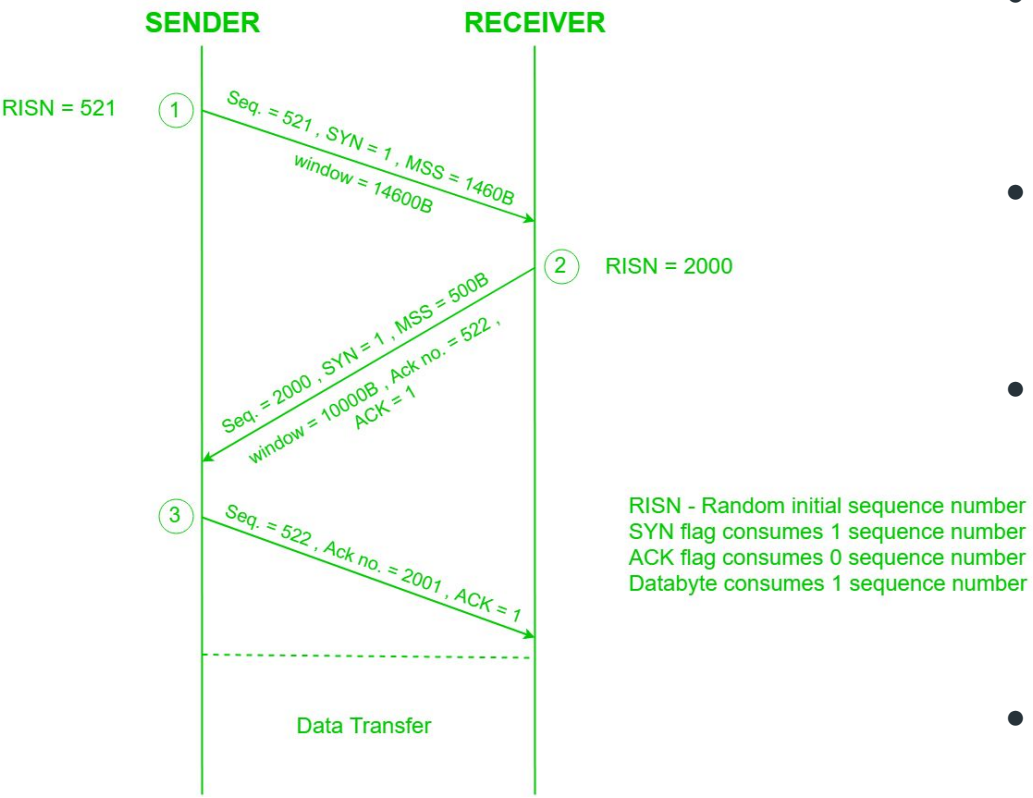
Connection establishment using three-way handshaking



- A SYN segment cannot carry data, but it consumes one sequence number.
- A SYN + ACK segment cannot carry data, but does consume one sequence number.
- **An ACK segment, if carrying no data, consumes no sequence number.**


Sender
Receiver
SYN
Seq: 1400 Ack:
SYN + ACK
Seq: 4000 Ack: 1401
ACK
Seq: 1400 Ack: 4001
Data
Seq: 1401 Ack: 4001
Data: 1401–1500
Data
Seq: 1501 Ack: 4001
Data: 1501–1600
ACK
Seq: 4000 Ack: 1601
Time
Time

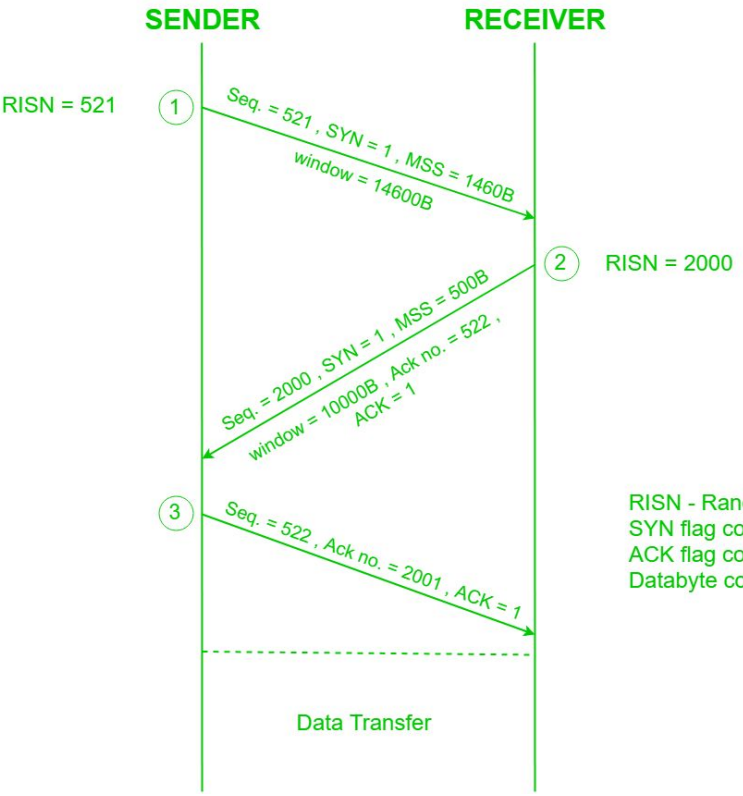
Connection establishment using three-way handshaking



https://www.geeksforgeeks.org/tcp-connection-establishment/

1. Sender starts the process with the following:
   - **Sequence number (Seq=521):** contains the random initial sequence number generated at the sender side.
   - **Syn flag (Syn=1):** request the receiver to synchronize its sequence number with the above-provided sequence number.
   - **Maximum segment size (MSS=1460 B):** sender tells its maximum segment size, so that receiver sends datagram which won't require any fragmentation. MSS field is present inside **Option** field in TCP header.
   - **Window size (window=14600 B):** sender tells about his buffer capacity in which he has to store messages from the receiver.

Connection establishment using three-way handshaking



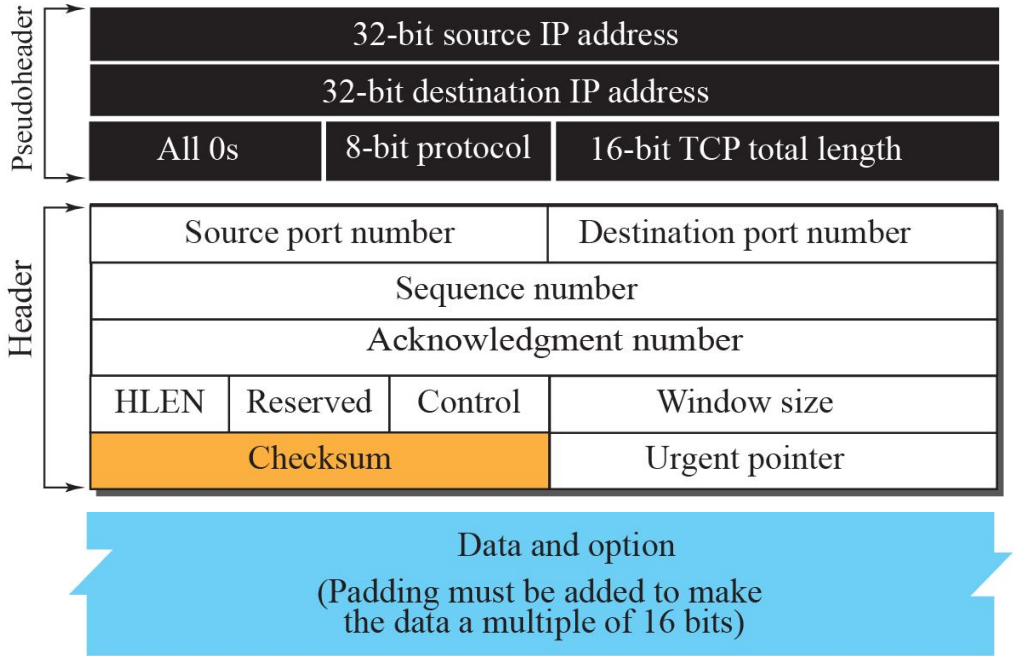
RISN - Random initial sequenc
SYN flag consumes 1 sequence number
ACK flag consumes 0 sequenc
Databyte consumes 1 sequenc

2. TCP is a full-duplex protocol so both sender and receiver require a window for receiving messages from one another.

- **Sequence number (Seq=2000):** contains the random initial sequence number generated at the receiver side.
- **Syn flag (Syn=1):** request the sender to synchronize its sequence number with the above-provided sequence number.
- **Maximum segment size (MSS=500 B):** receiver tells its maximum segment size, so that sender sends datagram which won't require any fragmentation.
- MSS field is present inside **Option** field in TCP header.
- Since $MSS_{receiver} < MSS_{sender}$, both parties agree for minimum MSS i.e., 500 B to avoid fragmentation of packets at both ends.

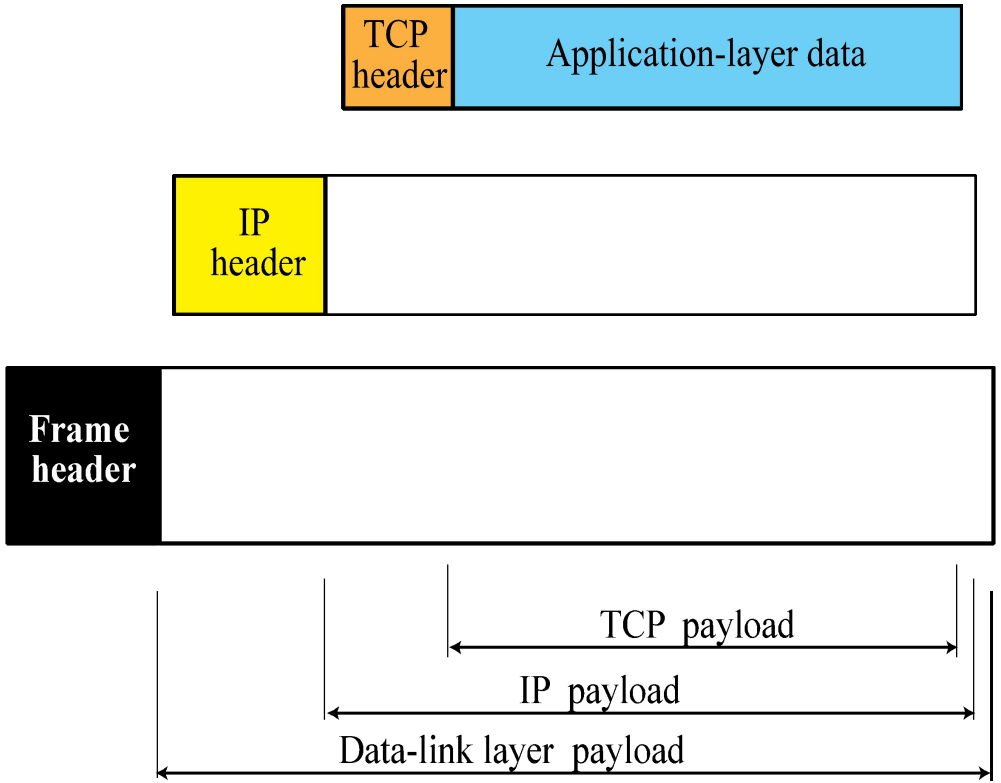
```
Therefore, receiver can send maximum of 14600/500 = 29 packets. This is the receiver's sending window size.
```

Pseudoheader added to the TCP segment



The use of the checksum in TCP is mandatory.

Encapsulation



TCP/IP Protocol Suite

Rules for Generating the ACKs

1. When one end sends a data segment to the other end, it must include an ACK. That gives the next sequence number it expects to receive. (Piggyback)

2. The receiver needs to delay sending (until another segment arrives or 500ms) an ACK segment if there is only one outstanding in-order segment. It prevents ACK segments from creating extra traffic.

3. There should not be more than 2 in-order unacknowledged segments at any time. It prevent the unnecessary retransmission

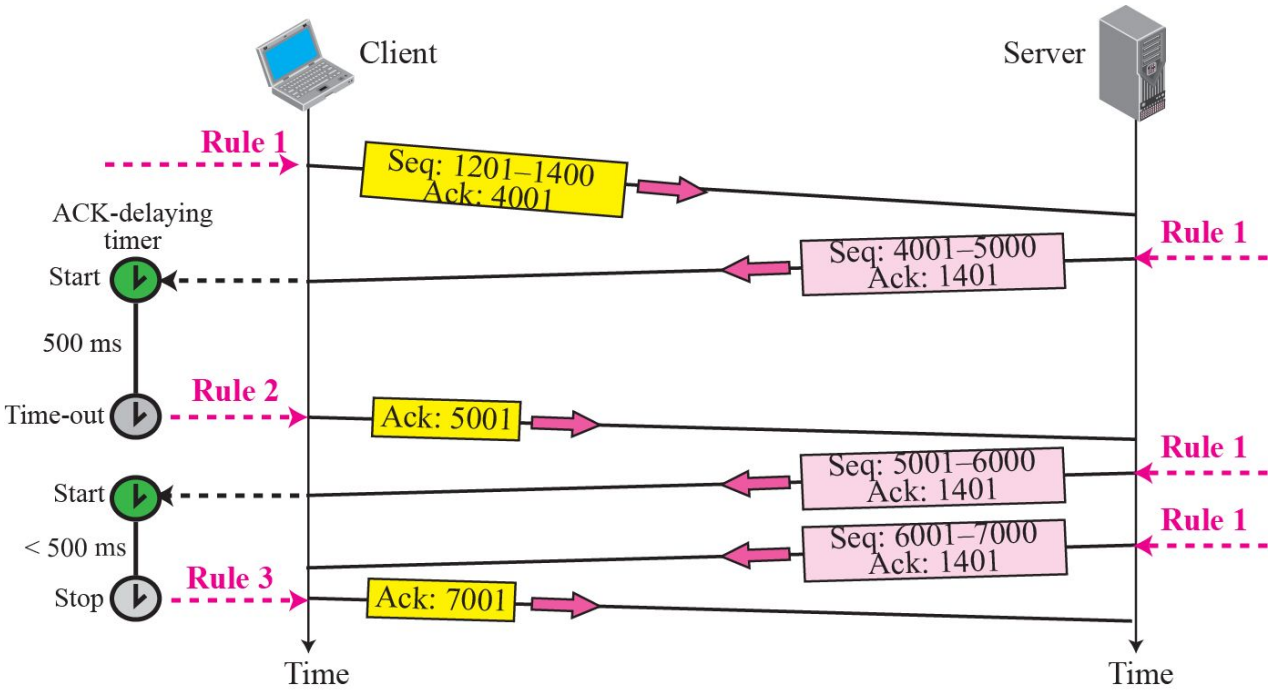
Rules for Generating the ACKs Cont..

4. When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected (not received) segment. (**for fast retransmission**)

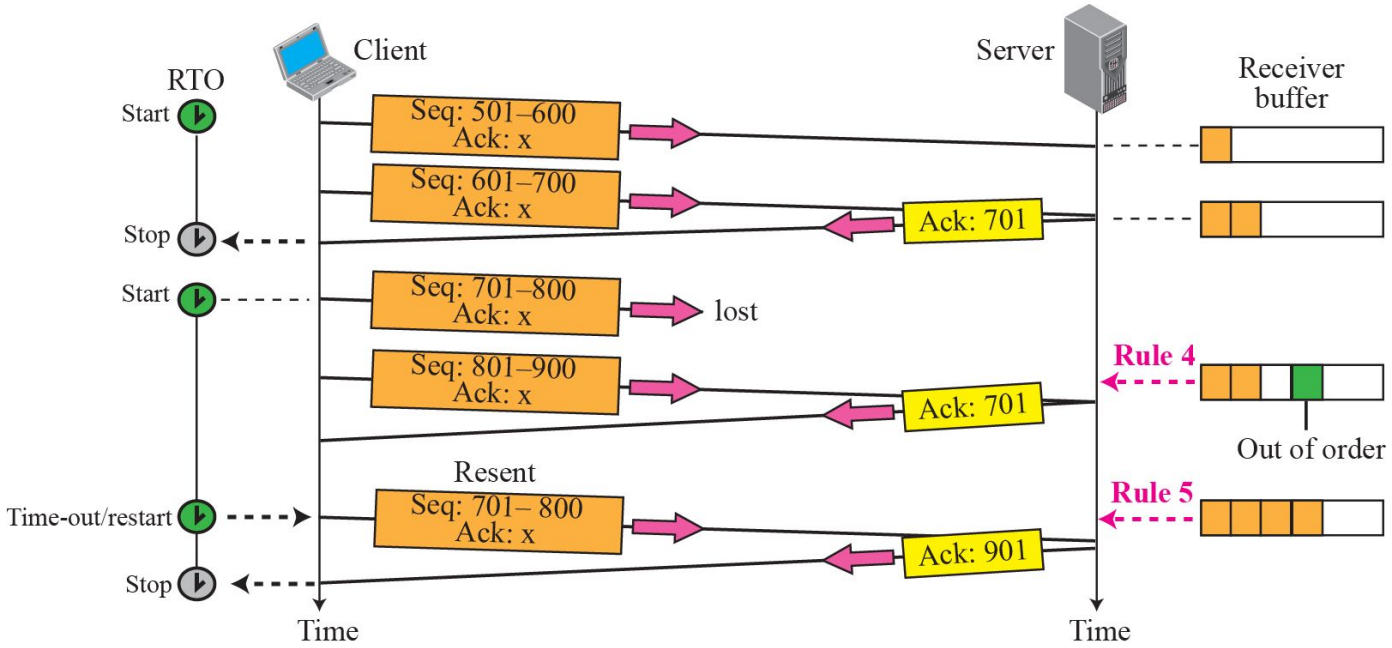
5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected.

6. If a duplicate segment arrives, the receiver immediately sends an ACK.

Some Scenarios: Normal operation



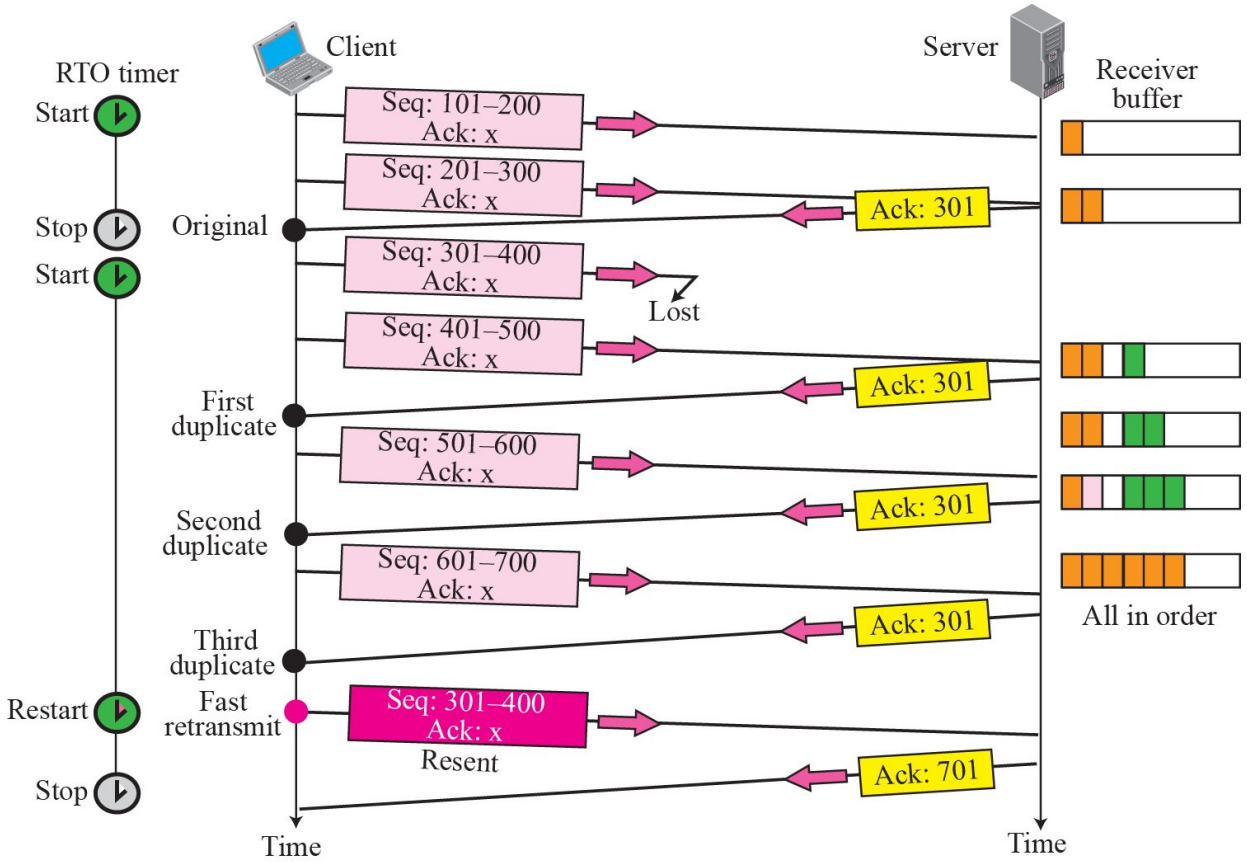
Lost segment



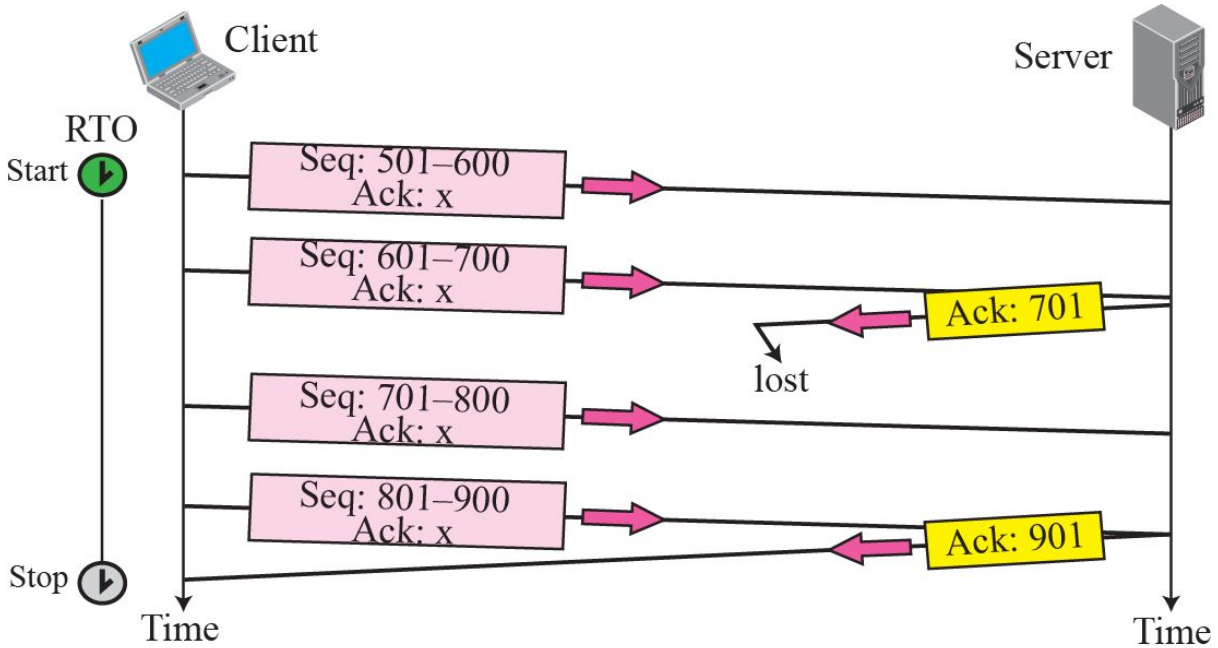
The receiver TCP delivers only ordered data to the process.

Fast retransmission

- Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!
- Retransmission after 3 duplicates Acknowledgement (or) early Retransmission

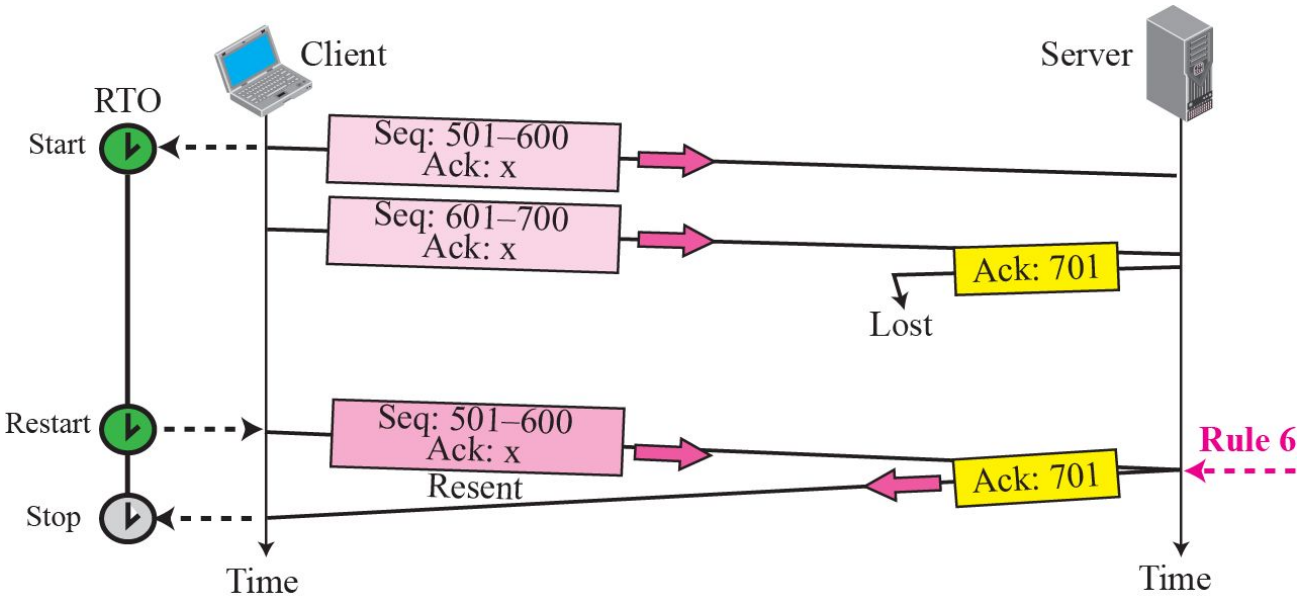


Lost acknowledgment



Lost acknowledgment corrected by cumulative ack

Lost acknowledgment corrected by resending a segment



ACK and Out of Order Handling in TCP

Acknowledgement in TCP – Cumulative acknowledgement

Receiver has received bytes 0, 1, 2, _, 4, 5, 6, 7

- TCP sends a cumulative acknowledgement with ACK number 3, acknowledging everything up to byte 2
- Once 4 is received, a duplicate ACK with ACK number 3 (next expected byte) is forwarded
- After timeout, sender retransmits byte 3
- Once byte 3 is received, it can send another cumulative ACK with ACK number 8 (next expected byte)

Window size or Advertisement Window

- A sender should never send more than what the receiver receives.
- During connection establishment phase, both sides advertise window size.
- Importance of persistence timer (PT).

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

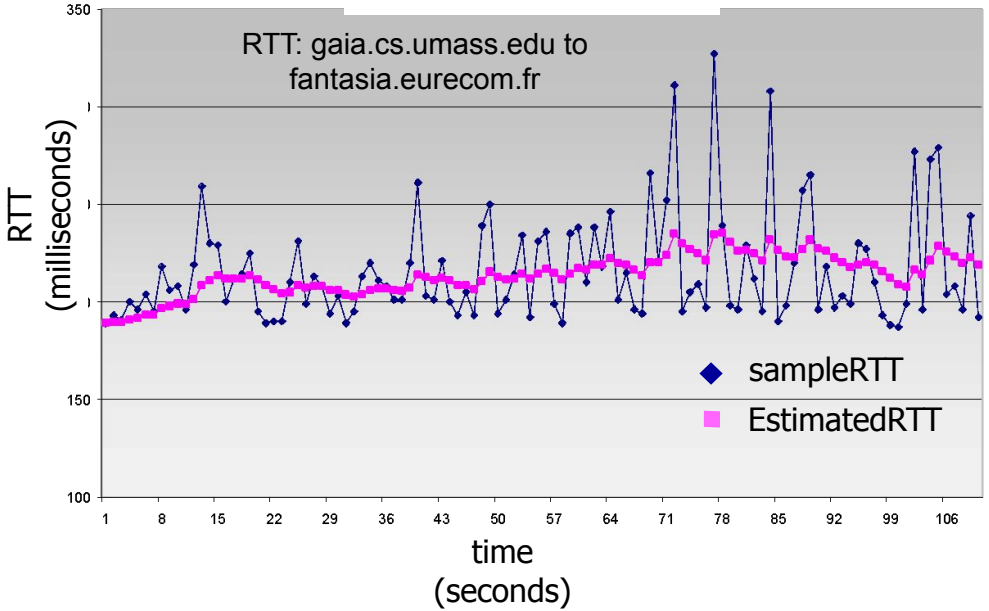
Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

TCP round trip time, timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT:`** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

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

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Principles of congestion control

Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)

congestion control: too many senders, sending too fast

- different from flow control!

flow control: one sender too fast for one receiver

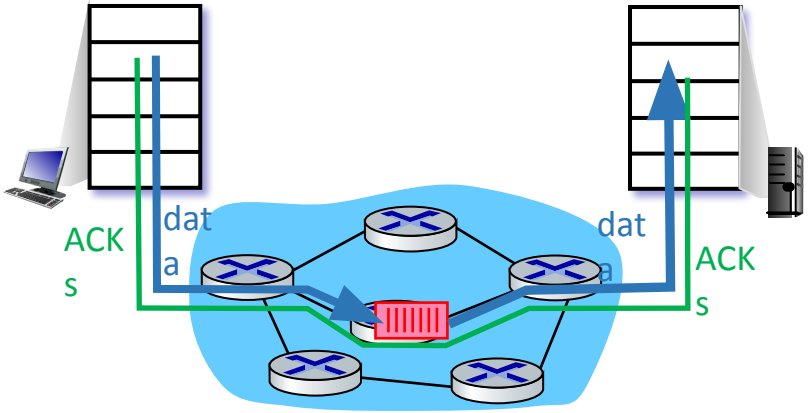
Congestion control

- Congestion in a network may occur if the load on the network—the number of packets sent to the network—is greater than the capacity of the network—the number of packets a network can handle.
- Congestion control refers to the mechanisms and techniques to control the congestion and keep the load below the capacity.

Approaches towards congestion control

End-end congestion control:

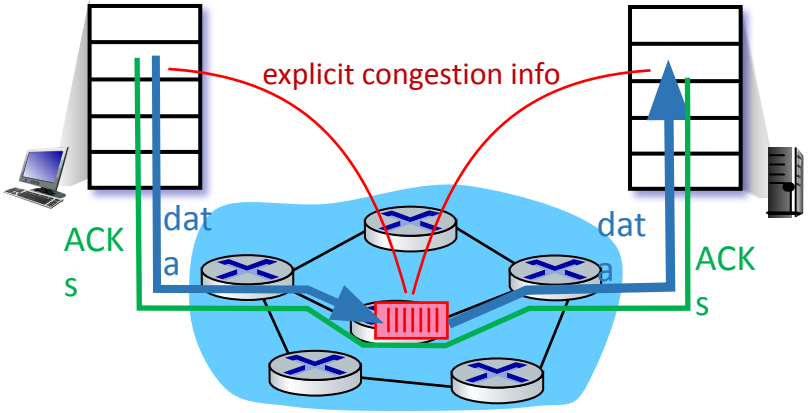
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
  - Approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate



- TCP ECN (Explicit Congestion Notification)

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

TCP: Triggering congestion control

- Two ways to trigger a congestion notification in TCP – (1) RTO, (2) Duplicate ACK
- RTO: A sure indication of congestion, however time consuming
- Duplicate ACK: Receiver sends a duplicate ACK when it receives out of order segment
  - A loose way of indicating congestion
  - TCP arbitrarily assumes that THREE duplicate ACKs (DUPACKs) imply that a packet has been lost – triggers congestion control mechanism
  - The identity of the lost packet can be inferred – the very next packet in sequence
  - Retransmit the lost packet and trigger congestion control

TCP congestion control: AIMD

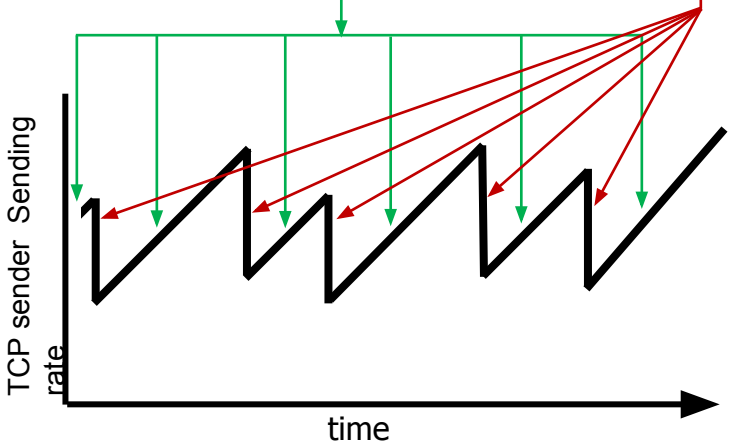
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event.

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



- Chiu and Jain (1989): Let *w(t)* be the sending rate. *a (a > 0)* is the additive increase factor, and *b (0<b<1)* is the multiplicative decrease factor

$$w(t+1) = \begin{cases} w(t) + a & \text{if congestion is not detected} \\ w(t) \times b & \text{if congestion is detected} \end{cases}$$

TCP AIMD: more

Multiplicative decrease detail: sending rate is

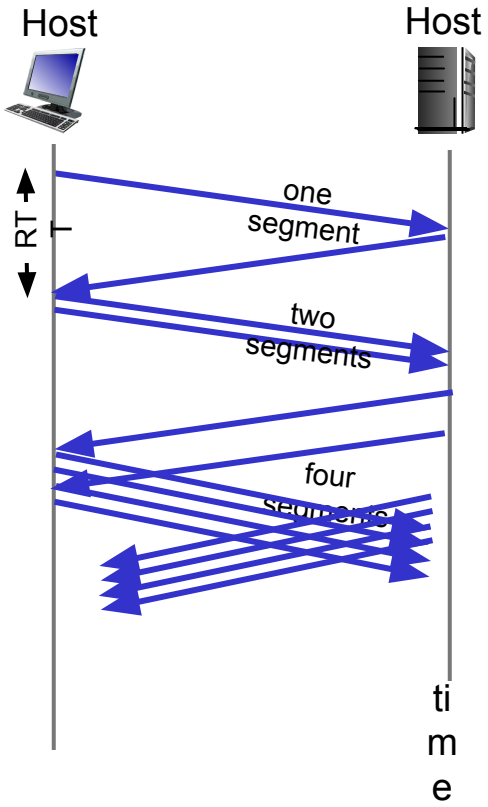
- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

TCP Congestion Control

- TCP maintains a **Congestion Window (CWnd) –** number of bytes the sender may have in the network at any time
- **Sender Window (SWnd) = Min (CWnd, RWnd)**
- RWnd – Receiver advertised window size

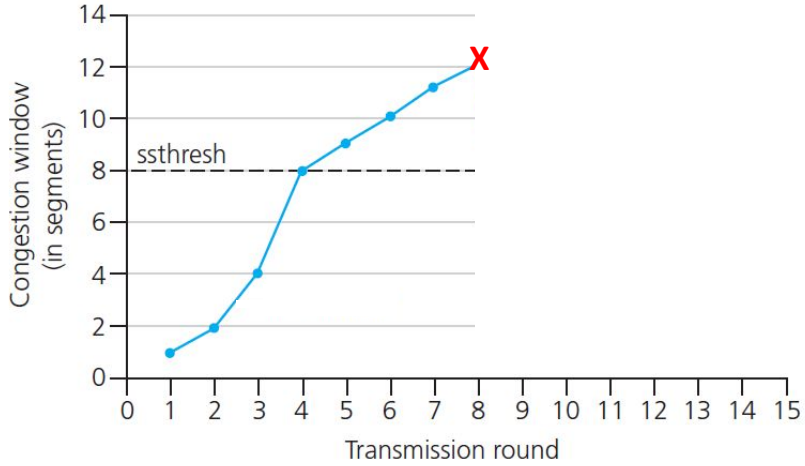
TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: from slow start to congestion avoidance

The sender has two parameters for congestion control:
- **Congestion Window** (***cwnd***; Initial value is MSS bytes)
- **Threshhold Value** (***ssthresh***; Initial value is 65536 bytes)



Implementation:

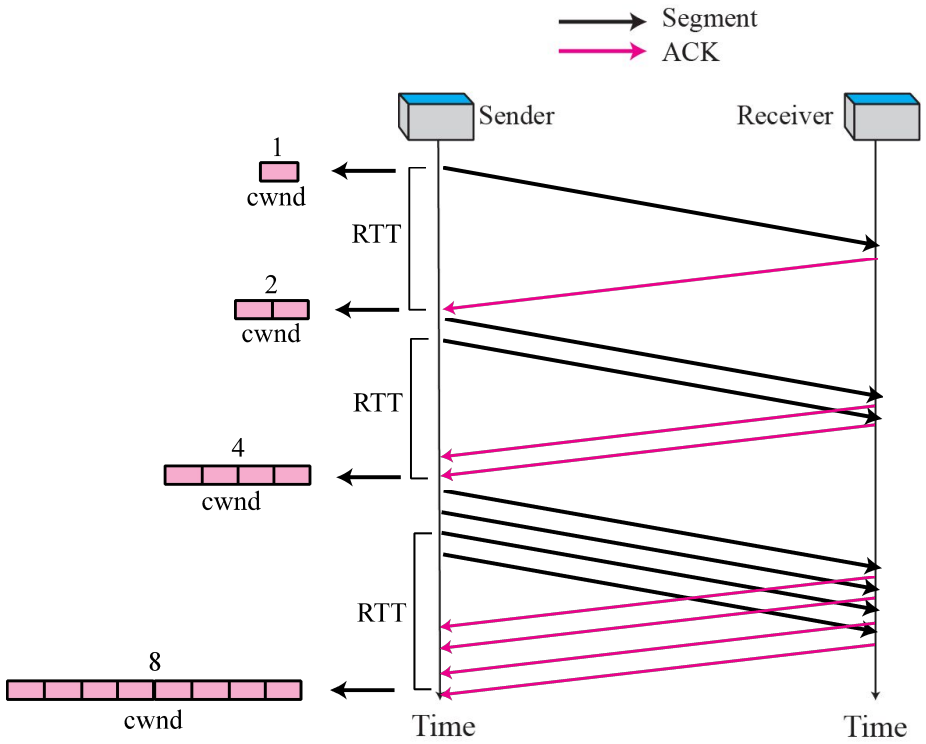
- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

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

Slow Start Cont..

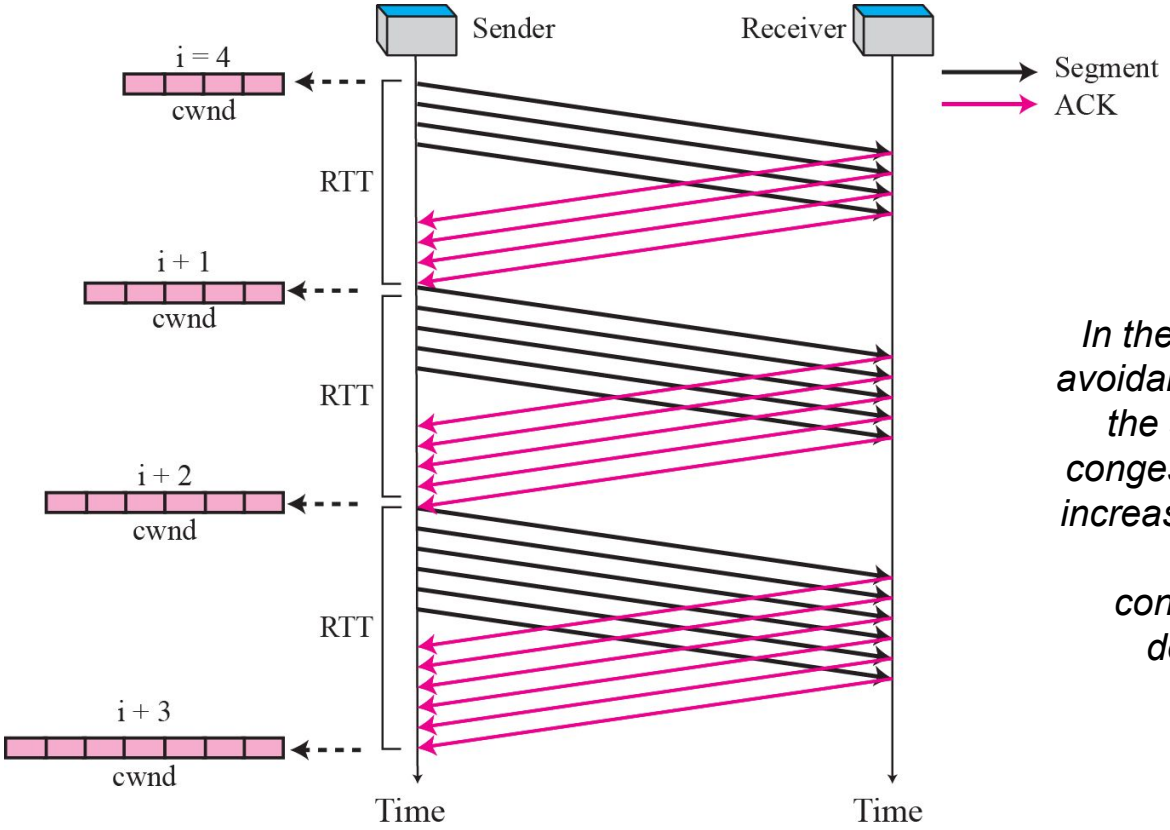
- Slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssthresh).**
- Whenever a packet loss is detected by a timeout, the ssthresh is set to be half of the congestion window

Congestion Control: Slow start, exponential increase



In the slow start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.

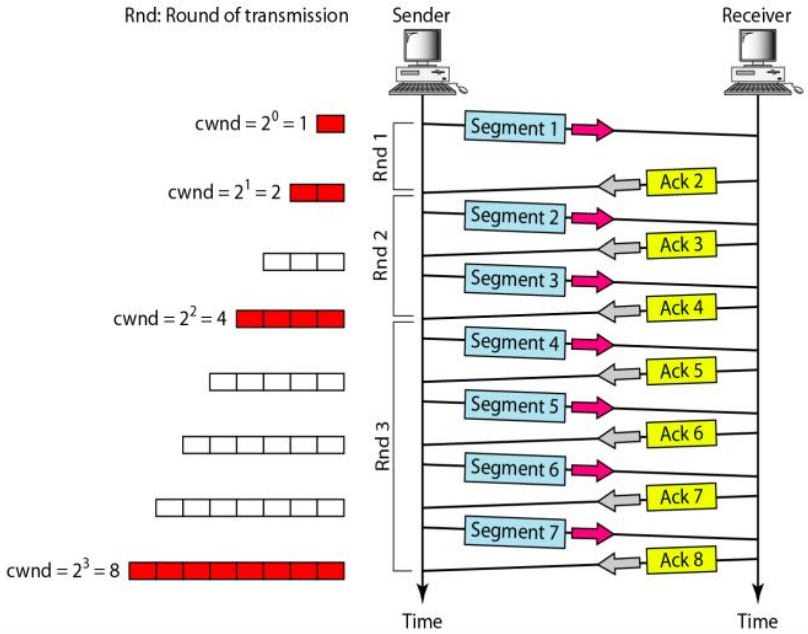
Congestion avoidance, additive increase



In the congestion avoidance algorithm the size of the congestion window increases additively until congestion is detected.

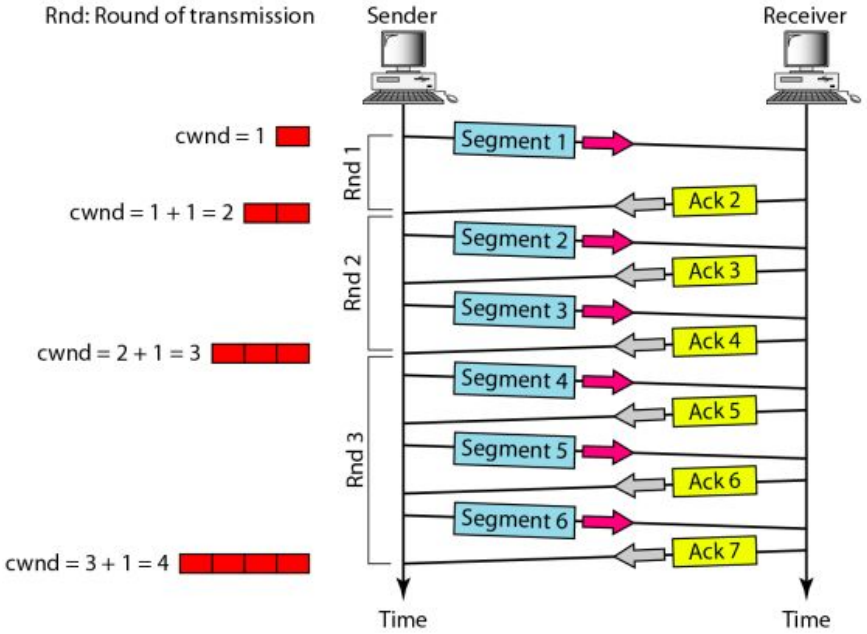
Slow start, exponential increase

Slow start, exponential increase



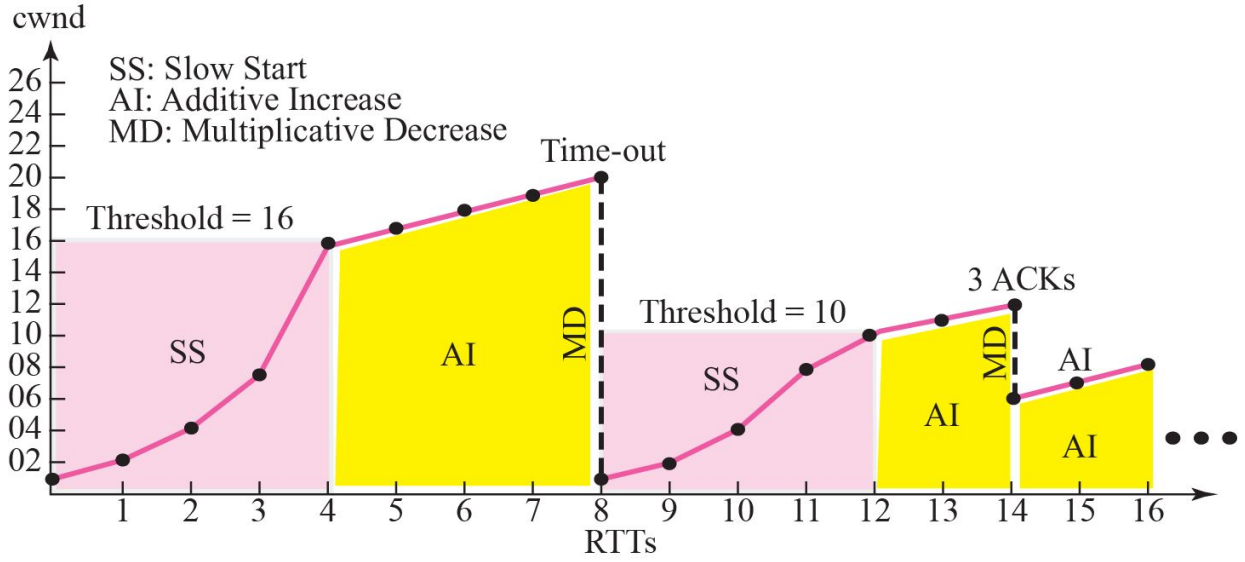
In the slow-start algorithm, the size of the congestion window starts with one Max Seg. Size and increases exponentially until it reaches a threshold.

Congestion avoidance, additive increase



In the congestion avoidance algorithm, the size of the congestion window increases additively until congestion is detected.

Congestion example



Fast Retransmission - TCP Tahoe

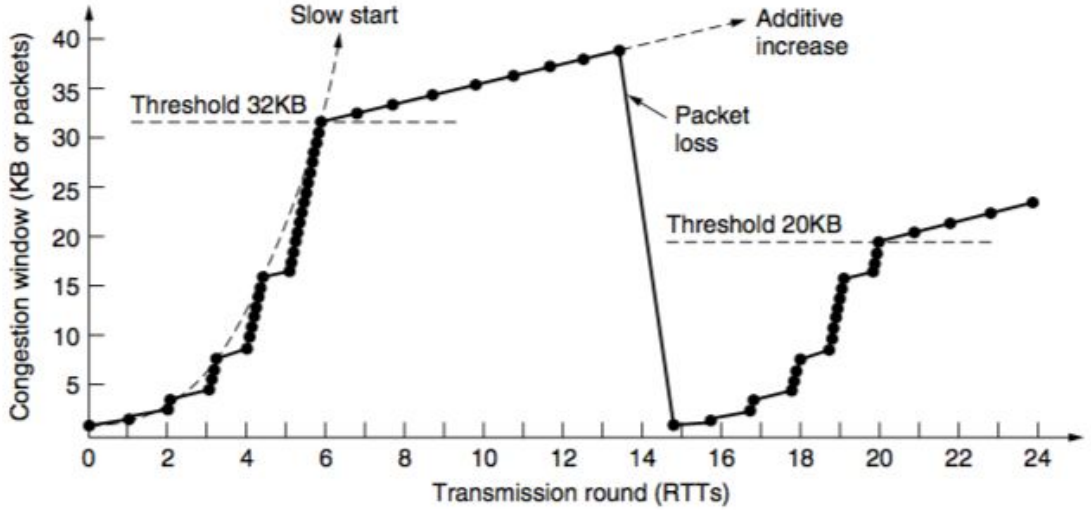
Use THREE DUPACK as the sign of congestion

Once 3 DUPACKs have been received,

Retransmit the lost packet (**fast retransmission)**

Set ssthresh as half of the current CWnd

Set CWnd to 1 MSS



Fast Recovery – TCP Reno

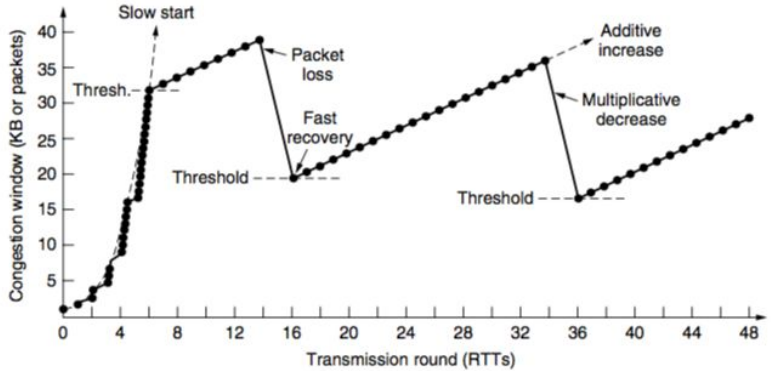
- **Once a congestion is detected through 3 DUPACKs, do TCP really need to set CWnd = 1 MSS ?**

- DUPACK means that **some segments are still flowing in the network** – a signal for temporary congestion, but not a prolonged one

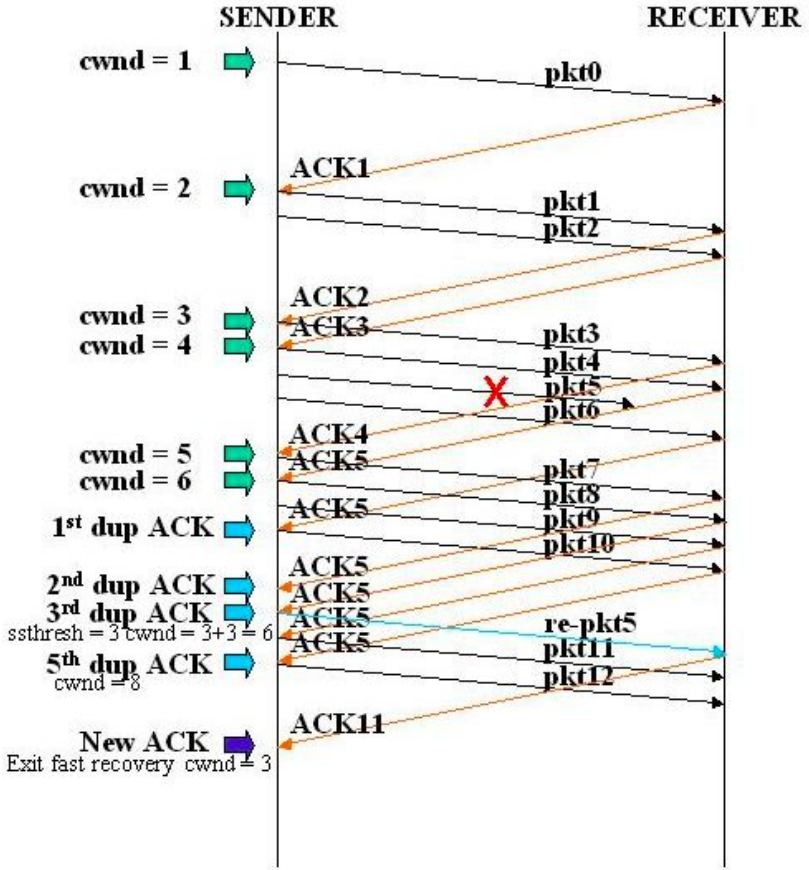
- Immediately transmit the lost segment (**fast retransmit)**, then transmit additional segments based on the DUPACKs received **(fast recovery)**

Fast Recovery – TCP Reno

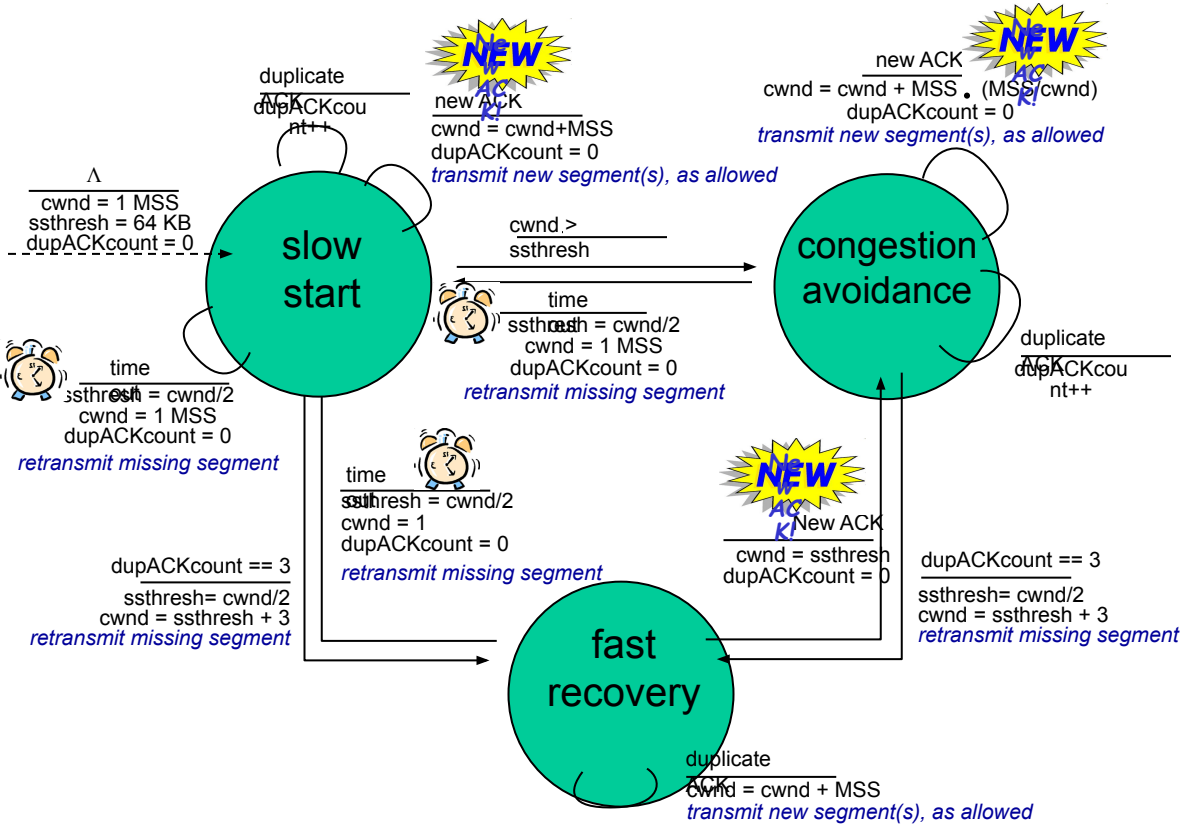
- **Fast recovery:**
- set ssthresh to half of the current congestion window. Retransmit the missing segment.
- set cwnd = ssthresh + 3.
- Each time another duplicate ACK arrives, set cwnd = cwnd + 1. Then, send a new data segment if allowed by the value of cwnd.
- Once receive a new ACK (an ACK which acknowledges all intermediate segments sent between the lost packet and the receipt of the first duplicate ACK), exit fast recovery. This causes setting cwnd to ssthresh (the ssthresh in step 1). Then, continue with linear increasing due to congestion avoidance algorithm.



Example: Fast Recovery – TCP Reno



Summary: TCP congestion control



TCP Congestion Control Algorithms

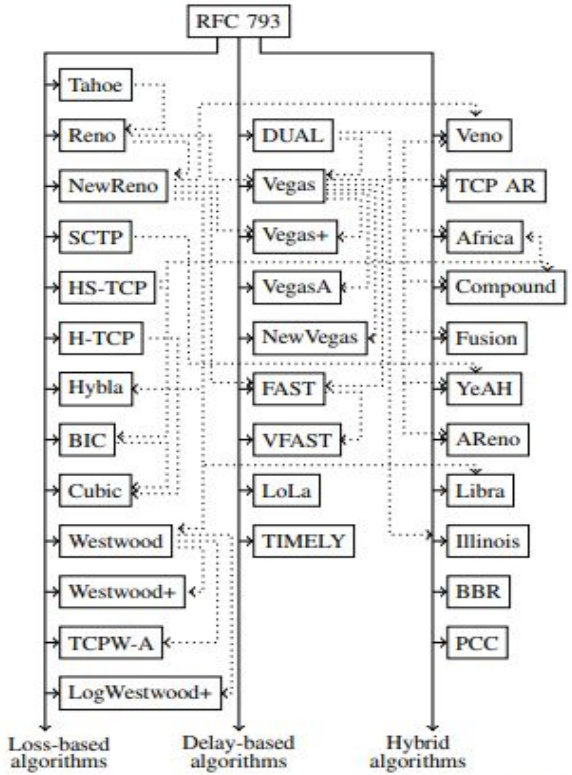


Fig. 3: Classification of different congestion control algorithms. Dotted arrows indicate that one was based on the other.

- TCP Cubic: Used by Many Linux systems

Chapter 3: roadmap

- Transport-layer services
- **Connectionless transport: UDP**
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

UDP: User Datagram Protocol

- Simple and quick 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

Why is there a UDP?

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

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

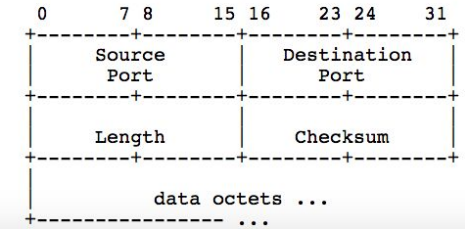
User Datagram Protocol

Introduction

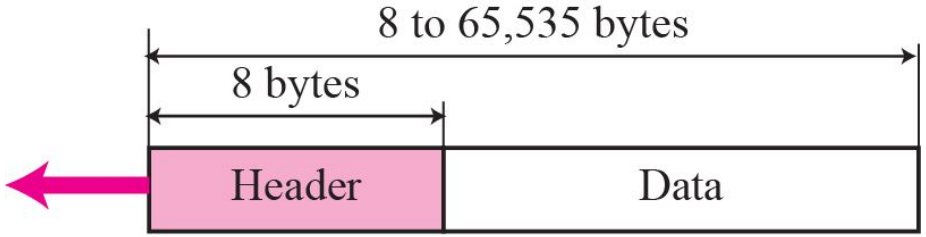
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

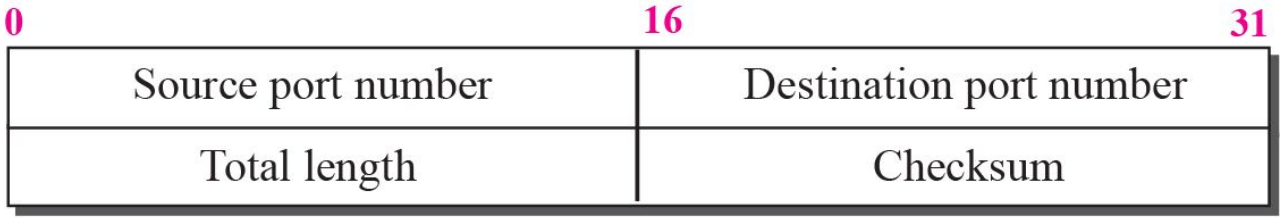
Format



UDP Header



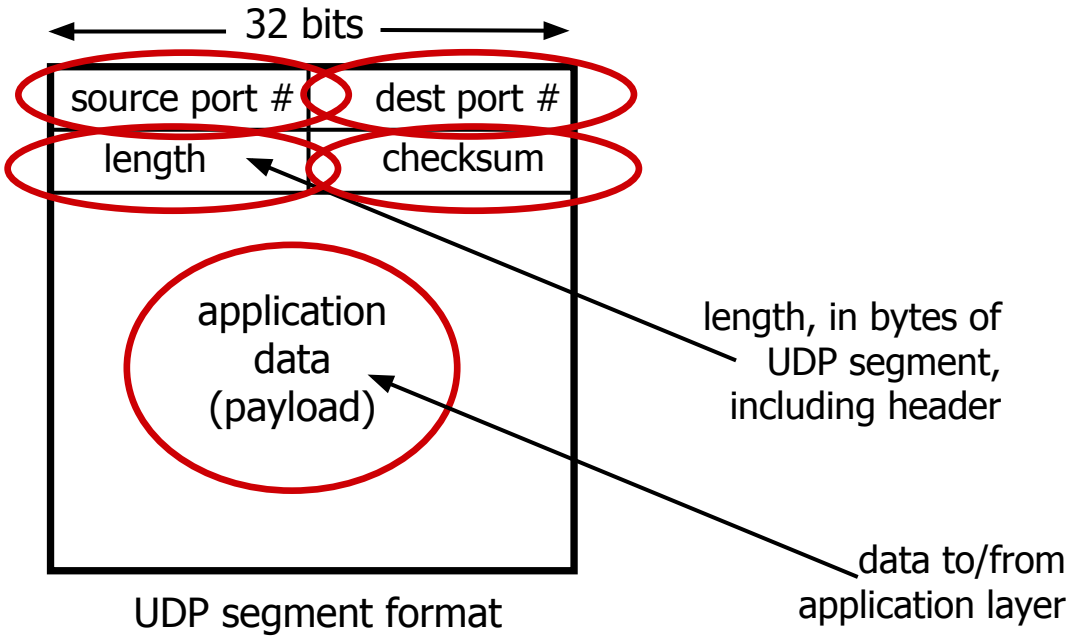
a. UDP user datagram



b. Header format

Source: TCP/IP Protocol Suite

UDP segment header cont..



Questions

The following is a dump of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?

Answers

Solution

a. The source port number is the first four hexadecimal digits $(CB84)_{16}$ or 52100.

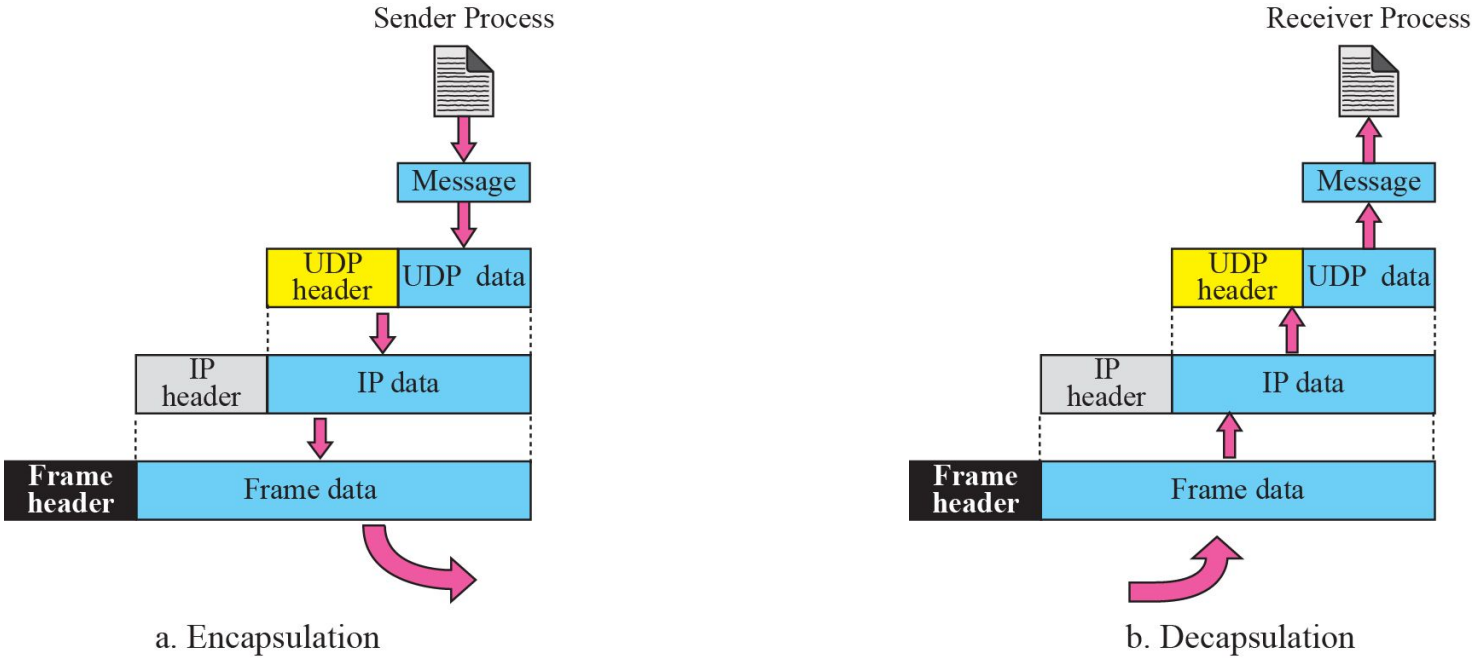
b. The destination port number is the second four hexadecimal digits $(000D)_{16}$ or 13.

c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.

d. The length of the data is the length of the whole packet minus the length of the header, or 28 – 8 = 20 bytes.

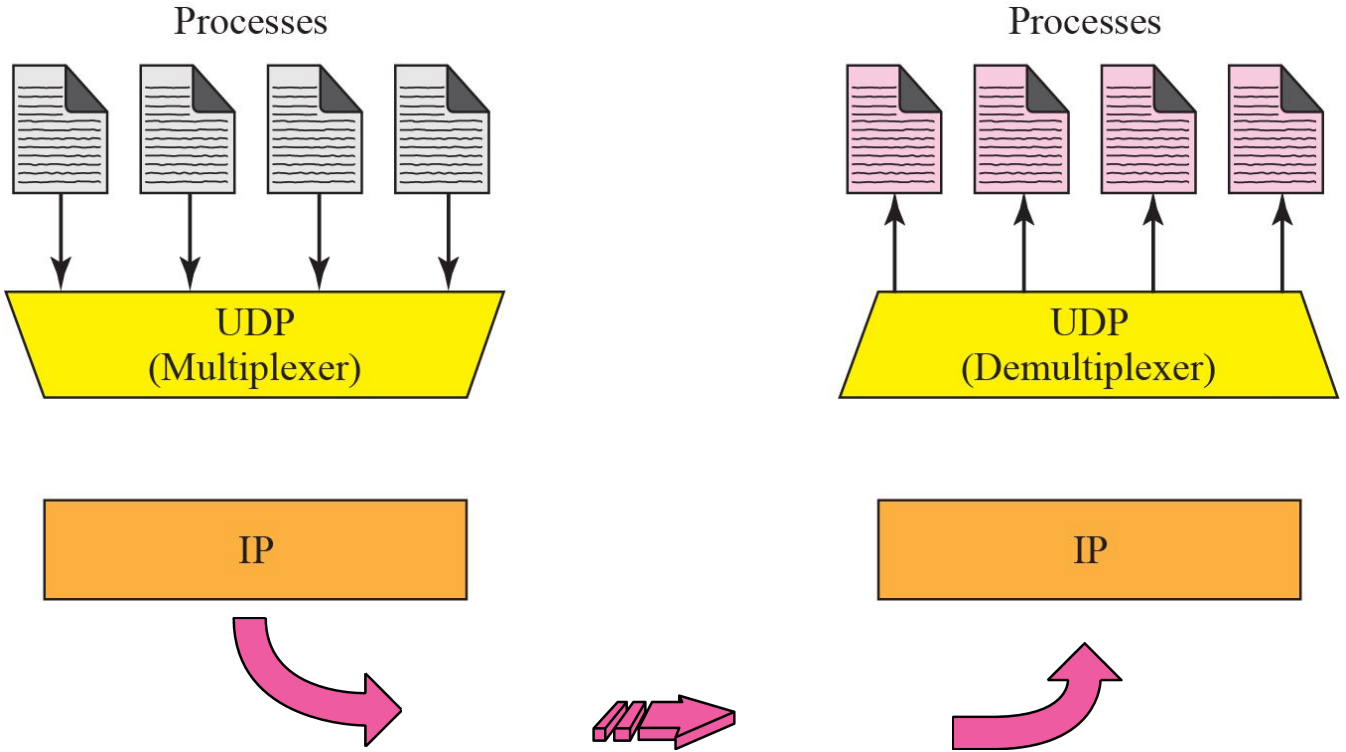
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.

Encapsulation and decapsulation



Source: TCP/IP Protocol Suite

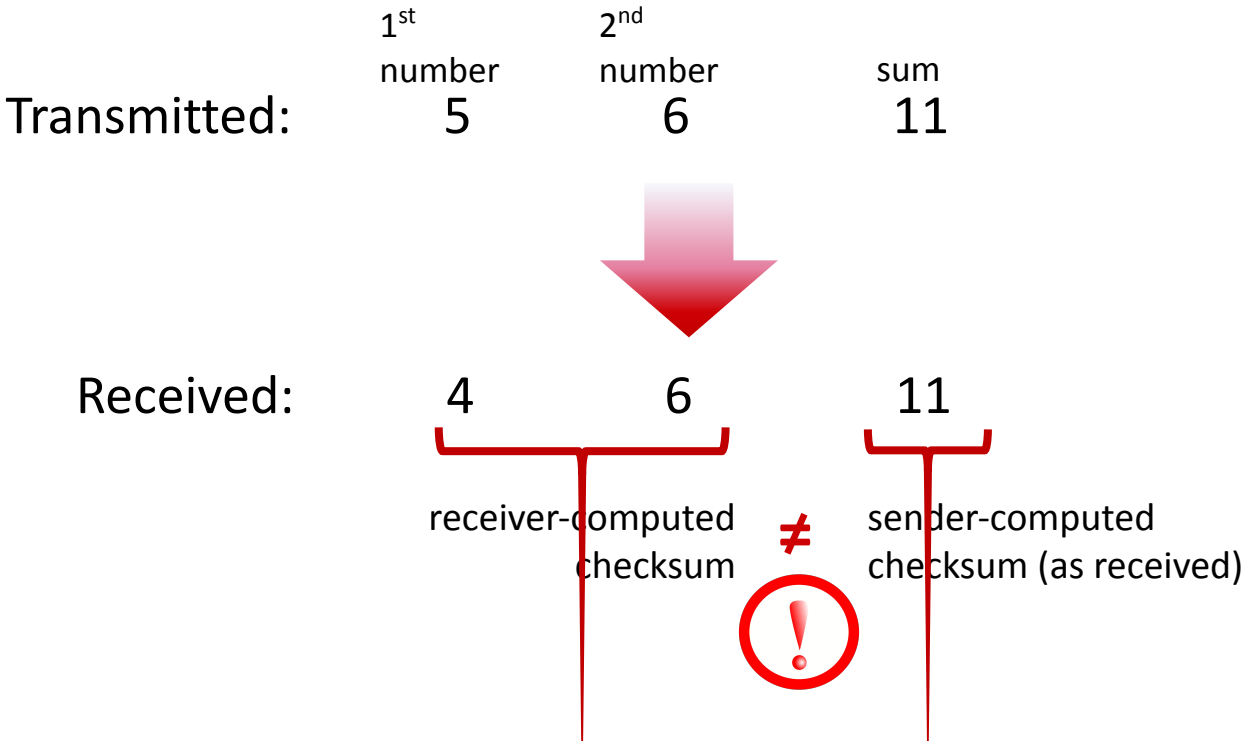
Multiplexing and demultiplexing



SOURCE: TCP/IP Protocol Suite

UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment



UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

sender:

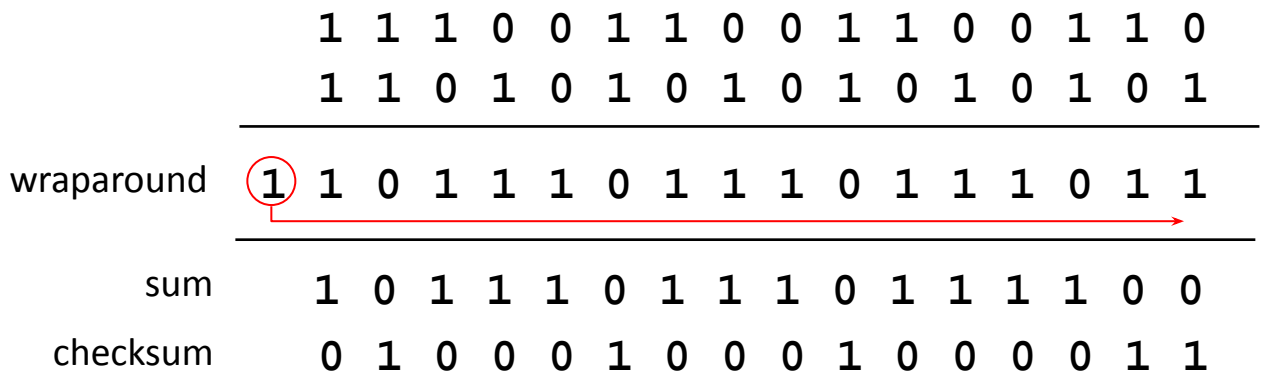
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an example

example: add two 16-bit integers

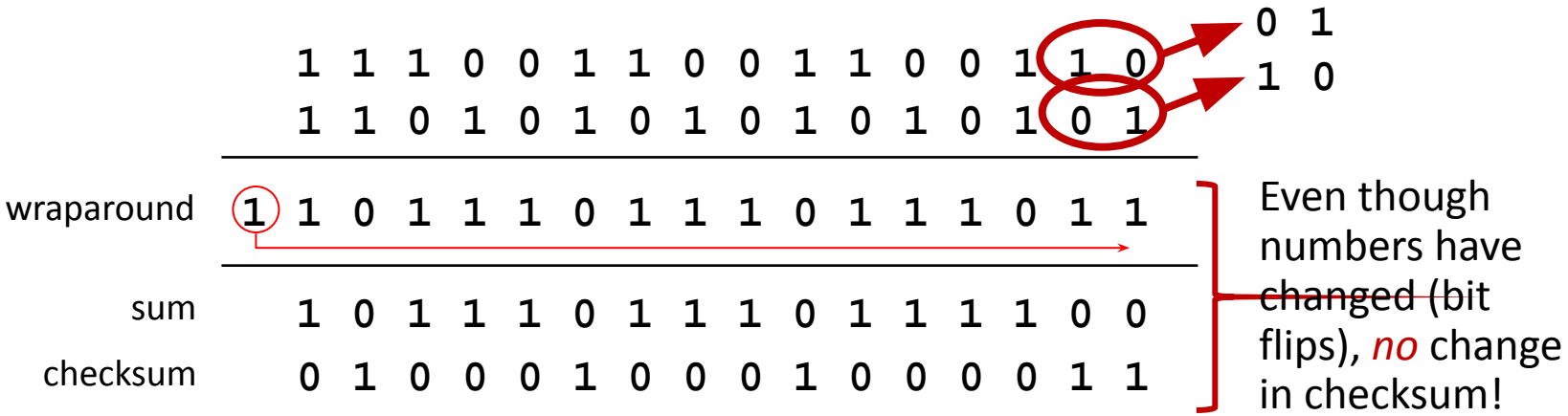


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

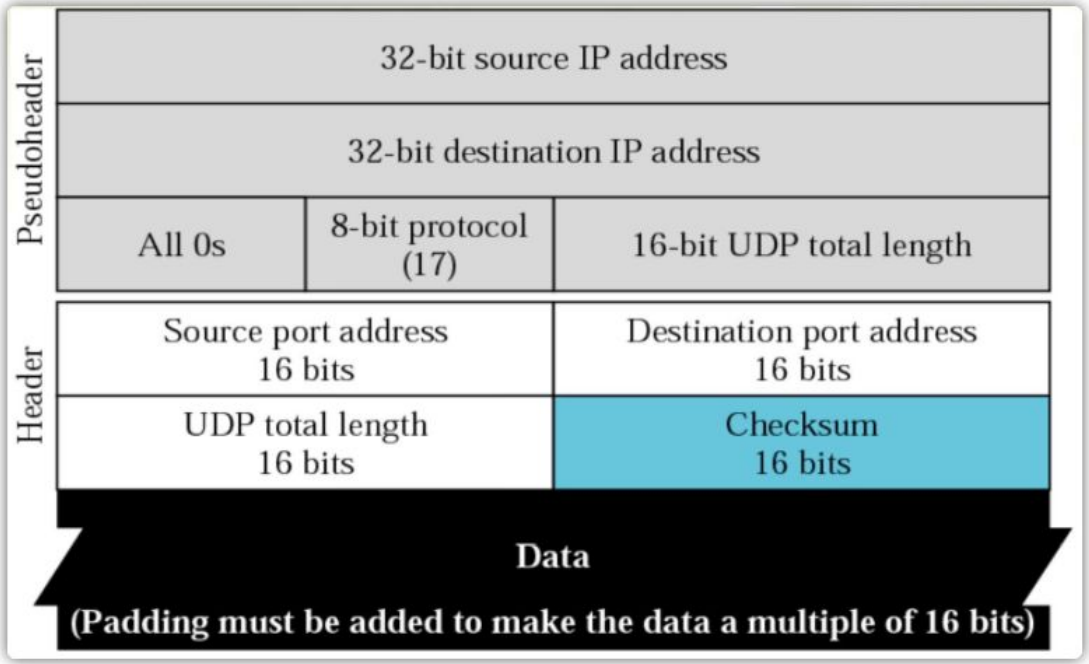
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

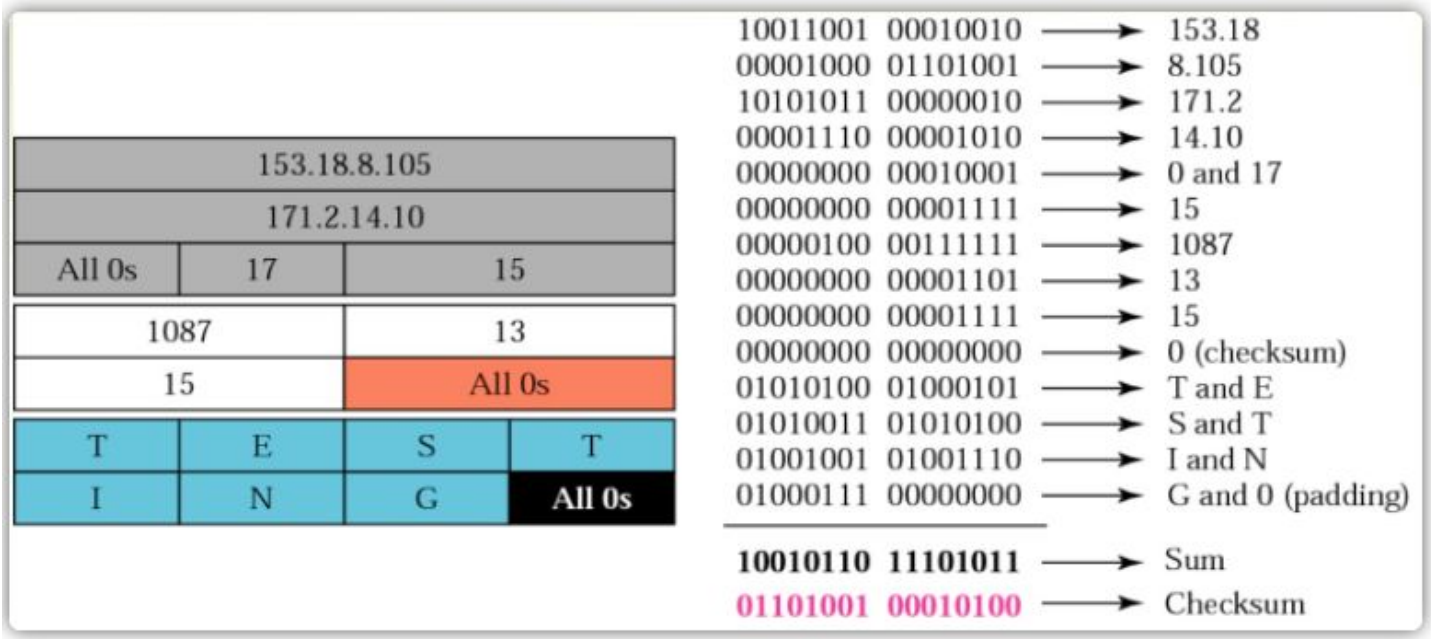
example: add two 16-bit integers



UDP Checksum Calculations



Cont..



At receiver, add everything including checksum and complement if solution is zero then packet is correctly received.

Source: TCP/IP Protocol Suite, Forouzan

Point to Note

- UDP is an example of the connectionless simple protocol we discussed in as a part of Transport layer services with the exception of an optional checksum added to packets for error detection.

Summary: UDP

- Simple protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum) **Optional**
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Optional slides

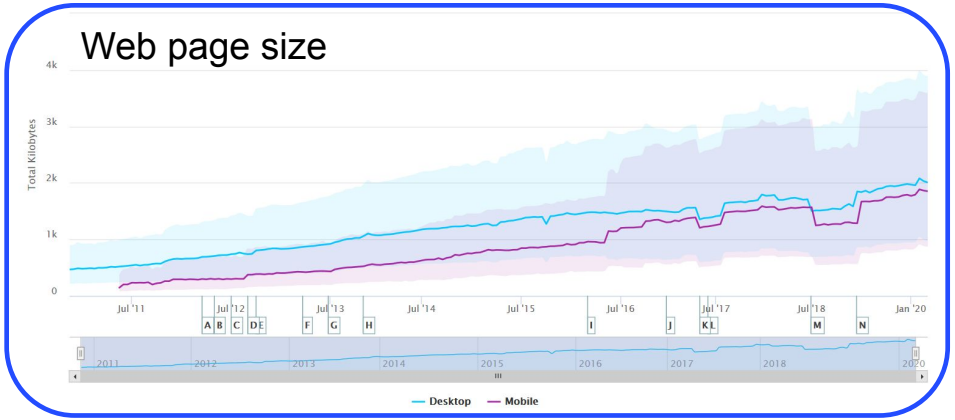
A New Transport Protocol

QUIC: Quick UDP Internet Connections

HTTP/3: HTTP over QUIC is next Generation

Introduction: Change

- **Increasing scale of ..... everything**
  - Flow size changes
  - Flow count increases (e.g., web pages)
  - Flow diversity increase (e.g., web pages)
    - Multiple connections



HTTP Network Stack

HTTP
TLS 1.2+
TCP
IP

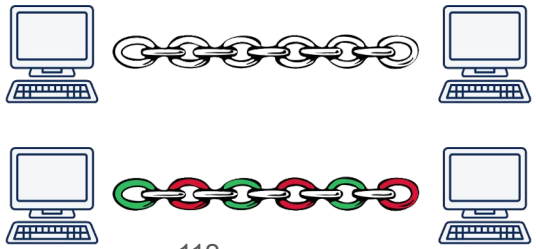
HTTP / 1.1

- January 1997
- **Many parallel TCP connection (6 connections per host name)**
- **HTTP head of line blocking**

HTTP / 2

- May 2015
- **Using Single connection per host**
- **Many parallel streams**
- **TCP head of line blocking**

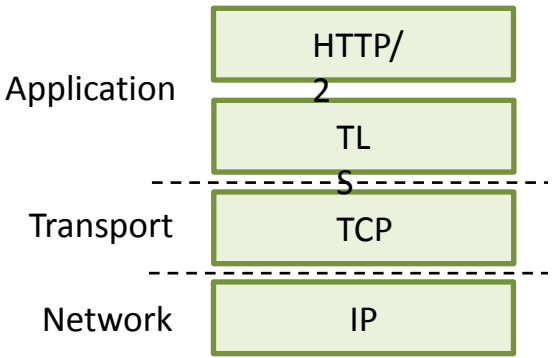
TLS – Transport Layer Security
TCP – Transport Control Protocol
IP – Internet Protocol



https://http3-explained.haxx.se/en/why-quic

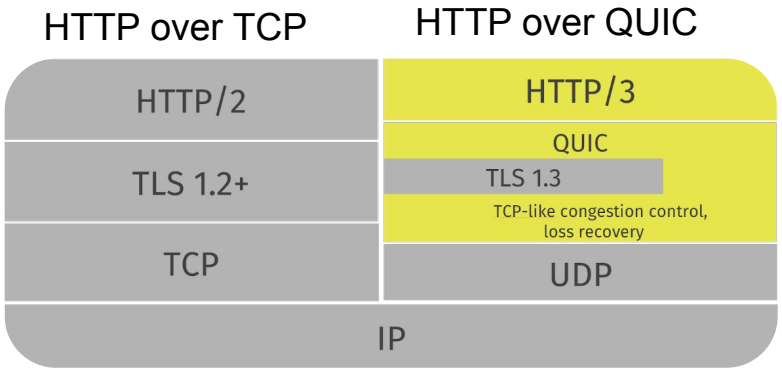
QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



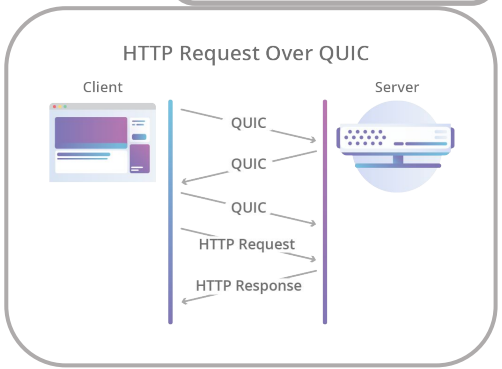
HTTP/2 over TCP

HTTP Over QUIC Network Stack



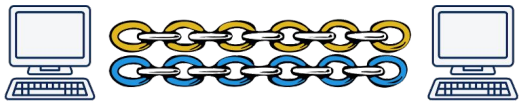
HTTP / 3

Workgroup: QUIC
Internet-Draft: draft-ietf-quic-http-latest
Published: 9 June 2020
Intended Status: Standards Track
Expires: 11 December 2020
Author: M. Bishop, Ed.
Akamai



- **No - TCP head of line blocking**
  - **streams are independent to each other**
- **Faster handshake**
  - **Earlier data**
- **More encryption, always**
- **Over UDP (*Connection less, No resend, No flow control*)**

- Steam impact not connection impact



https://http3-explained.haxx.se/en/why-quic

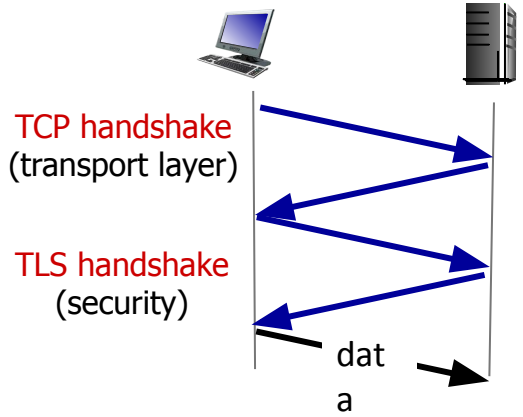
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

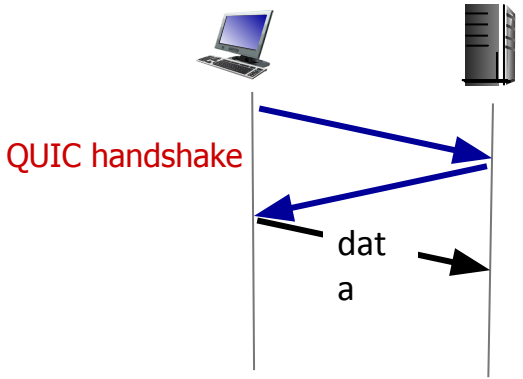
- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment



TCP (reliability, congestion control state) + TLS (authentication, crypto state)

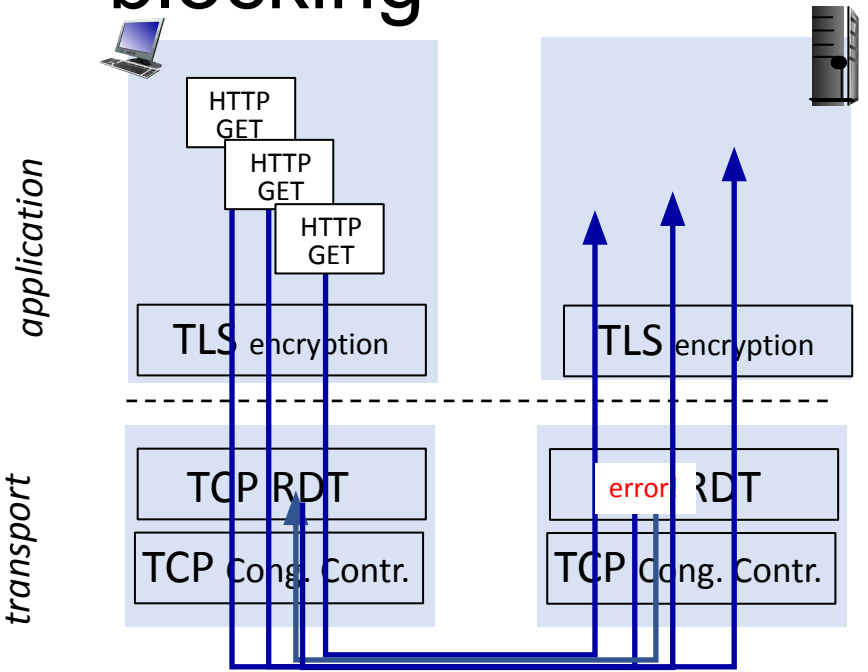
- 2 serial handshakes



QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

QUIC Status

IETF:
specifications in-progress, RFCs likely in 2021

RFC 9000

Implementations:
Apple, Facebook, Fastly, Firefox, F5, Google, Microsoft ...

Server deployments have been going on for a while
Akamai, Cloudflare, Facebook, Fastly, Google ...

Clients are at different stages of deployment
Chrome, Firefox, Edge, Safari
iOS, MacOS

Source: SIGCOMM 2020, QUIC Tutorial https://conferences.sigcomm.org/sigcomm/2020/tutorial-quic.html