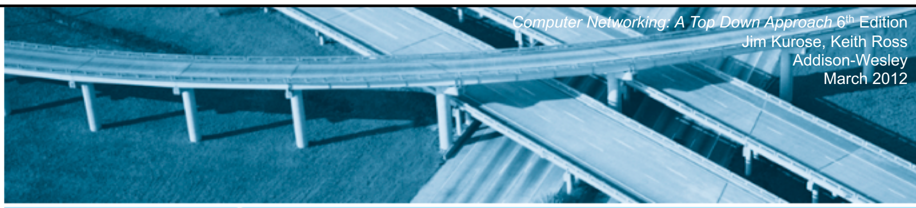


CHAPTER 3

Transport Layer

CHAPTER 3

Transport Layer

our goals:

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

3-2

CHAPTER 3

Transport Layer

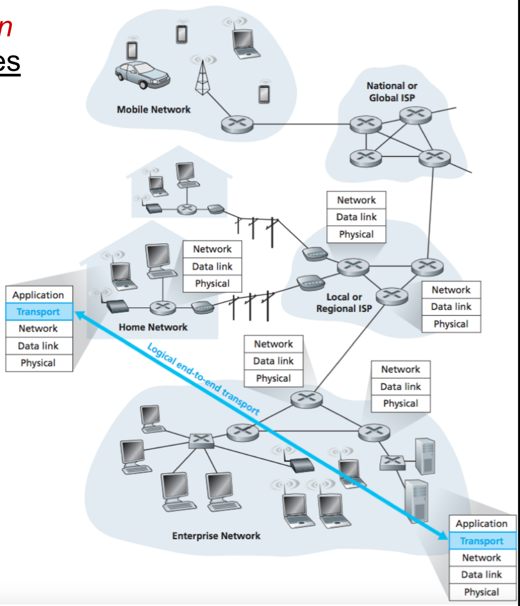
Roadmap:

3-3

CHAPTER 3

(3.1) Transport-Layer Services

- provide *logical communication* between <u>application processes</u> running on different hosts
- transport protocols run in end systems / hosts :
  - **<u>send side</u>**: breaks app messages into __________, passes to network layer
  - **<u>receive side</u>**: reassembles __________ into messages, passes to application layer
- more than one transport protocol available to apps
  - Internet: *TCP* and *UDP*



CHAPTER 3

Transport Layer vs. Network Layer

- **Transport layer:** Logical communication between ___________
  - relies on, enhances, network layer services
- **Network layer:** Logical communication between __________

Household analogy:

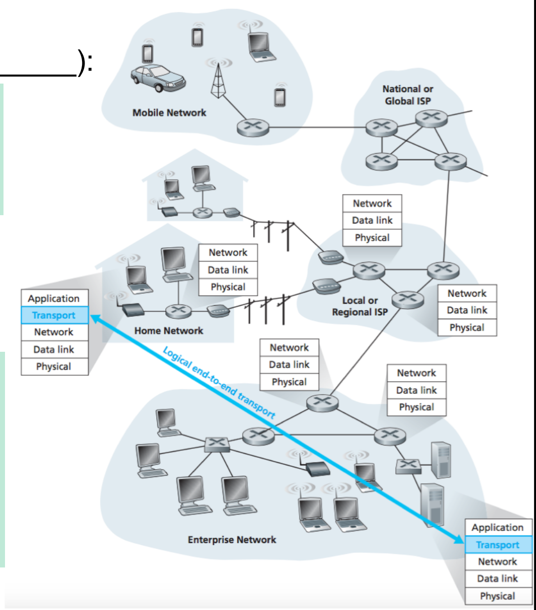
12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- application messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

CHAPTER 3

Internet Transport Layer Protocols

- Reliable, in-order delivery (_______):
  - connection setup
  - congestion control
  - flow control
- Unreliable, unordered delivery (________):
  - no-frills extension of "best-effort" IP
  - Services not available:
    - delay guarantees
    - bandwidth guarantees



CHAPTER 3

(3.2) 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

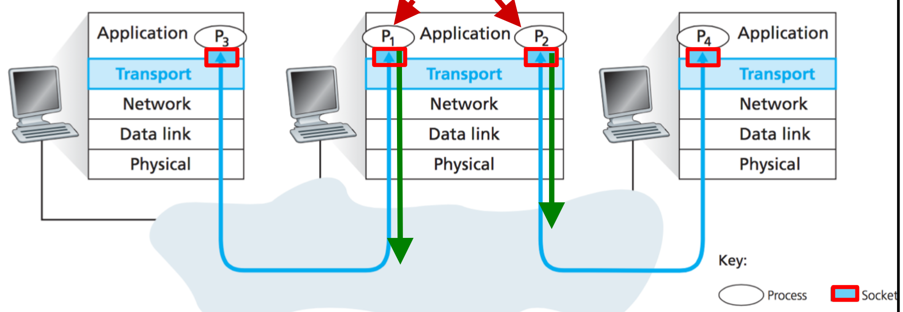


Figure: Transport-layer multiplexing and demultiplexing

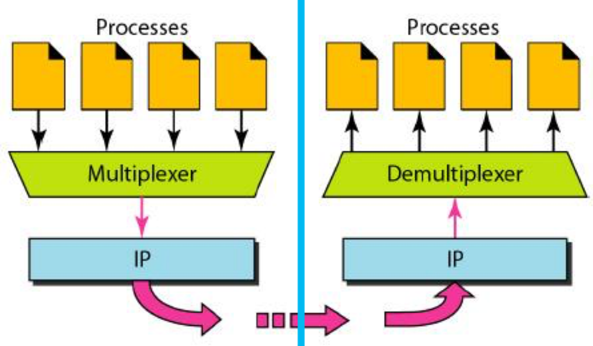
CHAPTER 3

Addition

(3.2) Multiplexing & Demultiplexing

Multiplexing at sender:

Demultiplexing at receiver:



http://www.eenadupratibha.net/pratibha/engineering/images/content_pics/content_three_Tra_layer_im2.JPG

3-8

CHAPTER 3

Demultiplexing

How it works?

- host receives datagrams :
  - each datagram has
    - ✓source IP address
    - ✓destination IP address
  - each datagram carries one transport-layer __________
  - each segment has
    - ✓source port number
    - ✓destination port number
- host uses *IP addresses* & *port numbers* to direct segment to appropriate ________

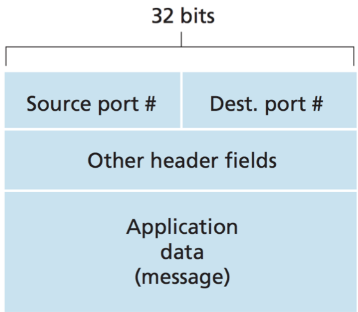


Figure: TCP/UDP transport-layer segment format

3-9

CHAPTER 3

Demultiplexing

(a) Connectionless

- ***Recall****:* when creating datagram to send into UDP socket, it fully identified by 2-tuples:
  - *destination IP address*
  - *destination port #*

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

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

3-10

CHAPTER 3

Demultiplexing

Example: (a) Connectionless

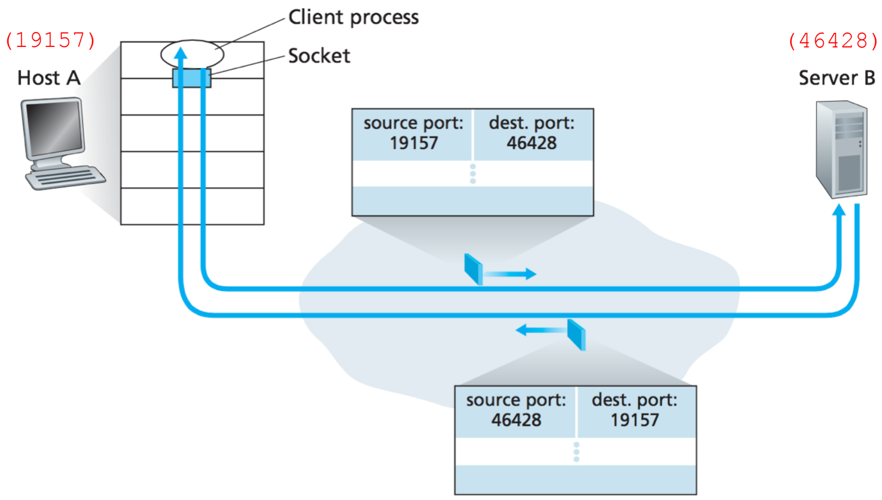


Figure: The inversion of source and destination port numbers

CHAPTER 3

Demultiplexing

(b) Connection-Oriented

- TCP socket identified by 4-tuples:
  - source IP address
  - source port number
  - ________________
  - ________________
- **Demux**: receiver 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-tuples
- web servers have different sockets for each connecting client:
  - non-persistent HTTP will have different socket for each request

CHAPTER 3

Demultiplexing

Example: (b) Connection-Oriented

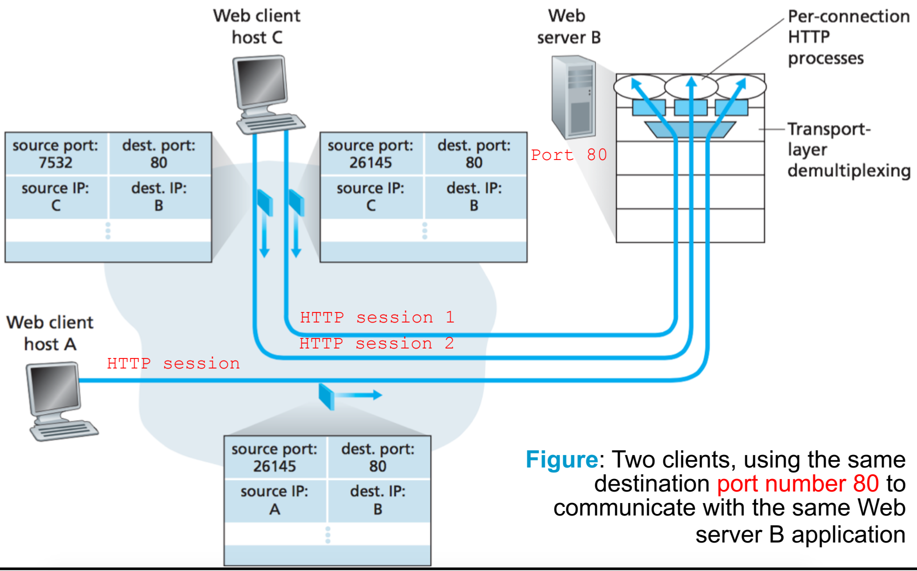


Figure: Two clients, using the same destination port number 80 to communicate with the same Web server B application

CHAPTER 3

(3.3) Connectionless Transport: UDP

- "*no frills*", "*bare bones*" Internet transport protocol
- "*best effort*" service, **UDP segments** may be:
  - ____________________
  - ____________________

Connectionless:

- No handshaking between UDP sender, receiver
- Each UDP segment handled independently of others

- UDP uses in:
  - streaming multimedia applications (loss tolerant, rate sensitive)
  - DNS
  - SNMP

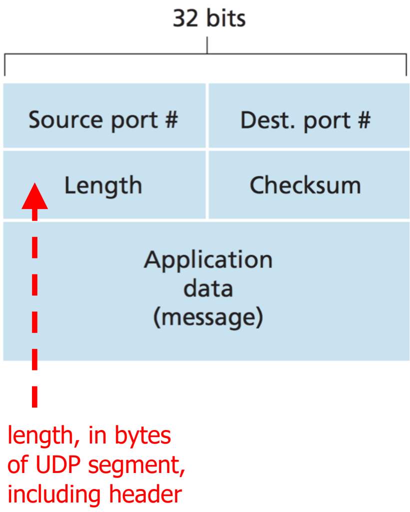
- Reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP (User Datagram Protocol)
DNS (Domain Name Services)
SNMP (Simple Network Management Protocol)

3-14

CHAPTER 3

UDP Segment Header



Why is there a UDP?

- **No connection** establishment (which can add delay)
- **Simple**: no connection state at sender, receiver
- **Small header size**
- **No congestion control**: UDP can blast away as fast as desired

3-15

CHAPTER 3

Self-Test

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	(a1) ________	(b1) ________
Remote terminal access	Telnet	TCP
Web	(a2) ________	(b2) ________
File transfer	(a3) ________	(b3) ________
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	(b4) ________
Internet telephony	typically proprietary	(b5) ________
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	(a4) ________	(b6) ________

Figure: Popular Internet applications and their underlying transport protocols

NFS (Network File System)
RIP (Routing Information Protocol)

CHAPTER 3

UDP Checksum

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

Sender:

- treat **segment** contents, including header fields, as sequence of 16-bit integers.
- *Checksum*: addition (one's complement sum) of **segment** contents.
- sender puts checksum value into UDP checksum field.

Receiver:

- compute checksum of received segment.
- check if computed checksum ***equals checksum*** field value:
  - NO - error detected
  - YES - no error detected.

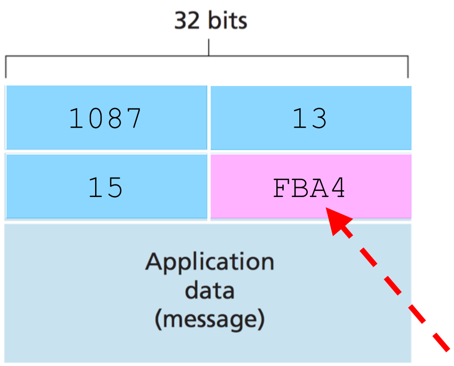
But maybe errors nonetheless? More later ….

3-17

CHAPTER 3

UDP Checksum

Calculation



At the sender:

00000100 00111111 → 1087
00000000 00001101 → 13
00000000 00001111 → 15

0000 0100 0101 1011 → SUM

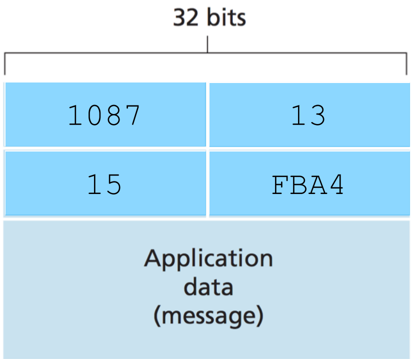
1st compliment:
1111 1011 1010 0100 → CHECKSUM

= $FBA4_{16}$

3-18

CHAPTER 3

UDP Checksum Calculation



At the receiver:

0000 0100 0011 1111 → Source Port
\+ 0000 0000 0000 1101 → Destination Port
\+ 0000 0000 0000 1111 → Length
\+ 1111 1011 1010 0100 → Checksum
1111 1111 1111 1111   **All 1's**

No Error !

3-19

CHAPTER 3

UDP Checksum Calculation (When Carryout Occurs)

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
																	1
Sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
Checksum (1's)		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Hexadecimal = ______$_{16}$

Note: When adding numbers, a carryout from the most significant bit (MSB) needs to be added to the result

CHAPTER 3

Exercise 3.1

Consider a sender host sends the segment with some contents given. Generate the checksum value.

50439	16397
15	Checksum
Application Data (Payload)	

3-21

CHAPTER 3

Exercise 3.2

Consider a receiver host received the segment with contents given, check if any error occurred.

1287	13
15	$7ADC_{16}$
Application Data (Payload)	

3-22

CHAPTER 3

(3.4) Principles of Reliable Data Transfer (`rdt`)

- important in *application*, *transport*, *link* layers
  - top-10 list of important networking topics!
- Characteristics of unreliable channel will determine complexity of *reliable data transfer* protocol (`rdt`)
- In this section we will examine the exploitation of TCP in many of the principles that we are about to describe.

3-23

CHAPTER 3

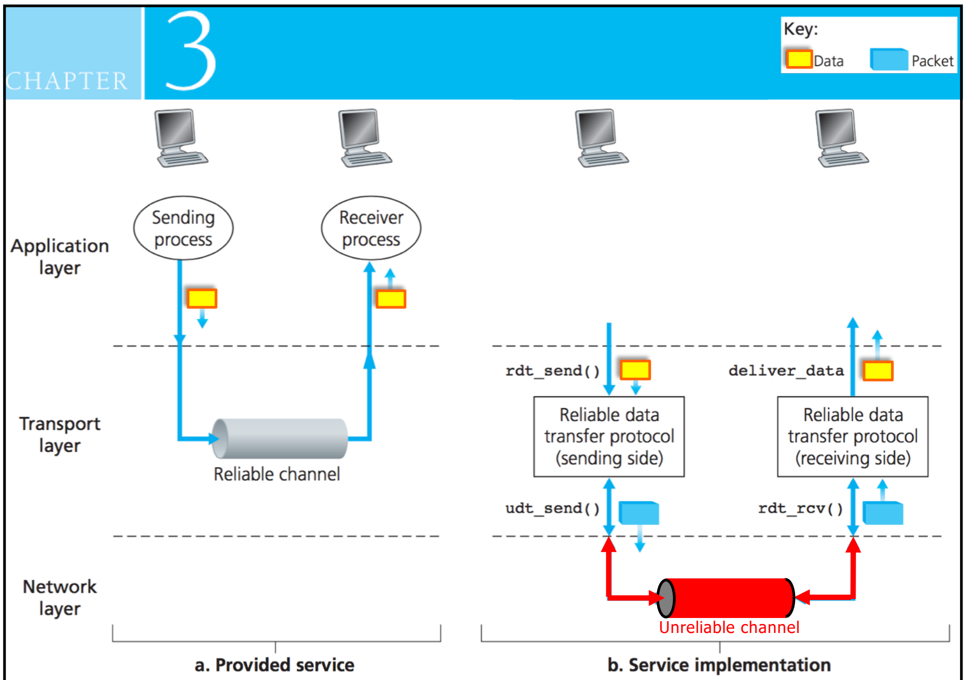


Figure: Reliable data transfer (`rdt`)

CHAPTER 3

Building a `rdt` Protocol

- Incrementally develop **sender**, **receiver** sides of *reliable data transfer protocol* (`rdt`)
- `rdt` protocol versions:
  - `rdt1.0`: reliable transfer over a reliable channel
    - underlying channel perfectly reliable
      - *no bit errors.*
      - *no loss* of packets.
    - *no* need to provide *feedback* to sender.
    - *no* need for the receiver to ask sender *to slow down* sending rate.
  - `rdt2.0`: channel with *bit errors*
  - `rdt3.0`: channels with *bit errors* and *loss* of packets.

unreliable channel

3-25

CHAPTER 3

Building a `rdt` Protocol

`rdt2.0`: Channel with bit errors

- Unreliable channel may flip bits in packet.
  - Checksum used **to detect** bit errors

Q: How **to recover** from errors?

Receiver explicitly tells sender that packet received `OK`

Receiver explicitly tells sender that packet had errors

- New mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - *Error detection*
  - *Receiver Feedback*: control messages (`ACK`, `NAK`) from receiver to sender.
  - *Retransmission*: sender retransmits packet on receipt of `NAK`.

CHAPTER 3

Building a `rdt` Protocol

Fatal flaw!

`rdt2.0`: Channel with bit errors

What happens if `ACK/NAK` corrupted?

- sender doesn't know what happened at receiver !
- can't just retransmit: possible **duplicates** !

Handling duplicates:

- sender retransmits current packet if `ACK/NAK` corrupted.
- sender adds ______________ to each packet.
- receiver discards (doesn't deliver up) duplicate packet.

`rdt2.0`: Stop and Wait protocol

Sender sends one packet, then waits for receiver's response.	Sender will not sends new packet, until receiver has received correct packet.

3-27

CHAPTER 3

Building a `rdt` Protocol

`rdt3.0`: Channel with errors and loss

New assumption:

Unreliable channel can also loss packets (data, `ACK`s)

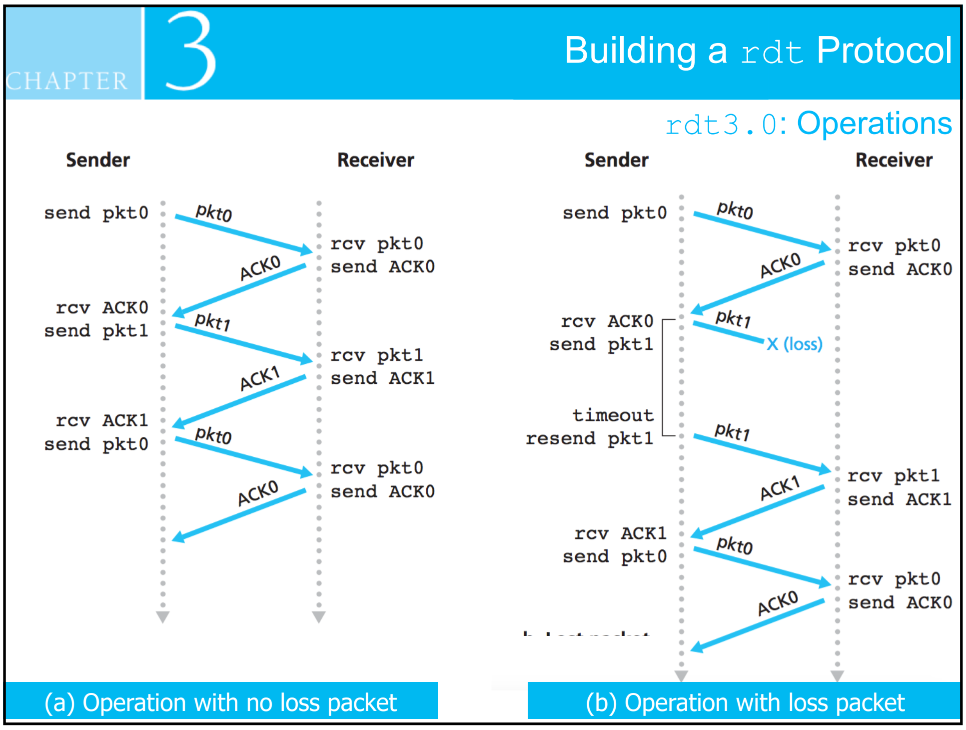
- checksum, sequence#, `ACK`s, retransmissions will be of help … but not enough.

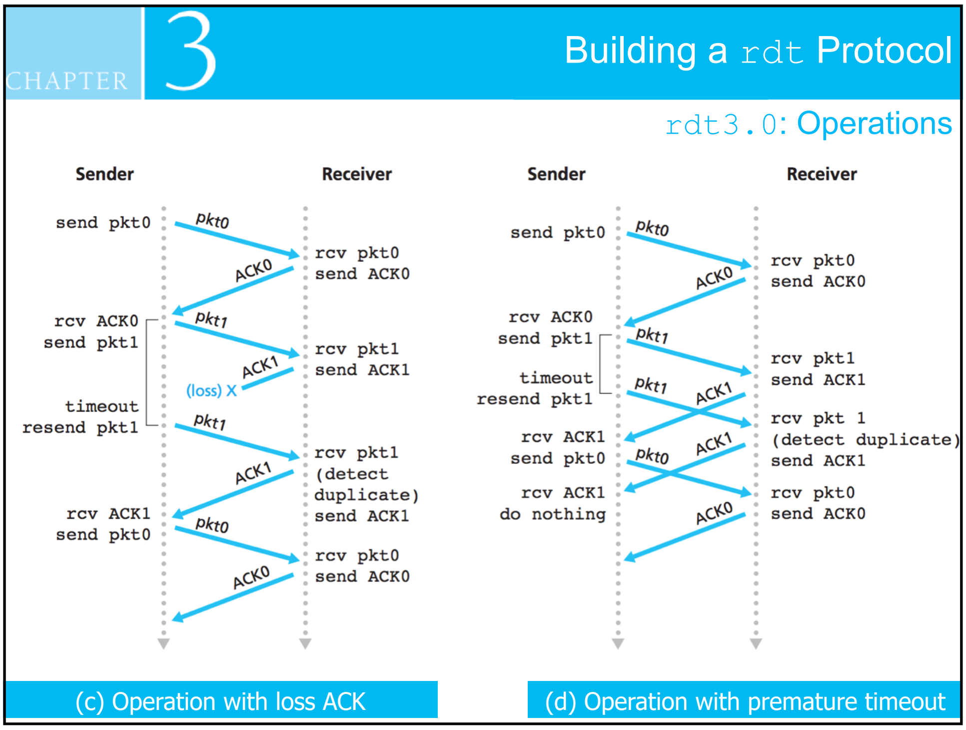
Approach:

Sender waits "reasonable" amount of time for `ACK`.

- Retransmits if no `ACK` received in this time.
- If packet (or `ACK`) just delayed (not lost):
  - retransmission will be __________, but sequence #'s already handles this.
  - receiver must specify sequence # of packet being `ACK`ed.
- Requires countdown timer.

3-28


CHAPTER 3
Building a rdt Protocol
rdt3.0: Operations
Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
(a) Operation with no loss packet
Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
X (loss)
timeout
resend pkt1
pkt1
rcv pkt1
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
(b) Operation with loss packet


CHAPTER 3
Building a rdt Protocol
rdt3.0: Operations
Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
ACK1
(loss) X
timeout
resend pkt1
pkt1
rcv pkt1
(detect
duplicate)
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
(c) Operation with loss ACK
Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
timeout
resend pkt1
pkt1
ACK1
rcv pkt 1
(detect duplicate)
send ACK1
rcv ACK1
send pkt0
pkt0
ACK1
rcv pkt0
send ACK0
rcv ACK1
do nothing
ACK0
(d) Operation with premature timeout

CHAPTER 3

Building a `rdt` Protocol

`rdt3.0`: Operations

- From previous 4 operations, `rdt3.0` sometimes known as:

`rdt3.0`: Alternating-bit protocol

Packet sequence# alternate between 0 and 1

CHAPTER 3

Exercise 3.3

Question:

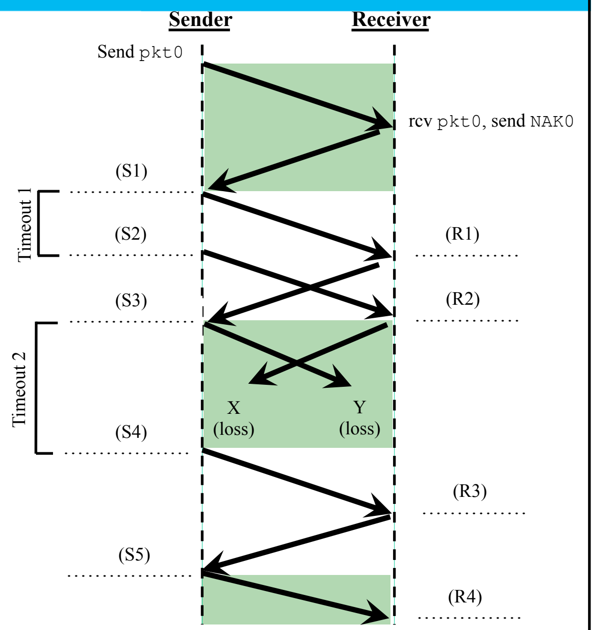
Supposed a sender has 3 packets to be sent to a receiver.

a) Complete the following figure by writing down the best answer for all S# and R#. Assume that no error after S1.

b) What are X and Y?

c) What is the problem between S1 and S2?

d) Is the any discard packet at receiver? Why?



CHAPTER 3

Building a `rdt` Protocol

`rdt3.0`: Performance Problem

- `rdt3.0` is a functionally correct protocol, but the performance is low;
- The performance problem is the fact that it is a ______________ protocol.

Example:

- Two hosts connected by a channel with a transmission rate, R, of 1*Gbps*;
- The RTT = 30 miliseconds;
- A host needs to transmit a packet, L, 1000 bytes

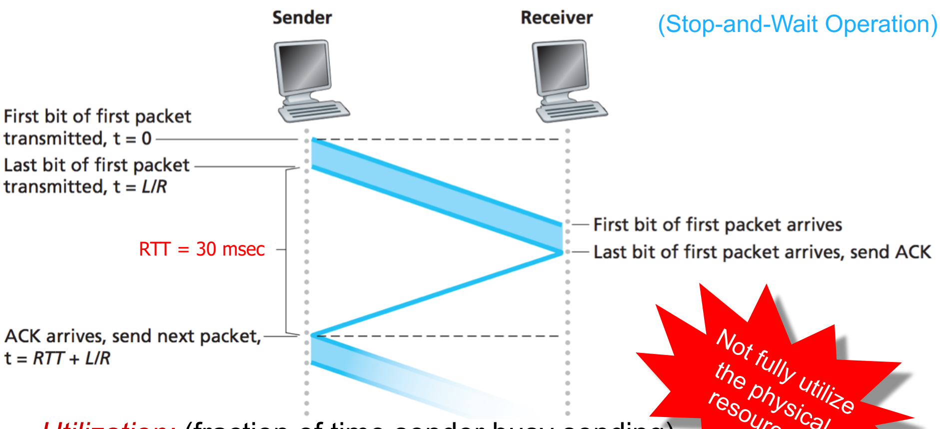
Transmission delay:

$$d_{trans} = \frac{L}{R}$$

CHAPTER 3

Building a `rdt` Protocol

(Stop-and-Wait Operation)



Utilization: (fraction of time sender busy sending)

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30 + 0.008} = 0.000267 = 0.0267\%$$

$Throughput =$

3-34

CHAPTER 3

Pipelined `rdt` Protocol

Solution

- ________: sender allows multiple, "*in-flight*", yet-to-be-acknowledged packets.
  - Range of sequence # must be increased.
  - Buffering more than one packet at sender and/or receiver.

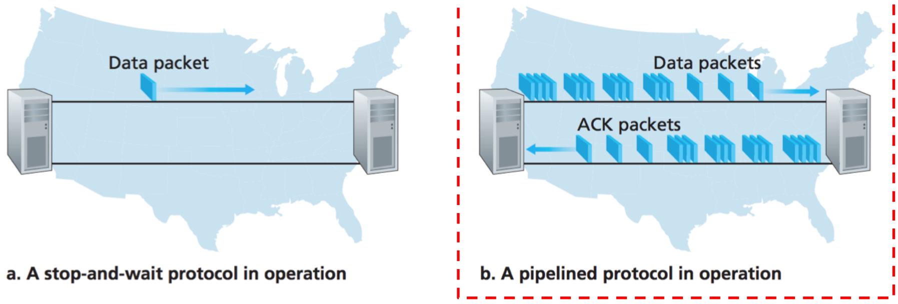


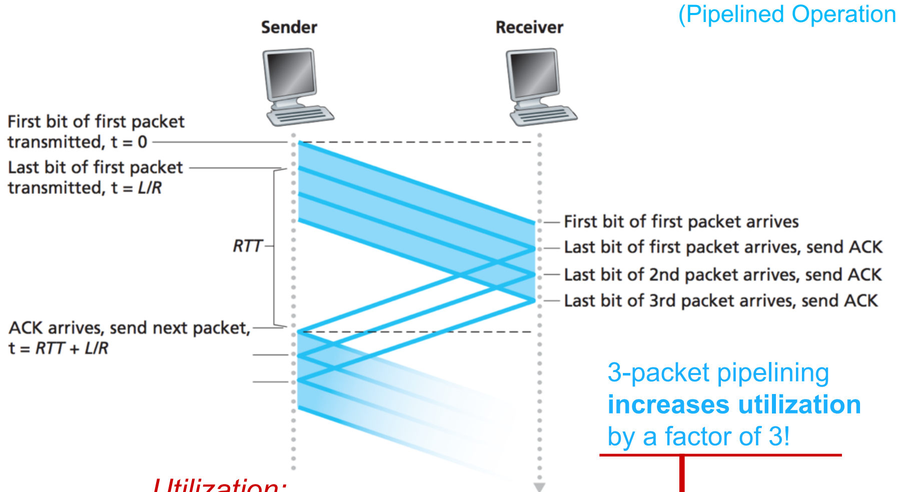
Figure: Stop-and-wait versus pipelined protocol

3-35

CHAPTER 3

Pipelined `rdt` Protocol

(Pipelined Operation)



3-packet pipelining **increases utilization** by a factor of 3!

Utilization:

$$U_{sender} = \frac{3L/R}{RTT + L/R} =$$

CHAPTER 3

Pipelined `rdt` Protocol

Pipelined Protocols

- The range of ***sequence #*** needed and the ***buffering*** requirements depend on the manner in which a data transfer protocol responds to:
  - lost, corrupted, and overly delayed packets.

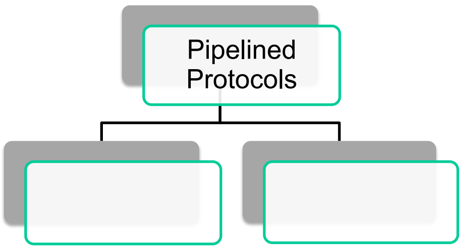


Figure: Two basic approaches of pipelined toward error recovery

3-37

CHAPTER 3

Pipelined `rdt` Protocol

- sender can have up to *N* un`ACK`ed packets in both pipeline protocol

Go-Back-*N*:

- receiver only sends *cumulative* *`ACK`*
  - doesn't `ACK` packet if there's a gap.
- sender has timer for oldest un`ACK`ed packet.
  - when timer expires, retransmit ***all*** un`ACK`ed packets.

Selective Repeat:

- receiver sends *individual* *`ACK`* for each packet.
- sender maintains timer for each un`ACK`ed packet.
  - when timer expires, retransmit ***only*** that un`ACK`ed packet.

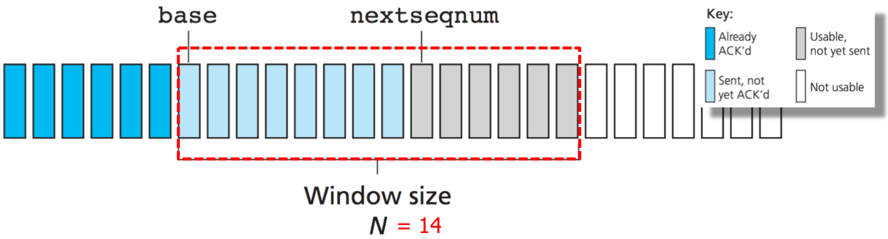
3-38

CHAPTER 3

Pipelined `rdt` Protocol

Go-Back-N (GBN): Sender

- "window" size *N* and each *k*-bit has `seq#` in packet header.
- "window" of up to *N*, consecutive un`ACK`ed packets allowed.



`ACK(n)`: `ACK`s all packets up to, including `seq#n` - *"cumulative `ACK`"*

- may receive duplicate `ACK`s (see receiver).

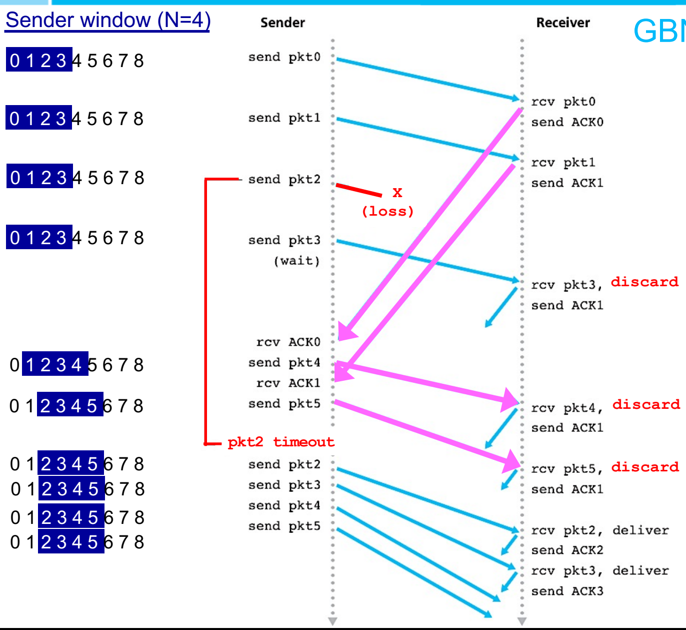
- timer for **oldest** in-flight packet
- `timeout(n)`: retransmit packet `n` and all higher `seq#` packets in window.

3-39

CHAPTER 3

Pipelined `rdt` Protocol

GBN: Operation



3-40

CHAPTER 3

Exercise 3.4

Suppose Host A and Host B use a *Go-Back-N* (GBN) protocol with size *N* = 3 and a long-enough range of sequence numbers.

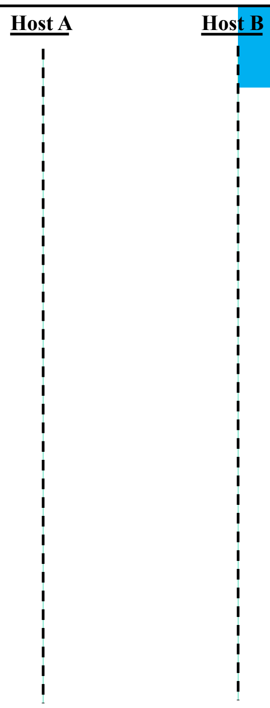
Assume Host A send six application messages to Host B and that all messages are correctly received, except for the first acknowledgment and the fifth data segment.

Draw a timing diagram, showing the data segments and the acknowledgements sent along with the corresponding sequence and acknowledge numbers, respectively.

(P19): page 321

2-41

Solution 3.4



CHAPTER 3

Pipelined `rdt` Protocol

Selective Repeat (SR)

- receiver *individually* acknowledges all correctly received packets:
  - *buffers packets*, as needed, for eventual in-order delivery to upper layer.
- sender only resends packets for which `ACK` not received
  - sender timer for each un`ACK`ed packet.
- sender window:
  - has *N* consecutive `seq#`'s.
  - limits `seq#`s of sent, un`ACK`ed packets (up-to "window size *N*") .

3-43

CHAPTER 3

Pipelined `rdt` Protocol

SR

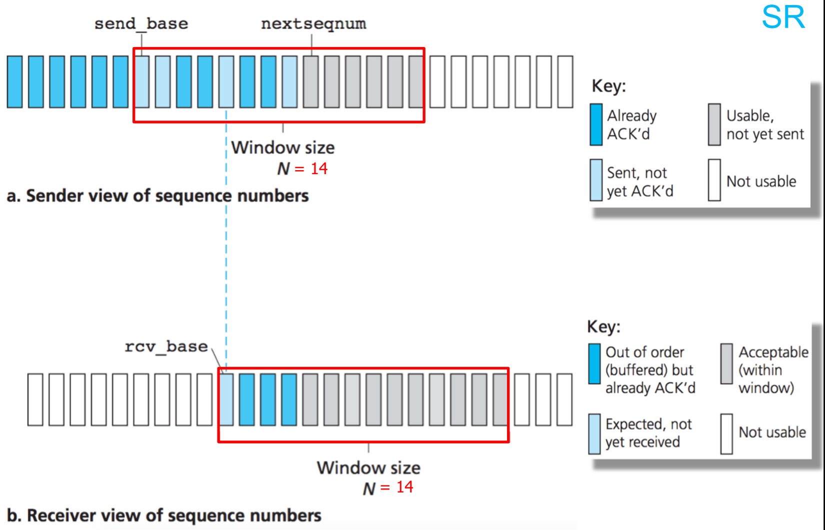


Figure: Selective-Repeat (SR) sender and receiver views of sequence-number space

3-44

CHAPTER 3

Pipelined `rdt` Protocol

SR: Events and Actions

Sender

Data from above (application):
- if next available `seq#` in window, send packet

`timeout(n):`
- resend packet `n`, restart timer

`ACK(n)` in `[send_base,send_base+N]`
- mark packet `n` as received
- if `n = send_base`, move forward window base to next unACKed with smallest `seq#`

Receiver

Packet `n` in `[rcv_base,rcv_base+N-1]`
- send `ACK(n)`
- If out-of-order : buffer
- If in-order : deliver (also deliver buffered, in-order packets), forward window to next not-yet-received packet

Packet `n` in `[rcv_base-N,rcv_base-1]`
- `ACK(n)`

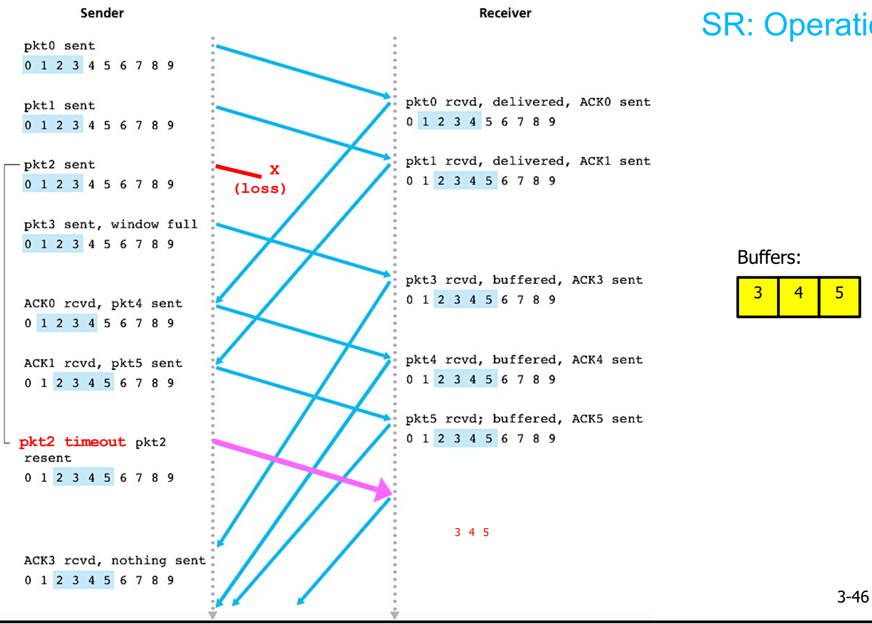
Otherwise: ignore the packet

3-45

CHAPTER 3

Pipelined `rdt` Protocol

SR: Operation



3-46

CHAPTER 3

Exercise 3.5

Suppose Host A and Host B use a *Selective Repeat* (SR) protocol with size *N* = 3 and a long-enough range of sequence numbers.

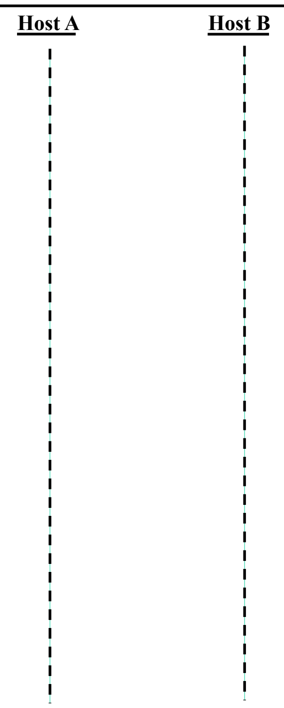
Assume Host A send six application messages to Host B and that all messages are correctly received, except for the first acknowledgment and the fifth data segment.

Draw a timing diagram, showing the data segments and the acknowledgements sent along with the corresponding sequence and acknowledge numbers, respectively.

(P19): page 321

2-47

Solution 3.5



CHAPTER 3

(3.5) Connection-Oriented Transport: TCP

Overview TCP

- Point-to-Point:
  - one sender, one receiver
- Reliable, in-order *byte stream:*
  - no “message boundaries”
- Pipelined:
  - TCP congestion and flow control set window size

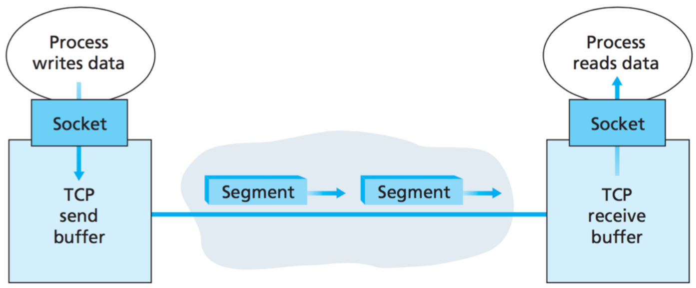


Figure: TCP send and receive buffers

3-49

CHAPTER 3

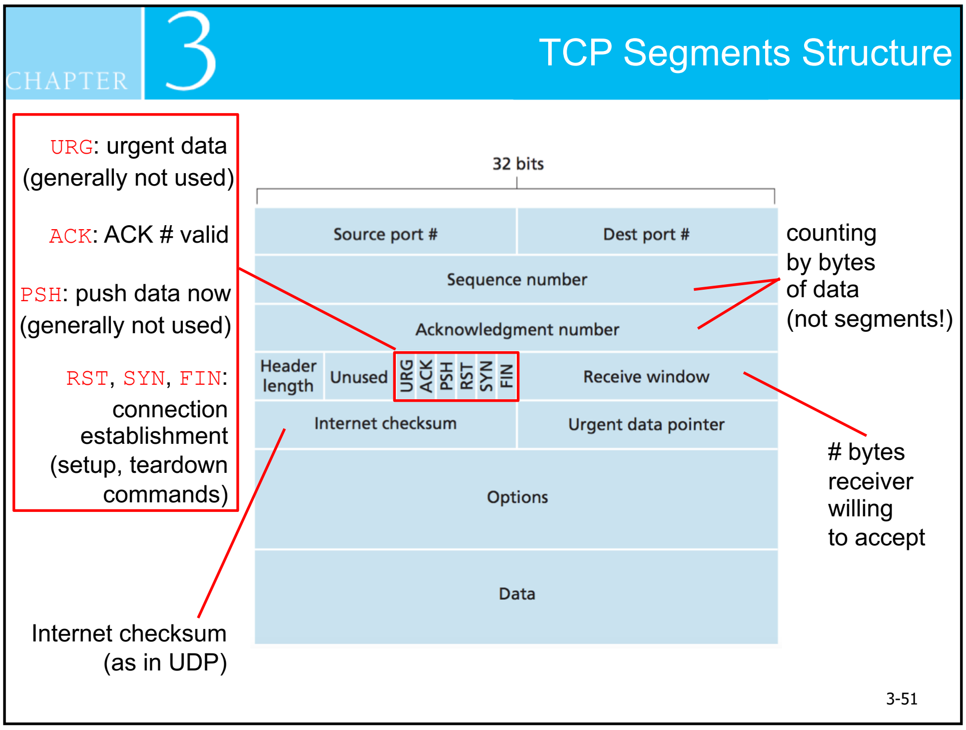
Overview TCP

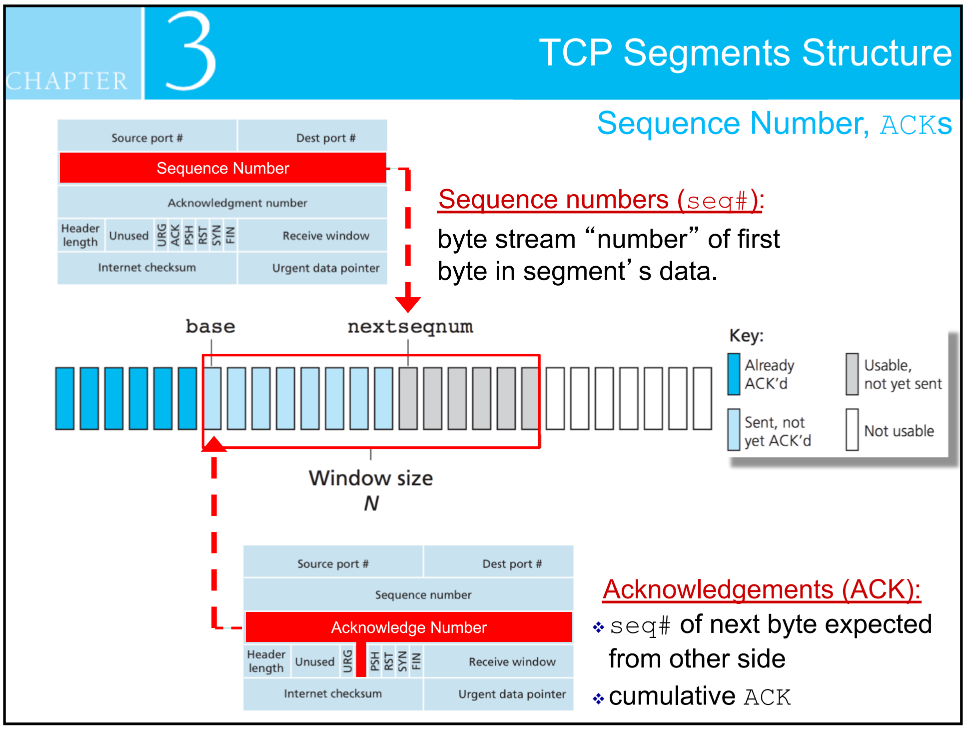
- Full-duplex data:
  - bi-directional data flow in same connection.
  - MSS: ______________________
  - **Example**:
    File size = 500 *Kb*, MSS = 1 *Kb*, so TCP construct 500 segments out of data stream.

$$\frac{500\,Kb}{1\,Kb}$$

- Connection-Oriented:
  - handshaking (exchange of control messages) inits sender, receiver state before data exchange.
- Flow controlled:
  - sender will not overwhelm receiver.

3-50


CHAPTER 3
TCP Segments Structure
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection establishment (setup, teardown commands)
32 bits
Source port #
Dest port #
Sequence number
Acknowledgment number
Header length
Unused
URG
ACK
PSH
RST
SYN
FIN
Receive window
Internet checksum
Urgent data pointer
Options
Data
counting by bytes of data (not segments!)
bytes receiver willing to accept
Internet checksum (as in UDP)
3-51


CHAPTER 3
TCP Segments Structure
Sequence Number, ACKs
Source port #
Dest port #
Sequence Number
Acknowledgment number
Header length
Unused
URG
ACK
PSH
RST
SYN
FIN
Receive window
Internet checksum
Urgent data pointer
Sequence numbers (seq#):
byte stream "number" of first byte in segment's data.
base
nextseqnum
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
Window size
N
Source port #
Dest port #
Sequence number
Acknowledge Number
Header length
Unused
URG
PSH
RST
SYN
FIN
Receive window
Internet checksum
Urgent data pointer
Acknowledgements (ACK):
seq# of next byte expected from other side
cumulative ACK

CHAPTER 3

TCP Segments Structure

Sequence Number, ACKs

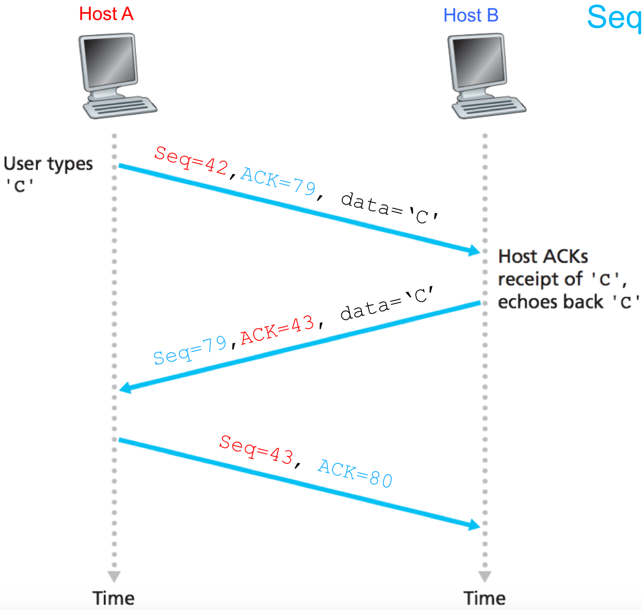


Figure: Sequence and acknowledgment numbers for a simple Telnet application over TCP

CHAPTER 3

TCP Round-Trip Time (RTT), 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`.

3-54

CHAPTER 3

TCP Reliable Data Transfer (`rdt`)

- TCP creates `rdt` service on top of IP's unreliable service by implementing:
  - pipelined segments.
  - cumulative `ACK`s.
  - single retransmission timer (refer to timer for oldest in-flight packet).

Let's initially consider simplified TCP sender:
- ignore duplicate `ACK`s
- Ignore :
  - flow control,
  - congestion control

- Retransmissions triggered by:
  - ________________.
  - duplicate `ACK`s.

Duplicate `ACK`,
indicating `seq#` of next expected byte.

(Due to some reason expected `seq#` is not received at receiver).

CHAPTER 3

TCP Reliable Data Transfer (`rdt`)

TCP Senders: Events and Actions

3 major events related to ***data transmission*** and ***retransmission*** in the TCP sender:

TCP Senders

Data received from application above
- 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 un`ACK`ed segment.
  - expiration interval: `TimeOutInterval`

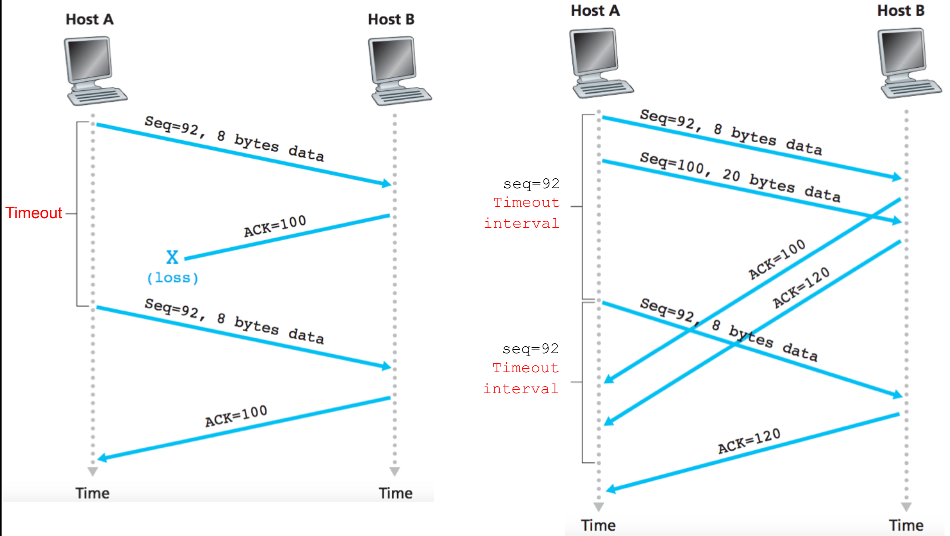
Timer Timeout
- Retransmit segment that caused timeout.
- Restart timer.

`ACK` Receipt
- if `ACK` acknowledges previously un`ACK`ed segments.
- update what is known to be `ACK`ed.
- start timer if there are still un`ACK`ed segments.

3-56

CHAPTER 3 TCP Reliable Data Transfer (rdt)

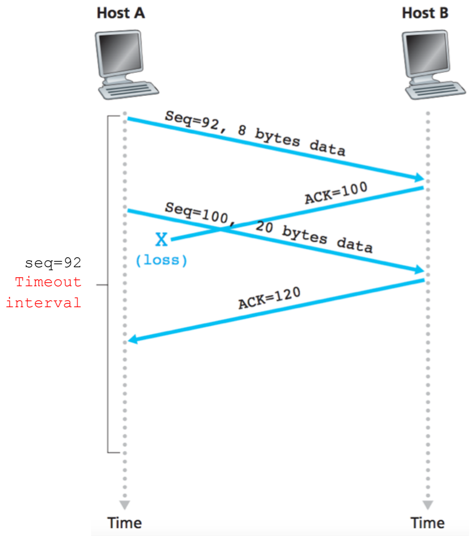
Transmission Scenarios



(a) Lost ACK scenario (b) Premature timeout

CHAPTER 3 TCP Reliable Data Transfer (rdt)

Transmission Scenarios



(c) Cumulative ACK

3-58

CHAPTER 3

TCP Reliable Data Transfer (rdt)

ACK Generation

Event	*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 *(retransmit – use oldest timer).*
Arrival of out-of-order segment higher-than-expect seq#. *Gap detected*	Immediately send *duplicate ACK*, indicating seq# of next expected byte *(TCP fast retransmit).*
Arrival of segment that partially or completely fills gap *(between seq#)*	*Immediate send ACK*, provided that segment starts at lower end of gap.

3-59

CHAPTER 3

TCP Reliable Data Transfer (rdt)

Fast Retransmit

- 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.

TCP fast retransmit

If sender receives 3 ACKs for same data ("________________ACKs"),

→ resend unACKed segment with smallest seq#

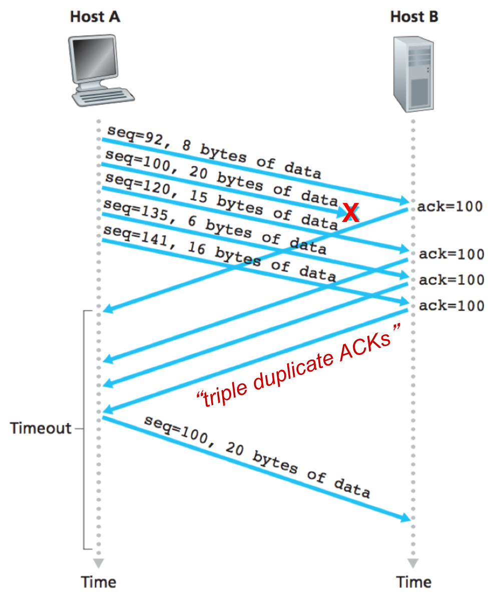
- likely that unACKed segment lost, so: don't wait for timeout.

3-60

CHAPTER 3

TCP Reliable Data Transfer (`rdt`)

Fast Retransmit



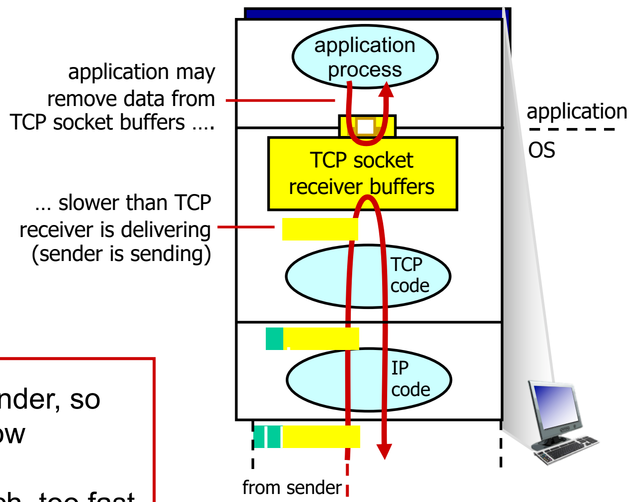
Event:
- Arrival of out-of-order segment.
- higher-than-expect `seq#`.
- Gap detected.

Action:
- Immediately send *duplicate `ACK`*.
- Indicating `seq#` of next expected byte.

Figure: TCP send and receive buffers

CHAPTER 3

TCP Flow Control



Flow Control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

Figure: Receiver protocol stack

CHAPTER 3

TCP Flow Control

- receiver "advertises" free buffer space by including rwnd value in TCP header of receiver-to-sender segments
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems auto adjust RcvBuffer
- sender limits amount of unACKed ("in-flight") data to receiver's rwnd value
- guarantees receive buffer will **not overflow**

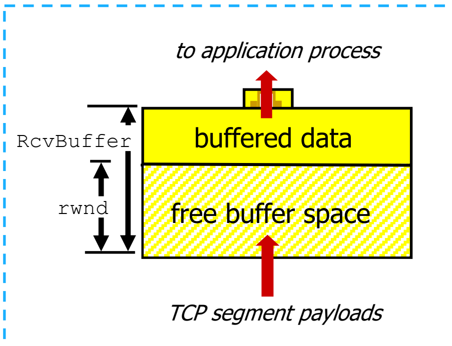


Figure: Receiver-side buffering

RcvBuffer → received buffer data

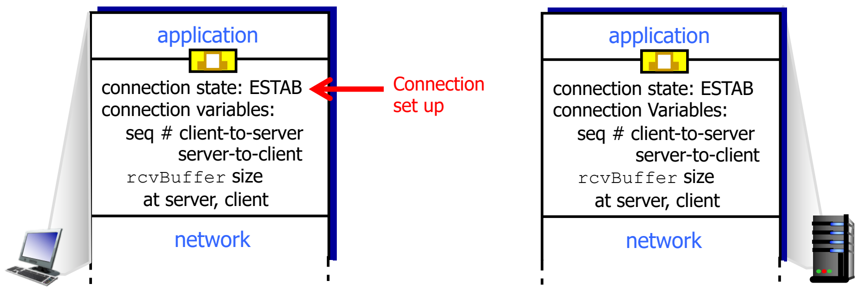
Rwnd → received window free buffer space

CHAPTER 3

TCP Connection Management

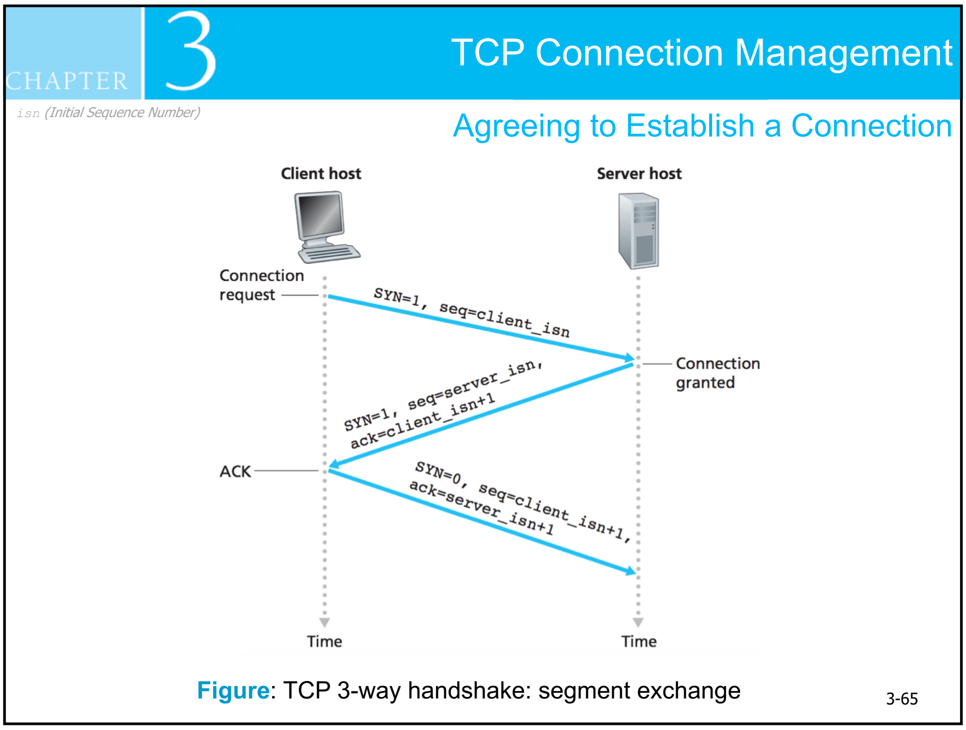
Before exchanging data, sender/receiver "*handshake*":

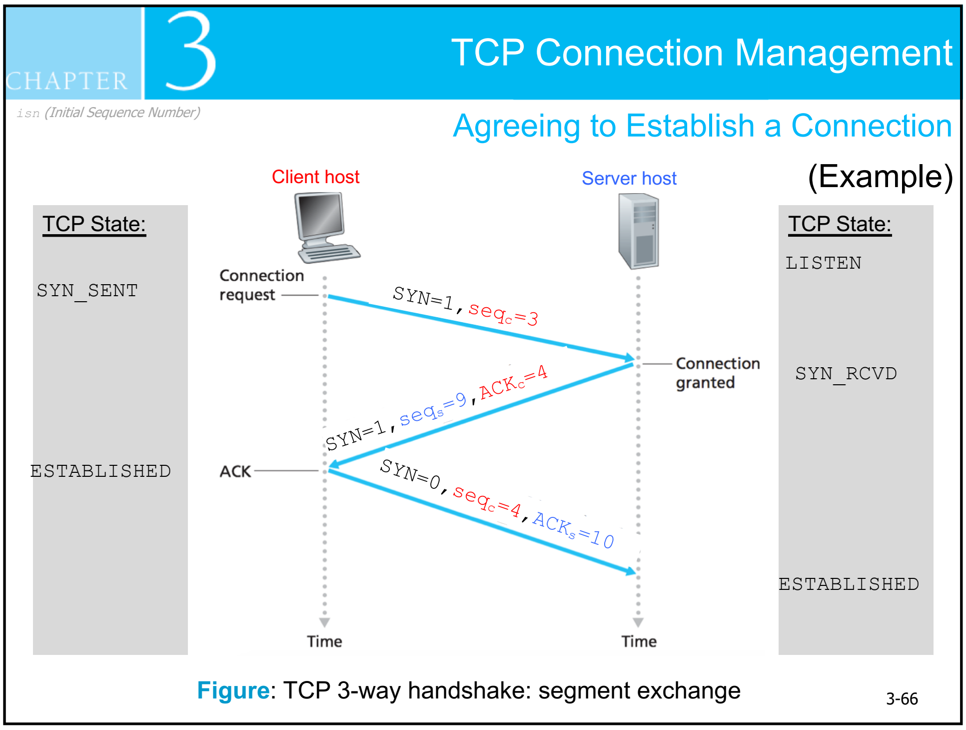
- Agree to establish connection (each knowing the other willing to establish connection).
- Agree on connection parameters.




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

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


CHAPTER 3
TCP Connection Management
isn (Initial Sequence Number)
Agreeing to Establish a Connection
Client host
Server host
Connection request
SYN=1, seq=client_isn
Connection granted
SYN=1, seq=server_isn, ack=client_isn+1
ACK
SYN=0, seq=client_isn+1, ack=server_isn+1
Time
Time
Figure: TCP 3-way handshake: segment exchange
3-65


CHAPTER 3
TCP Connection Management
isn (Initial Sequence Number)
Agreeing to Establish a Connection
(Example)
Client host
Server host
TCP State:
SYN_SENT
ESTABLISHED
Connection request
SYN=1, seqc=3
Connection granted
SYN=1, seqs=9, ACKc=4
ACK
SYN=0, seqc=4, ACKs=10
TCP State:
LISTEN
SYN_RCVD
ESTABLISHED
Time
Time
Figure: TCP 3-way handshake: segment exchange
3-66

CHAPTER 3

TCP Connection Management

Closing a Connection

- Client, server each close their side of connection
  - send TCP segment with `FIN` bit = 1.
- Respond to received `FIN` with `ACK`
  - on receiving `FIN`, `ACK` can be combined with own `FIN`
- Simultaneous `FIN` exchanges can be handled.

3-67

CHAPTER 3

TCP Connection Management

Closing a Connection

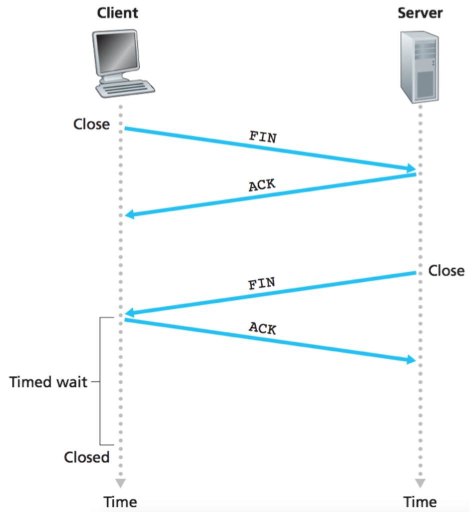


Figure: Closing a TCP connection

CHAPTER 3

TCP Connection Management

Closing a Connection (Example)

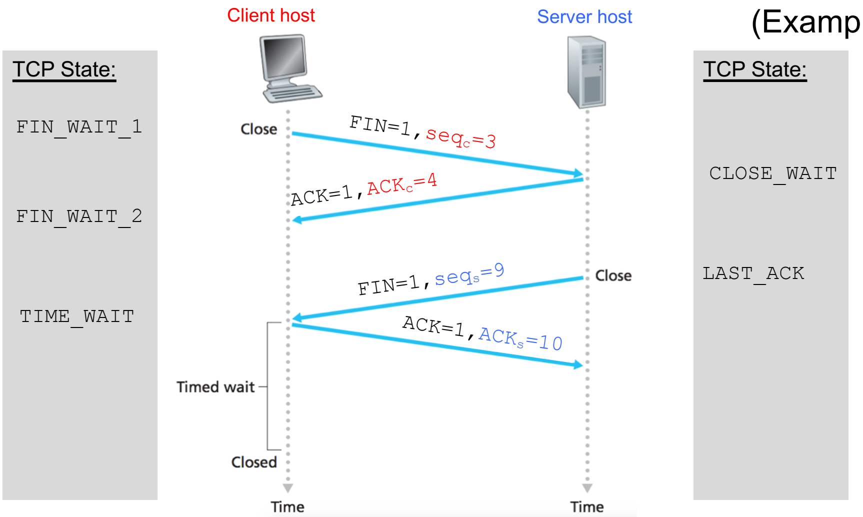


Figure: Closing a TCP connection

CHAPTER 3

(3.6) 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 (queuing in router buffers).

- a top-10 problem!

3-70

CHAPTER 3

SNA (System Network Architecture)
ECN (Explicit Congestion Notification)

Approaches toward congestion control

Approaches

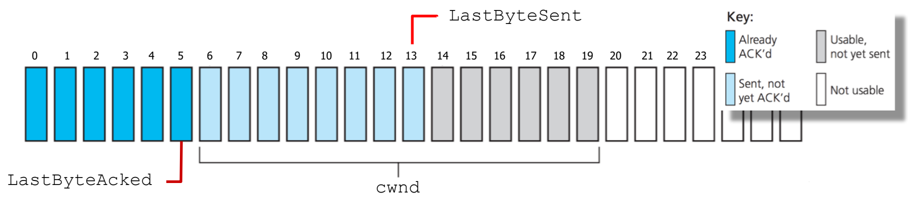
- <u>No explicit feedback from network</u>.
- Congestion inferred from end-system observed loss, delay (e.g. from timeout, duplicate `ACK`).
- Approach taken by TCP.

- <u>Routers</u> provide feedback to end systems:
  - single bit indicating congestion (as implemented by SNA, DECbit, TCP/IP ECN, ATM).
  - explicit rate for sender to send at.

CHAPTER 3

(3.7) TCP Congestion Control

cwnd (Congestion Window)



- sender limits transmission:

$$\boxed{\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}}$$

- `cwnd` is dynamic, function of perceived (recognized) network congestion

TCP sending rate:

- *roughly:* send `cwnd` bytes, wait RTT for `ACK`s, then send more bytes

$$\text{rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \textit{ bytes/sec}$$

CHAPTER 3

Overview

- There are generally **THREE** phases:
  - Slow Start;
  - Congestion Avoidance (CA);
  - Loss event because of:
    - ❑ ____________________
    - ❑ ____________________

3-73

CHAPTER 3

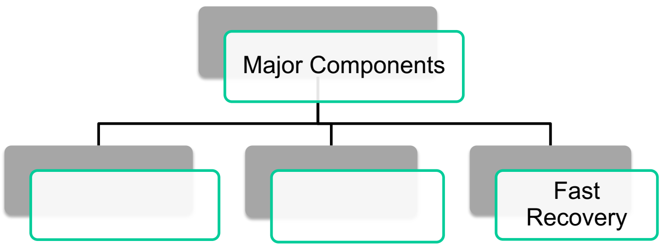


Figure: The components of TCP Congestion-Control algorithms

- Slow start and congestion avoidance are mandatory components of TCP Congestion-Control algorithms
  - **different**: how they increase the size of `cwnd` in response to received ACKs.

3-74

CHAPTER 3

TCP Slow Start & Congestion Avoidance (CA)

- 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.

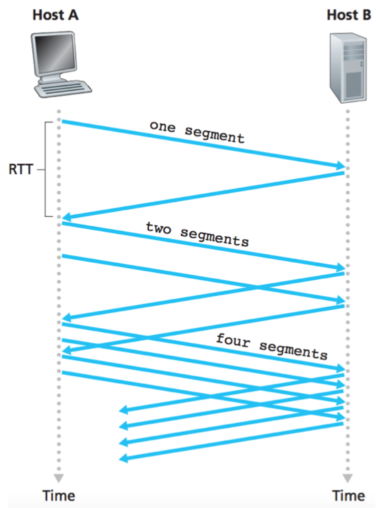


Figure: TCP slow start

MMS (Maximum Segment Size)
RTT (Round-Trip Time)

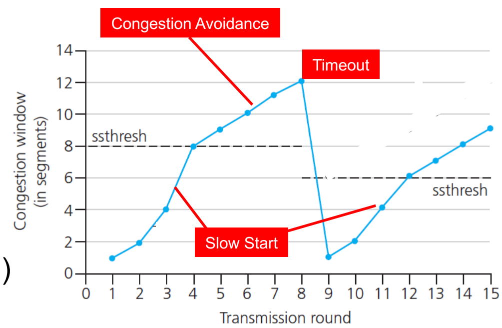
CHAPTER 3

TCP Slow Start & Congestion Avoidance (CA)

(Loss because of timeout)

Q: When should the exponential increase switch to linear?

A: When `cwnd` gets to 1/2 of its value before timeout. (Congestion Avoidance)



Implementation:

- variable `ssthresh` (slow-start threshold)
- on loss event:
  - `ssthresh` is set to 1/2 of `cwnd` just before loss event
  - Value of `cwnd` is set to 1 MSS (slow start)

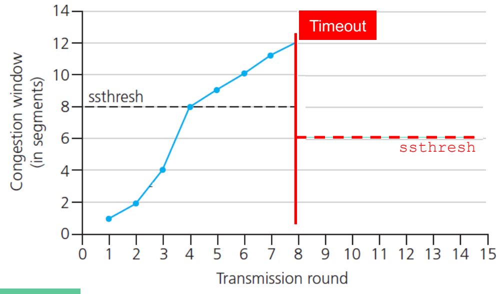
3-76

CHAPTER 3

TCP Slow Start & Congestion Avoidance (CA)

(Loss because of timeout)

Phase	TR	CW	SS	ssthresh
Slow Start (1)	1	1	1	8
	2	2	3	8
	3	4	7	8
	4	8	15	8
CA	5	9	24	8
	6	10	34	8
	7	11	44	8
	8	12	56	12 / 2 = 6



TR 1 to 4
- Slow Start, Exponential growth, `ssthresh=8`

TR 4 = `ssthresh` is detected and Congestion Avoidance (CA) starts

TR 5 to 8
- Operate at CA, Linear growth

Figure: Switching from slow start to CA)

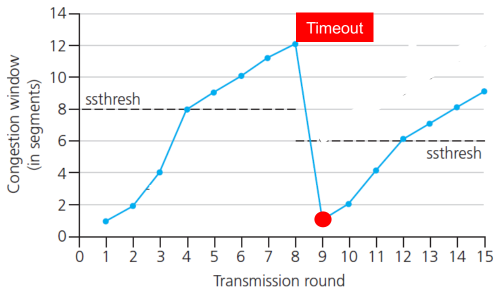
TR = Transmission Round
CW = Congestion Window
SS = Segment Send
`ssthresh` = Slow Start Threshold

CHAPTER 3

TCP Slow Start & Congestion Avoidance (CA)

(Loss because of timeout)

Phase	TR	CW	SS	ssthresh
Slow Start (2)	9	1	57	6
	10	2	59	6
	11	4	63	6
	12	6	69	6
CA	13	7	78	6
	14	8	86	6
	15	9	95	6



After TR 8
- Timeout is detected

TR 9 to 12 (refer table)
- CW=1, and `ssthresh=6`
- Start Slow, Exponential Growth

TR 12 to 15
- Operate at CA, Linear Growth

TR = Transmission Round
CW = Congestion Window
SS = Segment Send
`ssthresh` = Slow Start Threshold

3-78

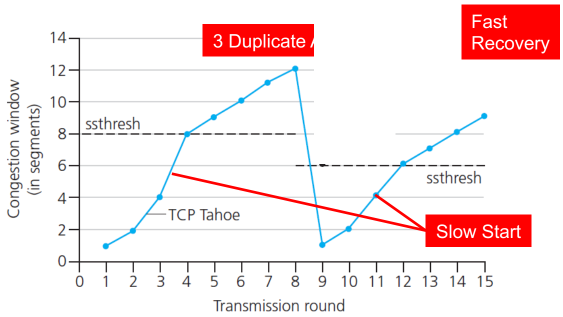
CHAPTER 3

TCP Fast Recovery

(Loss because of 3 Duplicate ACKs)

Earlier version of TCP
(TCP ________________)
entered Slow start

Newer version of TCP
(TCP ________________)
incorporated fast recovery



Implementation:

- on loss event, `ssthresh` is set to 1/2 of `cwnd` just before loss event
- `cwnd` is cut in half window then grows linearly

3-79

CHAPTER 3

TCP Fast Recovery

(Loss because of 3 Duplicate ACKs)

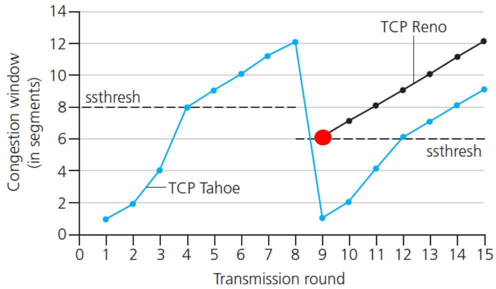
TR	CW	SS	ssthresh
8	12	56	12 / 2 = 6

TR	CW	SS	ssthresh
9	6	62	6
10	7	69	6
11	8	77	6
12	9	86	6
13	10	96	6
14	11	107	6
15	12	119	6

After TR 8 3DUP ACKs is detected

TR 9
→ CW=6

TR 9,10 to 15 TCP Reno
- Enters Fast Recovery
- Operate at CA
- Linear growth



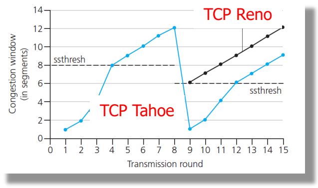
TR = Transmission Round
CW = Congestion Window
SS = Segment Send
`ssthreshold` = Slow Start Threshold

3-80

CHAPTER 3

TCP Fast Recovery

Detecting, Reacting to Loss Events



TCP Tahoe

- Loss indicated by timeout or 3 duplicate ACKs : (Slow Start)
  - `cwnd` set to 1 MSS;
  - window then grows exponentially (as in slow start) to `threshold`, then grows linearly

TCP Reno

- Loss indicated by ____________: (Slow Start)
  - `cwnd` set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by ____________ (Fast Recovery)
  - dup ACKs indicate network capable of delivering some segments
  - `cwnd` is cut in half window then grows linearly

CHAPTER 3

Exercise 3.6

Question:

Supposed host A connected to host B for transmitting segments over TCP with congestion control.

Assume that the initial threshold is 6 MSS.

If the timeout event occurred at transmission round (TR=8), answer the following questions:

a) Complete the table.

b) What is the new threshold after timeout?

c) What is/are the range of TRs involved in the congestion avoidance.

d) What is/are the range of TRs involved in the fast recovery?

e) At which TR the new threshold applied and how many segments sent at that TR?

Note: CW (Congestion Window), SS (Segment Send)

CHAPTER 3

Exercise 3.6

TR	CW	SS
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		

Note: CW (Congestion Window), SS (Segment Send)

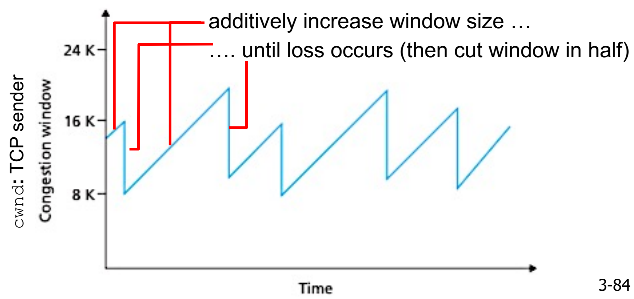
CHAPTER 3

TCP Congestion Control: Retrospective

Additive Increase Multiplicative Decrease (AIMD)

- TCP congestion control often referred as AIMD
- *Approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - ________________________: increase `cwnd` (congestion window) by 1 MSS every RTT until loss detected
  - ________________________: cut `cwnd` in half after loss

AIMD saw tooth behavior: probing for bandwidth



MMS (Maximum Segment Size)
RTT (Round-Trip Time)

CHAPTER 3

Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

Next:

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

3-85