Chapter 3: Transport Layer

- CSI 2470, Fall 2025
- Dr. Jie Hu

* Modified from the class notes by Kurose/Ross and by my Ph.D. advisor Dr. Do Young Eun

1

Chapter 3: Transport Layer

Our goals:

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

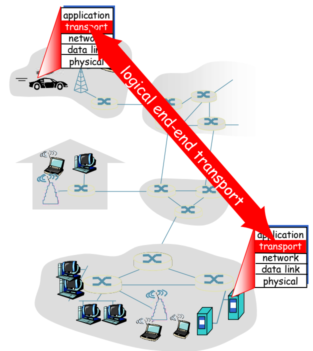
2

Contents

3

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



4

Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

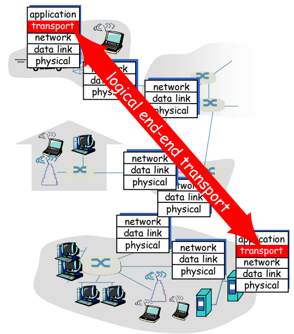
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill (could be Susan and Harvey)
- 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



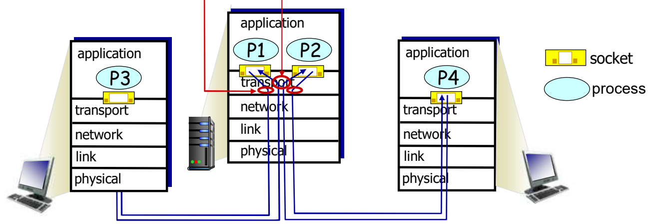
Principles of Transport Services

- Multiplexing/Demultiplexing
- Reliable data transfer
- Flow control
- Congestion control

Multiplexing/demultiplexing

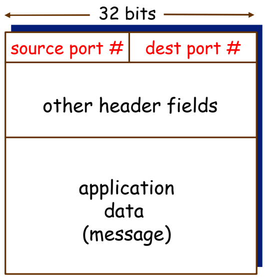
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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 1 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

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

9

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 #

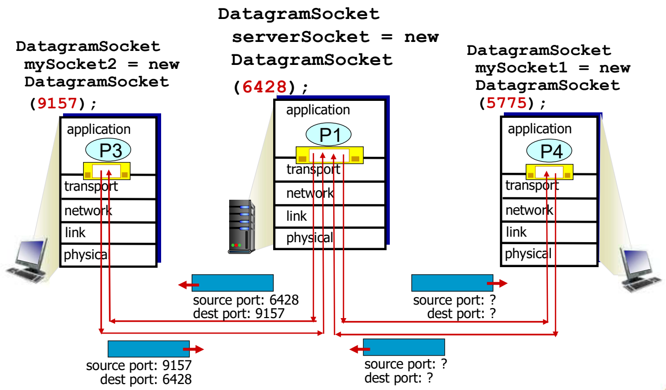
- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

10

Connectionless demux: example



Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

11

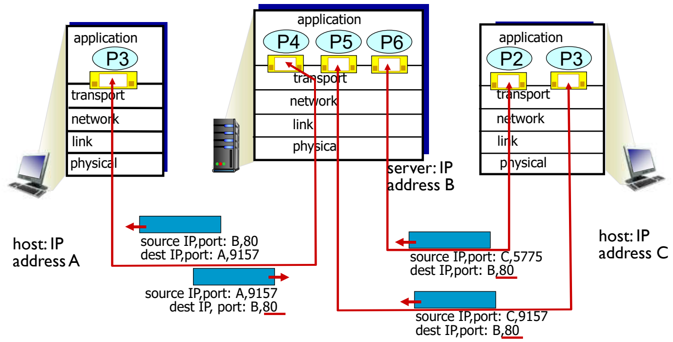
Connection-oriented mux/demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Receiving host 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

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

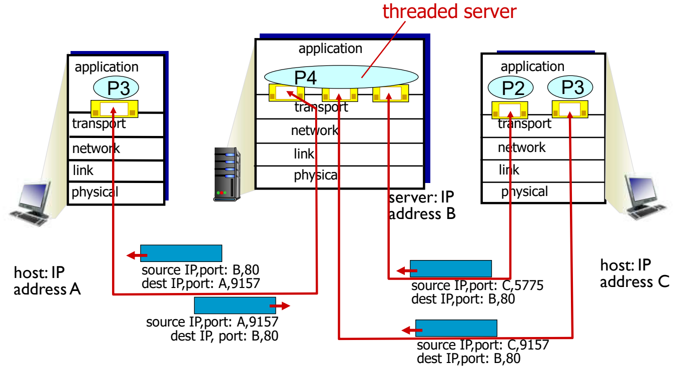
12

Connection-oriented demux: example

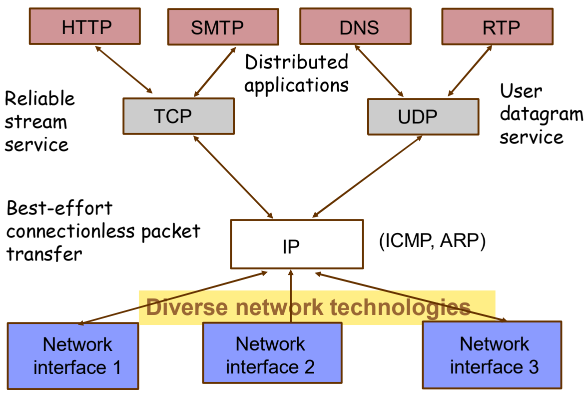


three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



TCP/IP Protocol Suite

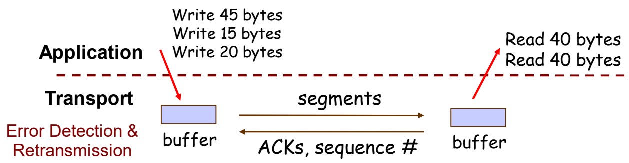


Principles of Transport Services

- Multiplexing/Demultiplexing
- Reliable data transfer
- Flow control
- Congestion control

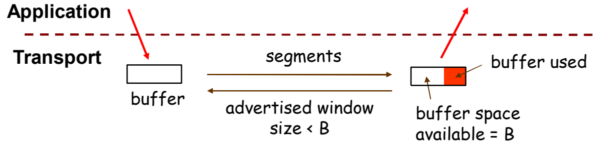
Reliable Byte-Stream Service

- Stream Data Transfer
  - transfers a contiguous stream of bytes across the network, with no indication of boundaries
  - groups bytes into segments
  - transmits segments as convenient
- Reliability
  - error control mechanism to deal with IP transfer impairments



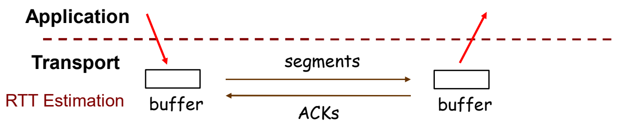
Flow Control

- Buffer limitations & speed mismatch can result in loss of data that arrives at destination
- Receiver controls rate at which sender transmits to prevent buffer overflow



Congestion Control

- Available bandwidth to destination varies with activity of other users
- Sender dynamically adjusts its transmission rate according to network congestion as indicated by RTT (round trip time) & ACKs
- "Elastic" utilization of network bandwidth



Contents

Connectionless Transport:UDP

- Best effort datagram service
- Simple transmitter & receiver
  - Connectionless: no handshaking & no connection state
  - Low header overhead
  - No flow control, no error control, no congestion control
  - UDP datagrams can be "lost" or "out-of-order"
- Multiplexing enables sharing of IP datagram service
- Applications
  - multimedia (e.g. RTP)
  - network services (e.g. DNS, RIP, SNMP)

Why is there a UDP?

Reliable transfer over UDP possible?

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

21

UDP: User Datagram Protocol [RFC 768]

← 32 bits →

source port #	dest port #
UDP length	UDP checksum
Application data (message)	

UDP segment format

port # assignment

0-255: Well-known ports
256-1023: Less well-known ports
1024-65536: Ephemeral client ports

- Source and destination port numbers
  - Client ports are ephemeral
  - Server ports are well-known
  - Max number is 65,535
- UDP length
  - Total number of bytes in datagram (including header)
  - 8 bytes ≤ length ≤ 65,535 bytes
- UDP Checksum
  - Optionally detects errors in UDP datagram

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

22

UDP checksum

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

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.
  - *But maybe errors nonetheless?* More later ....

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

23

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
```

Link layer does error-detection/correction.
Why UDP (or TCP) provides a checksum?

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

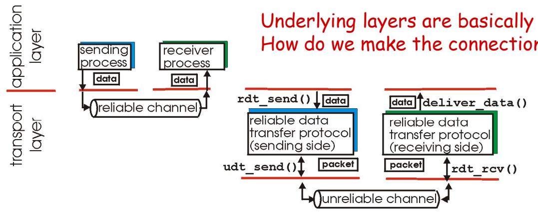
24

Contents

Principles of Reliable Data Transfer (rdt)

- Important in application, transport, link layers
- Top-10 list of important networking topics!

Underlying layers are basically unreliable.
How do we make the connection reliable?



(a) provided service (b) service implementation

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

rdt over Channels with Loss and Errors

- 1st question: what happens if the underlying channels cause errors?
  - Underlying channel may flip bits in packet
  - checksum to detect bit errors
- 2nd question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK

rdt over Channels with Loss and Errors

- 3rd question: what happens if a packet is lost?
  - sender waits "reasonable" amount of time for ACK
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #'s handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer

rdt over Channels with Loss and Errors

- 4th question: what happens if ACK/NAK is corrupted?
  - sender doesn't know what happened at receiver!
  - can't just retransmit: possible duplicate
  - sender adds *"sequence number"* to each packet
  - sender retransmits current packet if ACK/NAK is garbled
  - receiver discards (doesn't deliver up) duplicate packet

Developing rdt: stop-and-wait (SAW)

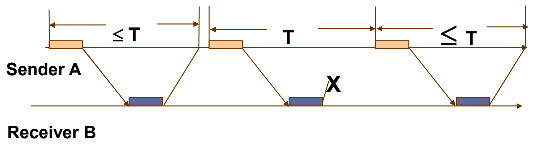
- Simplest scheme
  - Protocols in which the sender sends one packet and then waits for an acknowledgement before proceeding are called SAW.
  - Assume that no automatic buffering and queuing at the receiver, the sender must never transmit a new packet until the old one has been fetched.
- Assume acknowledgement delay bound T
  - Implicit request for retransmission
  - T: time-out period

Q: How to choose this time-out period?

- Only one outstanding packet at anytime

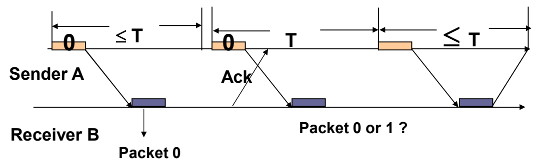
rdt: stop-and-wait (SAW)

- Assume no delay (or all packets suffer the same delay)
  - Sender does not need sequence number.
  - If the packet is error-free, B sends an ACK back to A; if the packet is in error, B sends a NAK back to A.



rdt: stop-and-wait (SAW)

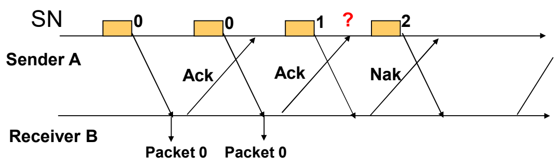
- **The trouble with unnumbered packets**
  - The ACK for packet 0 is abnormally delayed, so A retransmits packet 0, but B cannot tell whether it is 0 or 1.



Solution ? ➔ Use a sequence number in the frame header to identify successive packets.

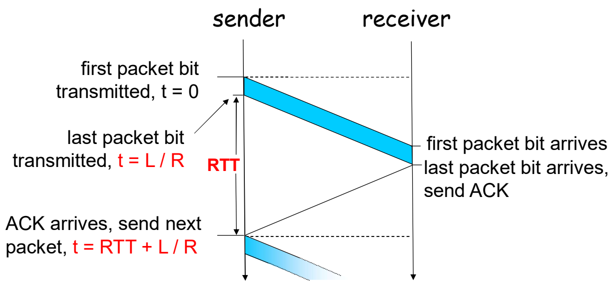
rdt: stop-and-wait (SAW)

- **The trouble with unnumbered ACKs**
  - If the transmitter at A times-out and sends packet 0 twice, node B can use the sequence numbers to recognize that packet 0 is being repeated. It must send an ACK for both copies, however, and (since ACKs can also be lost) the transmitter cannot tell whether the second ACK is for packet 0 or 1.



Solution ? → Instead of returning ACK or NAK on the reverse link, returns the number of the next packet awaited.

rdt: stop-and-wait (SAW) operation (no error)



No error & No excessive delay (no timeout)

Here, RTT is just twice of propagation delay (+ processing delay at the receiver)

rdt: stop-and-wait (SAW): Performance

- SAW works, but performance may stink.
- example: 1 Gbps link, 15 ms end-to-end one-way propagation delay, 1KByte packet:

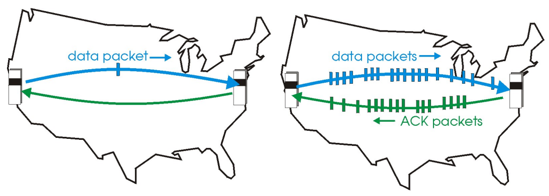
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- U_{sender}: utilization – fraction of time sender busy sending
- 1KB pkt every 30 msec → 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelined protocols

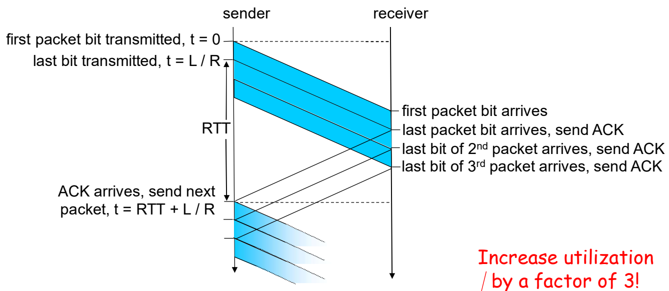
Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *Go-Back-N, Selective Repeat*

Pipelining: increased utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

We can do a lot better!

Sliding Window Protocols

- Major drawback of Stop-and-Wait
  - Only one frame can be in transmission at a time
  - Leads to inefficiency if propagation delay is much longer than the transmission delay
- *First we assume no error (delay) & loss* → sliding window protocol
  - With error and loss → Go-Back-N protocol or Selective-Repeat (will be discussed later)
- Sliding window flow control
  - Allows transmission of multiple packets
  - Assigns each packet a *k*-bit sequence number
  - Range of sequence number is $[0,1,\ldots,2^k-1]$, i.e., packets are counted modulo 2^k

Sliding Window Protocols

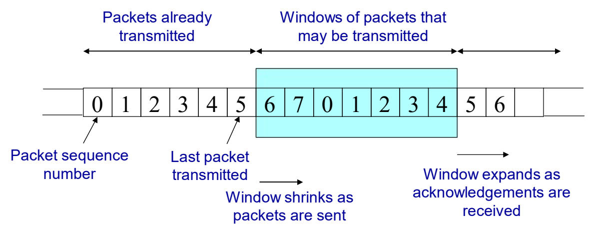
- Bidirectional protocols
  - To use the same connection for data in both directions, instead of one for data (forward) and one for ack (reverse). The receiver can tell whether the packet is data or ack by looking at the "type" in the packet header.
- Piggybacking (optional)
  - Temporarily delaying outgoing acks so that they can be hooked onto the next outgoing data frame (free ride for acks!).

Q: How does receiver ack multiple packets when piggybacking ?

- How long should the receiver wait for a packet to piggyback the ack?
  - Too long – will incur retransmission
  - Too short – separate acknowledgement

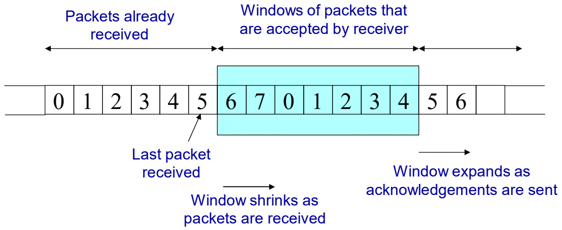
Operation of Sliding Window

- **Sending Window:** no error & loss; only delay here
  - The sequence numbers within the sender's window represent packets that are allowed to send, but as yet not acknowledged.
  - The sender must have a buffer which keeps the copy of all packets within the window because these packets may need retransmission.



Operation of Sliding Window

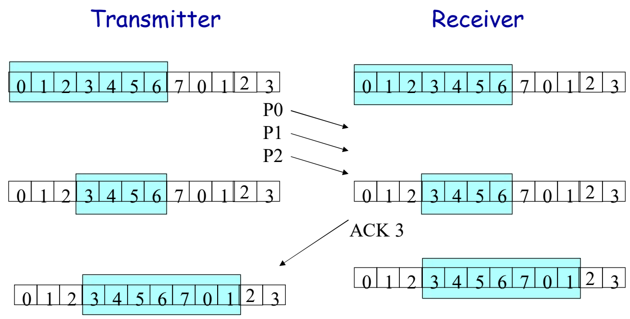
- **Receiving Window:** no error & loss; only delay here
  - The receiver maintains a *receiving window* corresponding to the sequence numbers of packets that the receiver is permitted to accept



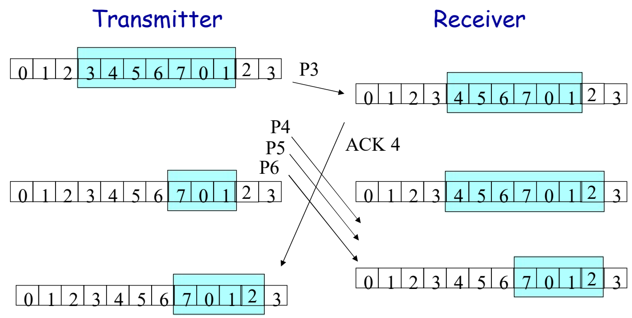
Operation of Sliding Window

- **Is " flow control " achieved?**
  - Receiver can control the size of the sending window
  - By limiting the size of the sending window, data flow from the sender to the receiver can be limited.
- Interpretation of *ACK-N* message:
  - Receiver acknowledges all packets until (but not including) sequence number *N*.

Example



Example (cont'd)



Go-Back-N (GBN): Overview

- If no error & loss → GBN = sliding window protocol
- Window size W = N
- N (=W): determines how many successive packets can be sent without a request for a new packet
  - Node A is not allowed to send packet ***i+N*** before ***i*** has been acked (i.e., before ***i+1*** has been requested)
- Timer set for each in-flight packet:
  - Timer value (timeout period) **T ≥ Wτ**
  - τ = transmission delay of a packet

GBN: Overview

- The sender transmits packets 0,1,2, …,W-1 and waits for up to T seconds for each of their ACKs. As soon as the receiver gets an ACK for packet 0, it transmits packet W.
- If time-out, "go back" and retransmit all subsequent packets
- out-of-order packet arrival at receiver
  - discard (don't buffer) → no receiver buffering!
  - Re-ACK packet with highest in-order seq. number (k), i.e., generate ACK (k+1)
    - Duplicate ACK (NAK-free)
    - Use NAK (which includes ACK for the last segment)

Two different versions to "complain"

Go-Back-N (GBN)

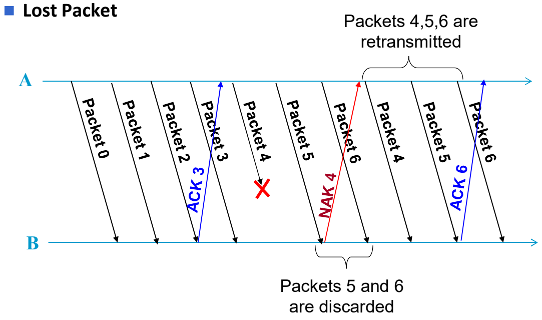
- Operations:
  - A station may send multiple packets as allowed by the window size.
  - receiver sends a ***NAK i*** if packet ***i*** is in error. After that, the receiver discards all incoming packets until the packet in error was correctly retransmitted.
  - If sender receives a ***NAK i***, it will retransmit packet ***i*** and all packets ***i+1, i+2***, … which have been sent, but not been acknowledged.

In reality…

- Real Implementation of GBN?
  - Different textbooks have slightly different versions for GBN
  - With NAK or NAK-free
  - Real TCP uses combinations
    - All TCPs use sliding window protocols
    - TCP-Reno/newReno: duplicate ACKs (NAK-free protocol)
    - TCP-SACK (Selective Acknowledgement Options): more like Selective-Repeat (will see shortly)
- Here, we focus on concept, not the actual implementations

Effect of transmission error

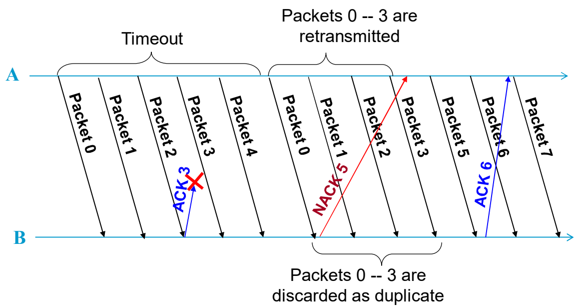
- **Lost Packet**



NAK 4 includes "acking everything up to Packet 3 (inclusive), and report problem in Packet 4"

Effect of transmission error

- **Lost ACK**



Conceptual diagram: Real protocol may handle out-of-sequence segments differently
(here, B decides to complain via NACK5. It may decide not to)

Go-Back-N: A-to-B Transmission

- After each transmission, A sets an acknowledgement timer (time-out timer) for the packet just transmitted.
- Case1: Damaged packet
  - A transmits packet ***i***. B detects an error and has previously successfully received packet ***(i-1).*** B must send a ***NAK i*** right away, indicating that the packet is rejected. When A receives this ***NAK***, it must retransmit packet ***i*** and all subsequent packets that it has transmitted.
  - Packet ***i*** is lost in transit. A subsequently sends packet ***(i+1).*** B receives a damaged packet ***(i+1***), and sends a ***NAK i.*** (must!!)
  - Packet ***i*** is lost in transit and A does not soon send additional packets. B receives nothing and returns neither an ***ACK*** or a ***NAK***. A will eventually time out and retransmit packet ***i***.

"damaged" = fail to pass CRC test, or out-of-order (future) arrival

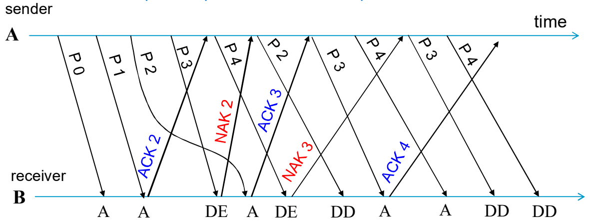
Go-Back-N: A-to-B Transmission

- Case 2: Lost ***ACK***.
  - B receives packet ***i*** and sends ***ACK(i+1),*** which is lost in transit. Since acknowledgments are cumulative, it may be that A will receive a subsequent ***ACK*** to a subsequent packet that will do the job of the lost ***ACK*** before the associated timer expires.
  - A's timer expires. A will then retransmit packet *i* and all subsequent packets.
- Case 3: Lost ***NAK***. If a ***NAK*** is lost, A will time out on the associated packet and retransmit that packet & all subsequent packets.
- **Note**: Each individual packet may not be individually acknowledged, since an arriving ***ACK(i+1)*** implies that all packets from packet *i* and backwards to the last acknowledged packets have been delivered sound and well.

Class Exercise

- Go Back N with large enough window size
- Fill in packet #, ACK or NAK #

Here packets may arrive out of order (possible in TCP layer)
Assume: duplicate packets are "silently" discarded



A: Accepted as "in order" packet
DE: Discard as Error DD: Discard as Duplicate

Go-Back-N: Sequence Numbers

- A sequence space can only support a window size of $\boldsymbol{2^k - 1}$ for ***k***-bits acknowledge field. (not $\boldsymbol{2^k = W}$)
- Why?
  - Consider a case in which a station transmits packet 0 and gets an ***ACK1***, and then transmits packet 1,2,3,4,5,6,7,0 (sequence space of k = 3) and gets another ***ACK1***.
  - *This is ambiguous* → Is this (1) all eight packets were received correctly, or (2) all packets were lost (discarded after found in error) in transit, and the receiving station is repeating its previous ***ACK1***

duplicated ACKs for "NAK-free version of protocol"

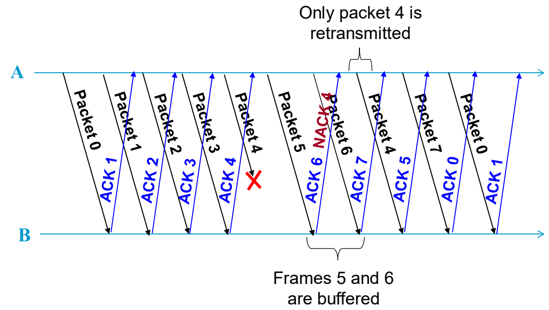
- Maximum # of outstanding packets (window size) should be up to 2^k-1, when the max. seq. # is 2^k

Selective-Repeat (SR)

- Similar to Go-Back-N. However, the sender only retransmits packets for which an ACK not received (or ***NAK*** is received).
- Advantage over Go-Back-N:
  - Fewer Retransmission
  - Don't have to retransmit all packets for a single error
- Disadvantages:
  - More complexity at sender and receiver
  - Each packet must be acknowledged individually (**no** cumulative acknowledgements)
  - Receiver may receive packets out of sequence (buffer required, as sender only retransmits packet in error, not all subsequent ones)

Selective-Repeat (SR)

- **Lost Packet**

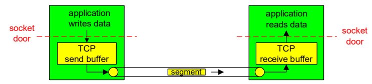


Contents

57

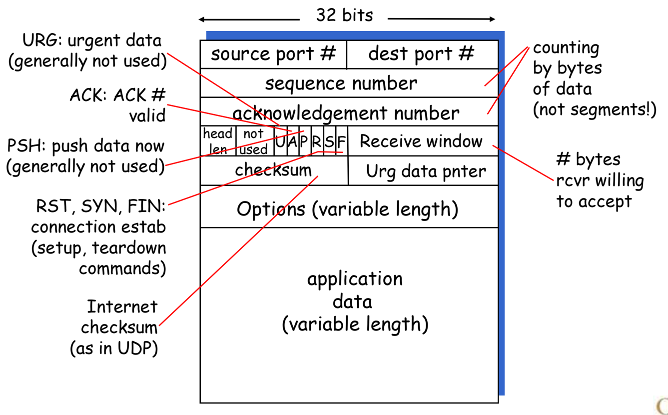
TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver



58

TCP segment structure: Overview



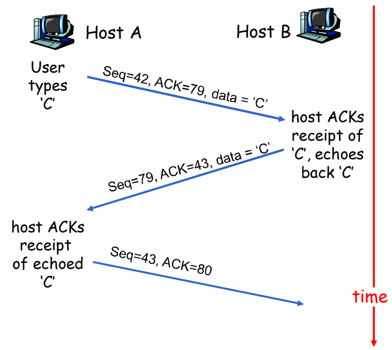
59

TCP sequence numbers and ACKs

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- 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 implementer (e.g., depends on GBN or SR)



simple telnet scenario

60

TCP Round Trip Time and Timeout

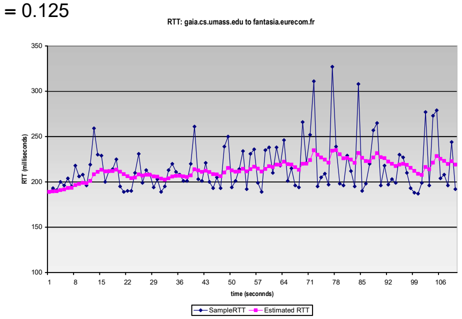
Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

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 and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP Round Trip Time and Timeout

Setting the timeout
- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT ->` larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
(typically, β = 0.25)
```

Then set timeout interval:

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


TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

rdt in TCP: sender events

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

65

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
          }
} /* end of loop forever */
```

SendBase = the sequence # of the oldest unacked bytes

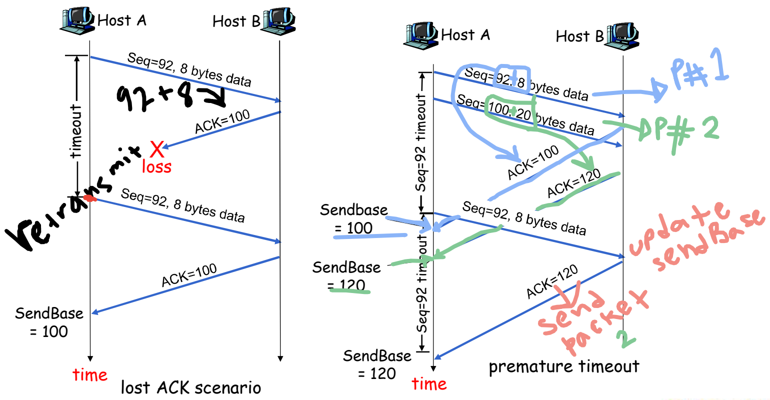
Comment:
• SendBase-1: last cumulatively ack'ed byte

Example:
• SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

66

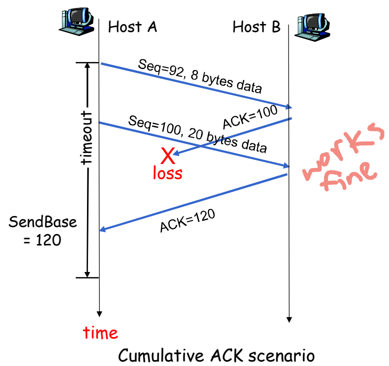
TCP: retransmission scenarios



Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

67

TCP retransmission scenarios (more)



Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

68

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte NAK-disabled here!!
Arrival of segment that partially or completely fills gap	Immediately send ACK, provided that segment starts at lower end of gap

more of selective-repeat nature

69

Fast Retransmit (for ACK-only TCP)

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

complaint

Note: there's no NAK here. If NAK-enabled, sender would immediately retransmit upon receipt of NAK, but here ACK only. So, question is: "when should the sender retransmit?"

70

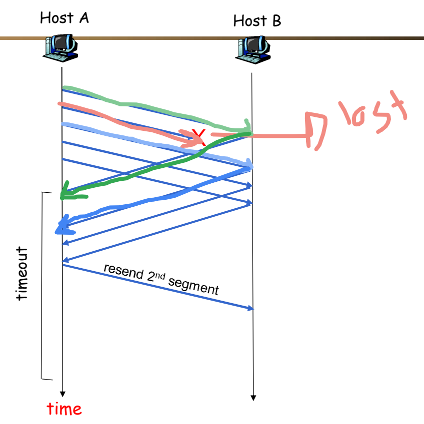


Figure: Resending a segment after triple duplicate ACK

71

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
          }
          else {    /* y =< SendBase */
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number  y
               }
```

a duplicate ACK for already ACKed segment

fast retransmit

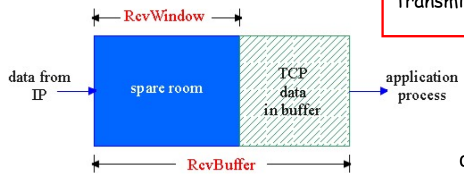
No new timer here!!

72

Contents

TCP Flow Control

- receiver side of TCP connection has a receiver buffer:

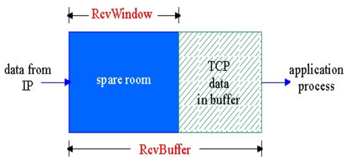


- app process may be slow at reading from buffer

flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

d-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

receiver buffer doesn't overflow:

LastByteRcvd – LastByteRead ≤ RcvBuffer

(Suppose TCP receiver discards out-of-order segments, i.e., Go-Back-N style protocol)

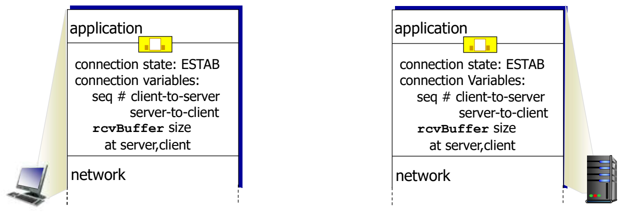
- spare room in buffer

= RcvWindow = RcvBuffer-[LastByteRcvd - LastByteRead]

TCP Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters




```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

```
Socket connectionSocket =
  welcomeSocket.accept();
```

Three-Way Handshake: Open Connection

- Step 1: client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data
- Step 2: server host receives SYN, replies with SYN-ACK segment
  - server allocates buffers
  - specifies server initial seq. #
- Step 3: client receives SYN-ACK, replies with ACK segment, which may contain data.
  - If "contain actual data" here, it takes one RTT to open a TCP connection

Three-Way Handshake: Open Connection

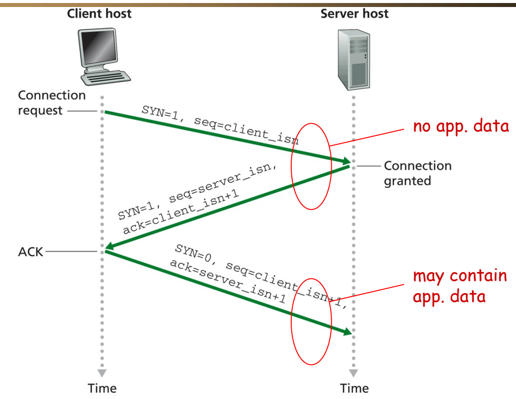


Figure 3.38 ♦ TCP three-way handshake: segment exchange

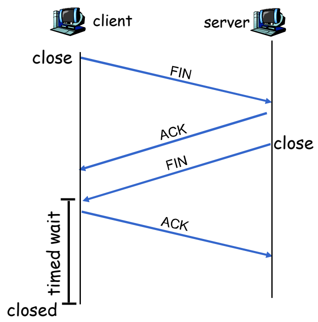
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



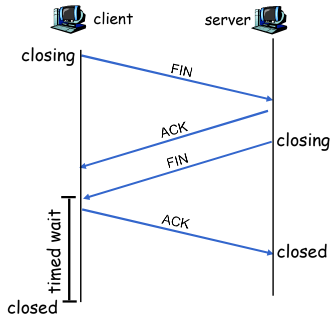
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



Contents

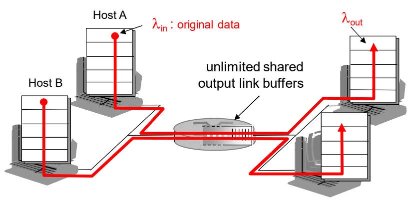
Principles of Congestion Control

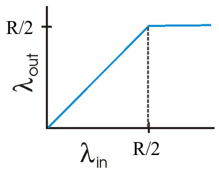
Congestion:

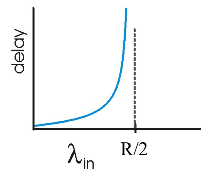
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/Costs of Congestion: Scenario 1

- two senders, two receivers
- one router with capacity R, *infinite* buffers
- no retransmission



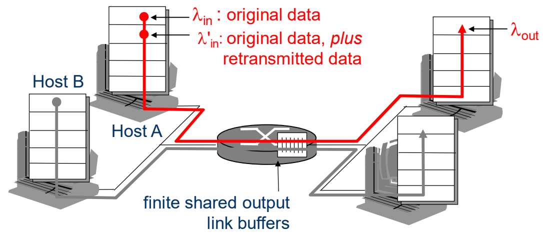




- large delays when congested
- maximum achievable throughput

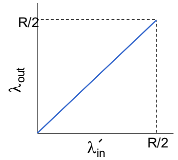
Causes/costs of congestion: Scenario 2

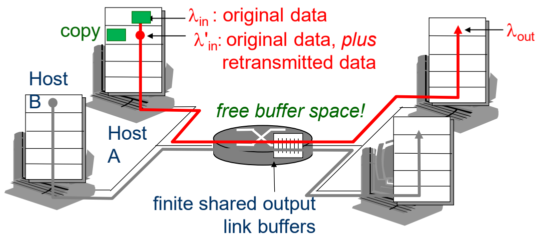
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



Congestion Scenario 2a: ideal case

- sender sends only when router buffers available

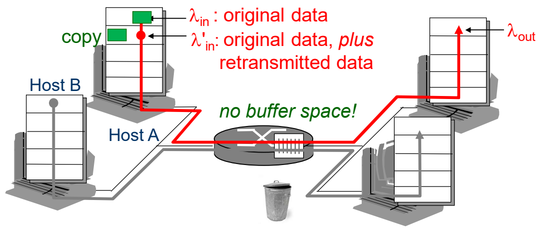




85

Congestion Scenario 2b: *known* loss

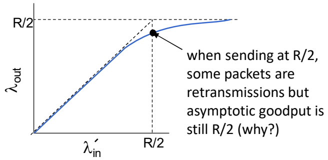
- packets may get dropped at router due to full buffers
  - sometimes lost
- sender only resends if packet *known* to be lost (admittedly idealized)

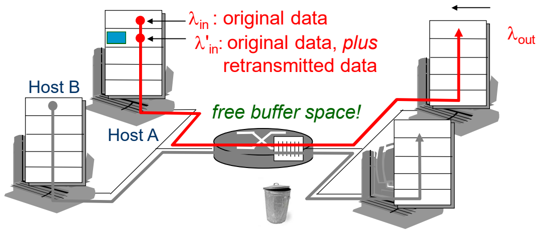


86

Congestion Scenario 2b: *known* loss

- packets may get dropped at router due to full buffers
  - sometimes not lost
- sender only resends if packet *known* to be lost (admittedly idealized)

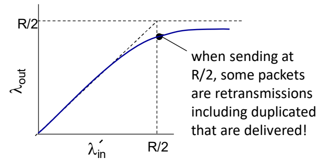


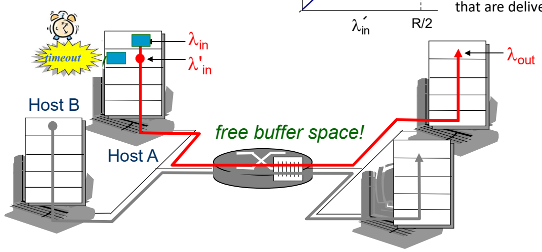


87

Congestion Scenario 2c: *duplicates*

- packets may get dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

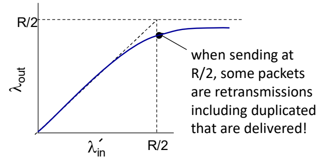




88

Congestion Scenario 2c: *duplicates*

- packets may get dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

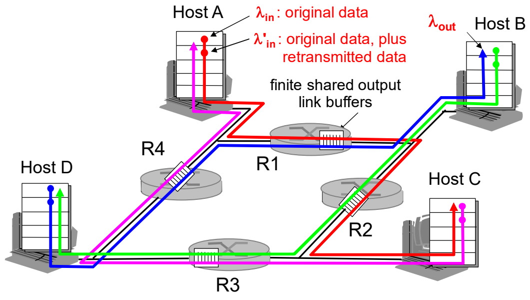
- more work (retransmission) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

89

Causes/Costs of Congestion: Scenario 3

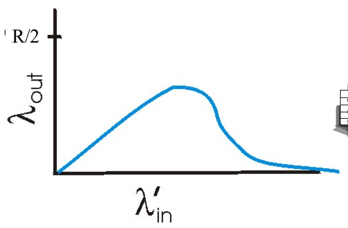
- four senders
- multihop paths
- timeout/retransmit
- capacities of routers = R

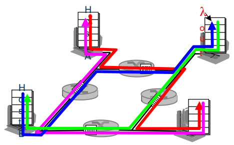
Q: what happens as λ_{in} and λ'_{in} increase ?



90

Causes/Costs of Congestion: Scenario 3





Another "cost" of congestion:

- when packet dropped, any upstream transmission capacity used for that (soon-to-be-lost) packet was "wasted"!

91

Approaches towards congestion control

Two broad approaches towards congestion control:

End-to-End congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate at which sender should send

92

TCP Congestion Control

- Congestion is everywhere and inevitable
- Fixing the problem at one router does not remove the congestion in the network; it simply moves the problem to another place!
- Approaches:
  - Senders reduce/adjust transmission rates
  - But, senders want higher rates, while not causing congestion
  - How does each sender figure out the maximum value of transmission rate, without causing congestion?
  - *You must find out the "rate" on your own!*
  - Fairness is here!
- TCP congestion control does all of these!
  - It's end-to-end congestion control: no network assistance

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

93

Flow Control/Congestion Control

- Two mechanisms in TCP that control the flow of information:
  - Flow Control mechanism (rwnd) (= "RcvWindow" in the text)
  - Congestion Control mechanism (cwnd) = ("CongWin" in the text)
- Effective window = min (rwnd, cwnd)
- "Flow Control" mechanism is simple.
  - Receiver uses a *window size* field in the ack to advertise the size of the window *rwnd* that reflects its buffer capacity.
  - An inadequate receiver buffer size may constrain the throughput of the connection regardless of the state of the network.
- Congestion Control mechanism is more complex.
  - **cwnd is dynamic, function of "perceived" network congestion**

Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

94

TCP Congestion Control (**cwnd**)

- Initially – there is a multiplicative increase of the window size (slow start)
- Normal operation: AIMD – Additive Increase and Multiplicative Decrease of the window size (congestion avoidance phase).
- How can end-systems detect congestion?
  - Router silently drops packet when congestion occurs.
- Assumption:
  - When there is a packet loss, congestion occurs "somewhere" in the network along your route

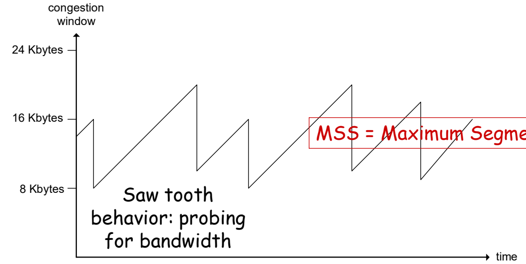
Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

95

TCP Congestion control

Key idea: Additive-Increase, Multiplicative-Decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **Cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **Cwnd** in half after loss



Jie Hu CSI 2470, Fall 2025 OAKLAND UNIVERSITY

96

TCP Congestion Control: details

- sender limits transmission:
  LastByteSent-LastByteAcked ≤ cwnd
- Roughly,

$$\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \ \text{Bytes/sec}$$

- **Cwnd** is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**Cwnd**) after loss event

three mechanisms:

- AIMD
- slow start
- conservative after timeout events

TCP Congestion Control: Algorithm

- Two "phases"
  - slow start
  - congestion avoidance
- Important variables:
  - **Cwnd**
  - **ssthresh:** defines threshold between slow start phase and congestion control phase
- "probing" for usable bandwidth:
  - Ideally: transmit as fast as possible (**Cwnd** as large as possible) without loss
  - *increase* **Cwnd** until loss (congestion)
  - loss: *decrease* **Cwnd**, then begin probing (increasing) again

Q: What is the "target" window size?

Slow Start

- When connection begins, **Cwnd** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- Increment window size by 1 MSS on each new ack
- Slow start phase ends when window size reaches (or exceeds) the slow-start threshold (= ssthresh)
- **cwnd** grows **exponentially** with time during slow start
  - factor of 1.5 per RTT if every other segment ack'd
  - factor of 2 per RTT if every segment ack'd
  - Could be less if sender does not always have data to send

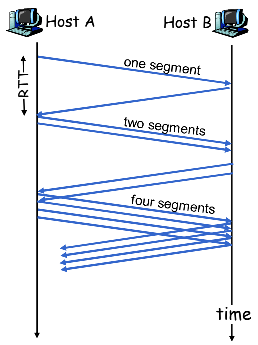
TCP Slow Start

Slowstart algorithm

```
initialize: Cwnd = 1
for (each segment ACKed)
    Cwnd++
until (loss event OR
     Cwnd > ssthresh)
```

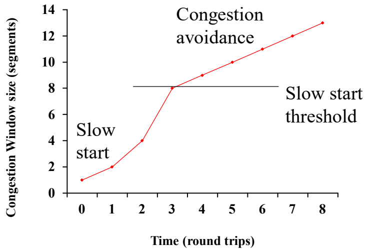
Cwnd unit = "MSS" here

- Exponential increase (per RTT) in window size (not so slow!)
- Loss event: timeout and/or three duplicate ACKs



Congestion Avoidance

- On each new ack, increase cwnd by 1/cwnd segment
  - In terms of bytes, cwnd ← cwnd + MSS * MSS/cwnd
- **cwnd** increases **linearly** with time during congestion avoidance



Example assumes that acks are not delayed

Jie Hu   CSI 2470, Fall 2025   OAKLAND UNIVERSITY

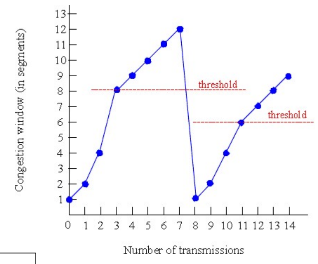
101

TCP Congestion Avoidance

Congestion avoidance

/* slowstart is over */
/* Cwnd > ssthresh */
Until (loss event) {
every cwnd segments ACKed:
Cwnd++
}
ssthresh = Cwnd/2
Cwnd = 1
perform slowstart

In case of TCP Tahoe



Cwnd unit = "MSS" here

Congestion Avoidance phase ends when there is a loss event.

Jie Hu   CSI 2470, Fall 2025   OAKLAND UNIVERSITY

102

Refinement: Inferring loss (TCP-Reno)

- After 3 dup ACKs:
  - **Cwnd** is cut in half
  - window then grows linearly
- But after timeout event:
  - **Cwnd** instead set to 1 MSS;
  - Enter slow-start

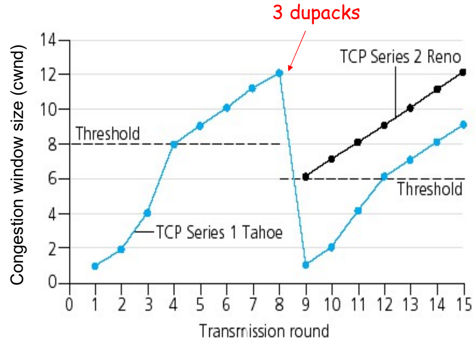
Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

Jie Hu   CSI 2470, Fall 2025   OAKLAND UNIVERSITY

103

Refinement (more)



Jie Hu   CSI 2470, Fall 2025   OAKLAND UNIVERSITY

104

Summary: TCP Congestion Control (Reno)

- When **Cwnd** is below **ssthresh**, sender in slow-start phase, window grows exponentially.
- When **Cwnd** is above **ssthresh**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **ssthresh** set to **Cwnd/2** and **Cwnd** set to **Threshold**.
- When timeout occurs, **ssthresh** set to **Cwnd/2** and **Cwnd** is set to 1 MSS, then enter slow-start.

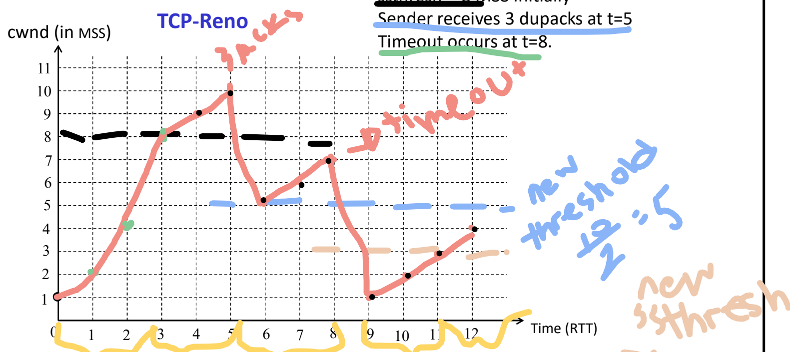
TCP sender congestion control (Reno)

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	Cwnd = Cwnd + MSS, If (Cwnd > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of Cwnd every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	Cwnd = Cwnd+MSS x (MSS/Cwnd)	Additive increase, resulting in increase of Cwnd by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = Cwnd/2, Cwnd = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. Cwnd will not drop below 1 MSS.
SS or CA	Timeout	Threshold = Cwnd/2, Cwnd = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	Cwnd and Threshold not changed

Cwnd unit = "bytes" here

Exercise

~~ssthresh = 8~~ MSS initially
Sender receives 3 dupacks at t=5
Timeout occurs at t=8.



Show how cwnd and ssthresh change over time on this plot.

TCP throughput: TCP-Reno

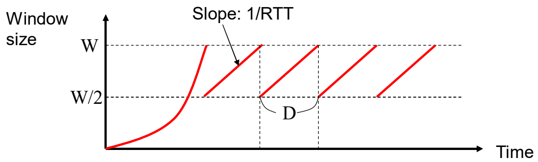
- What's the average "throughout" of TCP as a function of window size and RTT?
  - Ignore slow start
- Let *W* be the window size when loss occurs.
- When window is *W*, throughput is *W/RTT*
- Just after loss, window drops to *W/2*, throughput to *W/2RTT*.
- Average throughout: *0.75 W/RTT*

Detailed Throughput Analysis

- Assumptions:
  - Infinite data, infinite receiver window size (rwnd = ∞)
  - Single TCP connection, single router
  - Steady-state (Congestion-avoidance phase)
  - Constant RTT

109

Detailed Throughput Analysis



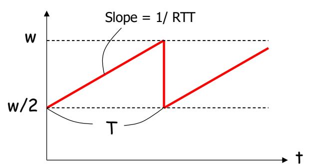
- D = W x RTT/2
- In D sec, the connection sends W/2+(W/2+1)+⋯+2W/2 ≈ $3W^2/8$ units (MSS)
- Throughput **R** = (3MSS x W^2/8) / D
- Loss rate **L** = 1/($3W^2$/8)
- Solving for W →

$$R \approx \frac{1.25MSS}{RTT\sqrt{L}}$$

110

TCP future: TCP over "long, fat pipes"

- Problem with large bandwidth-delay networks



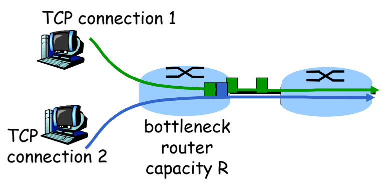
- ✓ **Single TCP connection**
- ✓ Packet size = 1500 bytes
- ✓ RTT = 100ms
- ✓ Capacity (throughput) = 10Gbps
- ✓ **T ≥ 1.5 hrs!**
- ✓ **Packet drop rate ≤ 1 out of 5 billion packets!**

- It takes too long to fill the pipe again after loss → under-utilization !
- Or, to fill the pipe, the packet drop rate must be even smaller than the physical fiber error rate!
- New versions of TCP for high-speed needed!

111

TCP Fairness

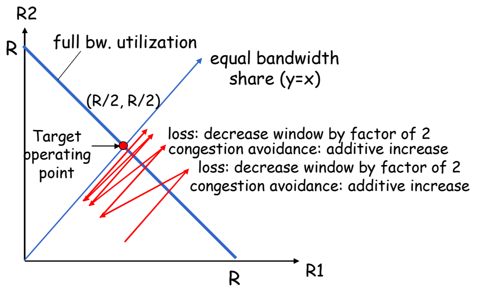
Fairness goal: if *n* TCP sessions share same bottleneck link of bandwidth R, each should have average rate of *R/n*



112

Why is TCP fair?

- 2 sessions share the common link with BW = R
- R1 and R2 are throughputs of each session
- Throughput increases as Cwnd grows and decreases as Cwnd does.



113

Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP-friendly

Fairness and parallel TCP connections

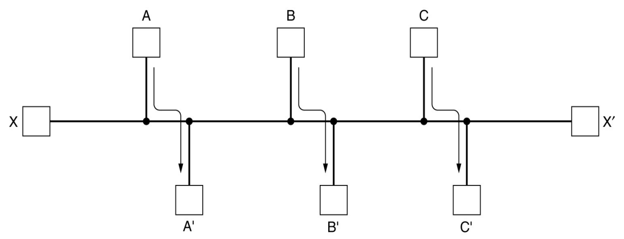
- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

114

Efficiency vs. Fairness

Advanced Issue

Each link provides same bandwidth R



Maximize the total throughput (sum of all throughputs)?

115

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"

116

End of Chapter 3!!

Long chapter, lots of stuff here

117

Backup with Notes Included

118

rdt: stop-and-wait (SAW): Performance

- SAW works, but performance may stink
- example: 1 Gbps link, 15 ms end-to-end one-way propagation delay, 1KByte packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

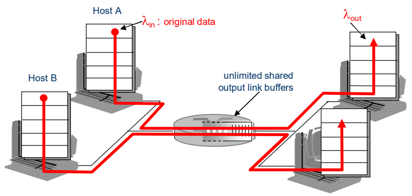
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

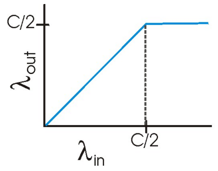
- U_{sender}: utilization – fraction of time sender busy sending
- 1KB pkt every 30 msec → 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

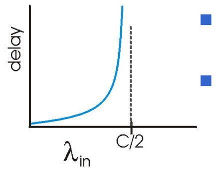
119

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission





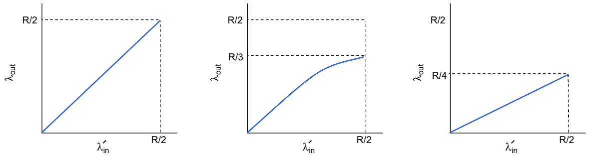


- large delays when congested
- maximum achievable throughput

120

Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}

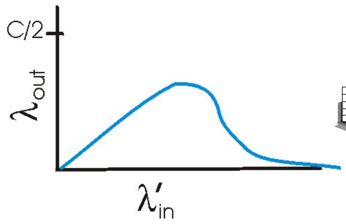


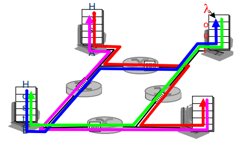
a. b. c.

"costs" of congestion:
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

121

Causes/costs of congestion: scenario 3





Another "cost" of congestion:
- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

122

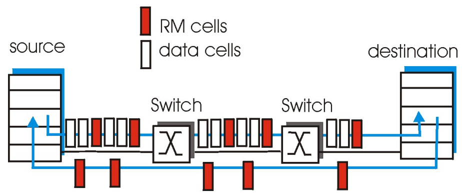
Case study: ATM ABR congestion control

ABR: available bit rate:
- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:
- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

123

Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus maximum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

124

How does TCP (end-host) detect a packet loss?

- Retransmission timeout (RTO)
- Duplicate acknowledgements
  - In practice, triple duplicate acks are considered to be a signal of a packet loss.

Detecting Packet Loss Using Retransmission Timeout (RTO)

- At any time, TCP sender sets retransmission timer for one TCP packet (or segment)
- If acknowledgement for the segment is not received before timer goes off, the segment is assumed to be lost
- RTO dynamically calculated
  - Time out period doubles for each timeout event.

Detecting Packet Loss Using Dupacks: Fast Retransmit Mechanism

- Duplicate acks (dupacks) may be generated when a packet is lost.
- TCP sender assumes that a packet loss has occurred if it receives **three dupacks** consecutively.
- Duplicate acks (dupacks) may also be generated due to out-of-order packet (segment) delivery.

12	8	11	10	9	7

3 dupacks are also generated if a packet is delivered at least 3 places beyond its in-sequence location

Fast retransmit useful only if lower layers deliver packets "almost ordered" ---- otherwise, fast transmit is unnecessary