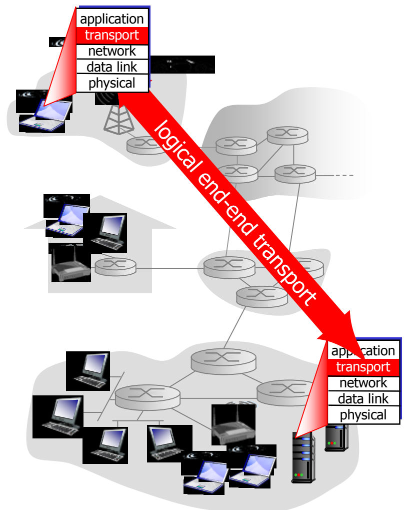
Transport Layer

our goals:

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

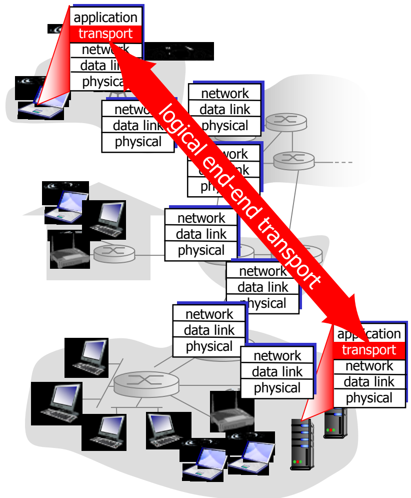
Transport services and protocols

- 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



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

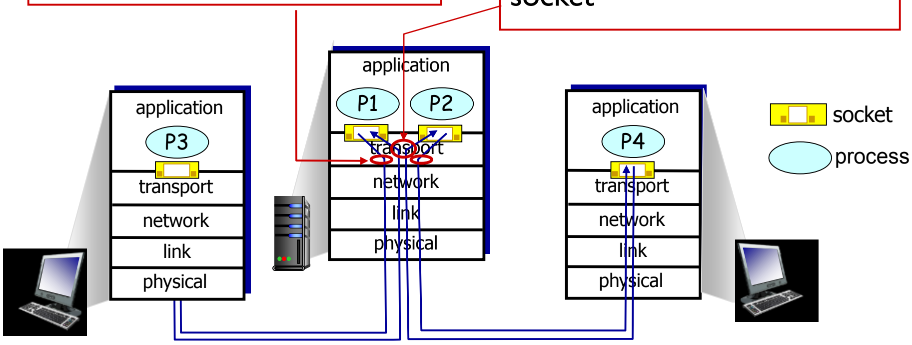


3

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



4

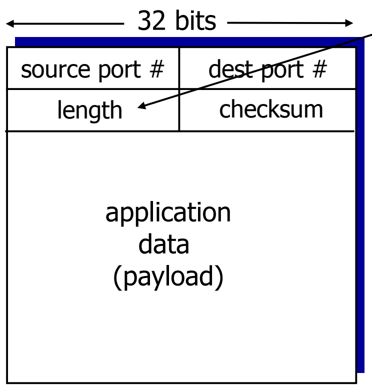
UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

5

UDP: segment header



length, in bytes of UDP segment, including header

UDP segment format

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

6

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:

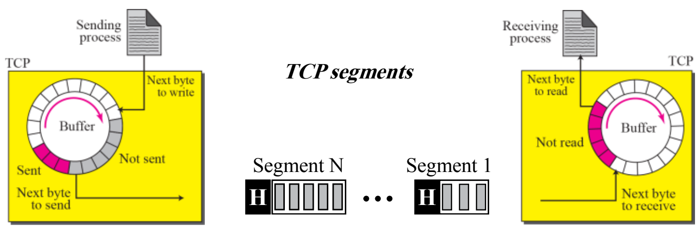
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

7

TCP Services

Table 15.1 *Well-known Ports used by TCP*

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20 and 21	FTP	File Transfer Protocol (Data and Control)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol



8

Example 1

Suppose a TCP connection is transferring a file of 5,000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1,000 bytes?

Solution

The following shows the sequence number for each segment:

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

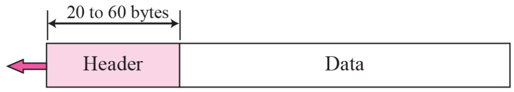
NOTE: *The bytes of data being transferred in each connection are numbered by TCP.*
The numbering starts with an arbitrarily generated number.

Note

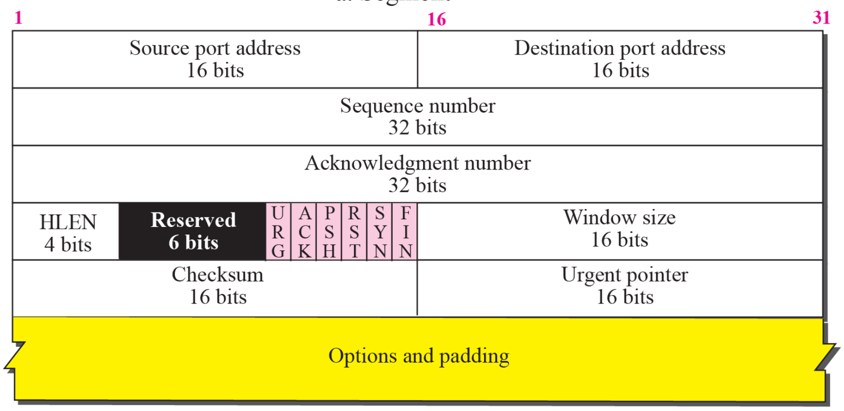
The value in the sequence number field of a segment defines the number assigned to the first data byte contained in that segment.

The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive. The acknowledgment number is cumulative.

TCP segment format



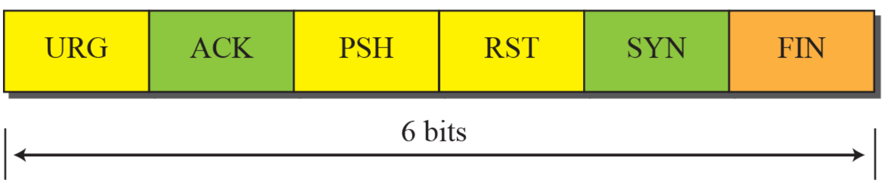
a. Segment



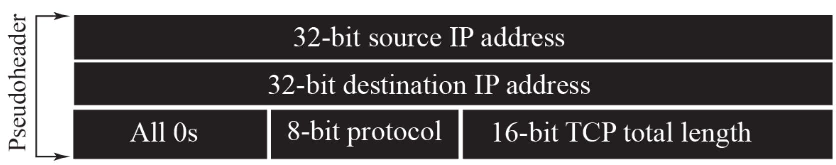
b. Header

Control field

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



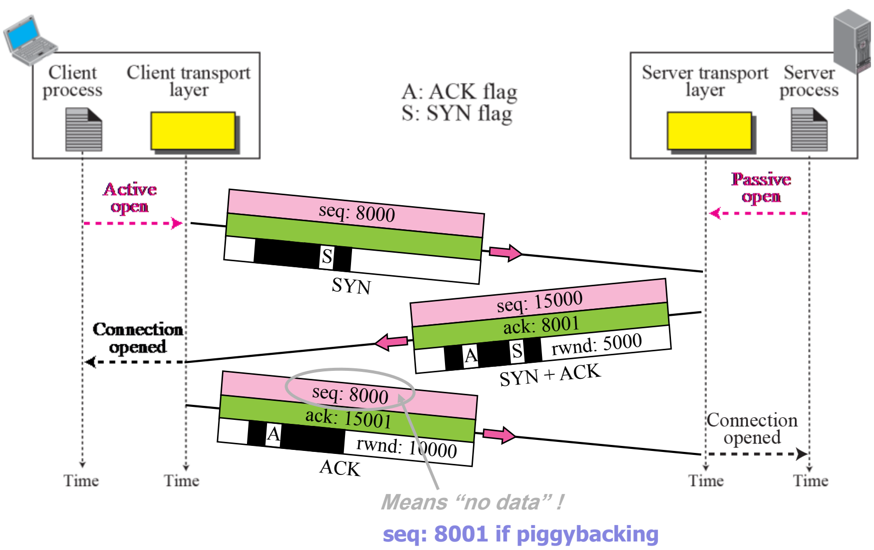
Pseudoheader added to the TCP segment



TCP CONNECTION

TCP is connection-oriented. It establishes a virtual path between the source and destination. All of the segments belonging to a message are then sent over this virtual path. You may wonder how TCP, which uses the services of IP, a connectionless protocol, can be connection-oriented. The point is that a TCP connection is virtual, not physical. TCP operates at a higher level. TCP uses the services of IP to deliver individual segments to the receiver, but it controls the connection itself. If a segment is lost or corrupted, it is retransmitted.

Connection establishment using three-way handshake



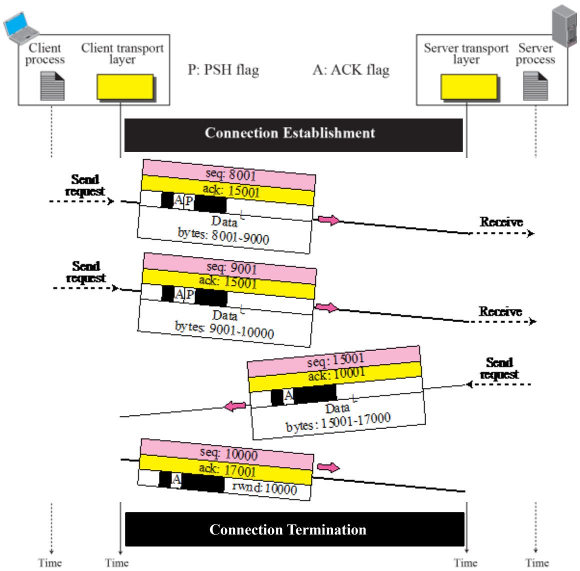
Note

A SYN segment cannot carry data, but it consumes one sequence number.

A SYN + ACK segment cannot carry data, but does consume one sequence number.

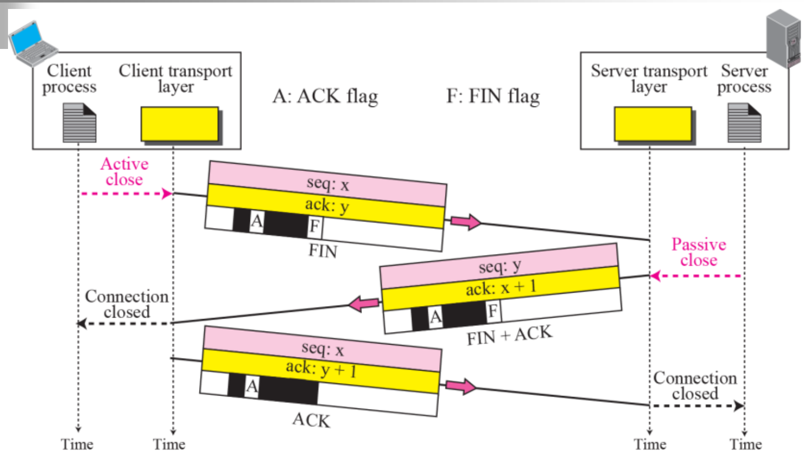
An ACK segment, if carrying no data, consumes no sequence number.

Data Transfer



Pushing data
Urgent data

Connection termination using three-way handshake

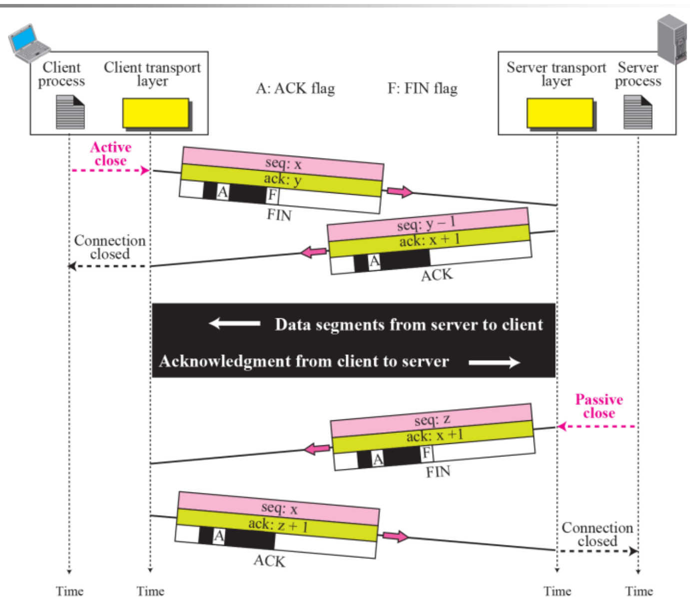


❖ ***The FIN segment consumes one sequence number if it does not carry data.***

❖ ***The FIN + ACK segment consumes one sequence number if it does not carry data***

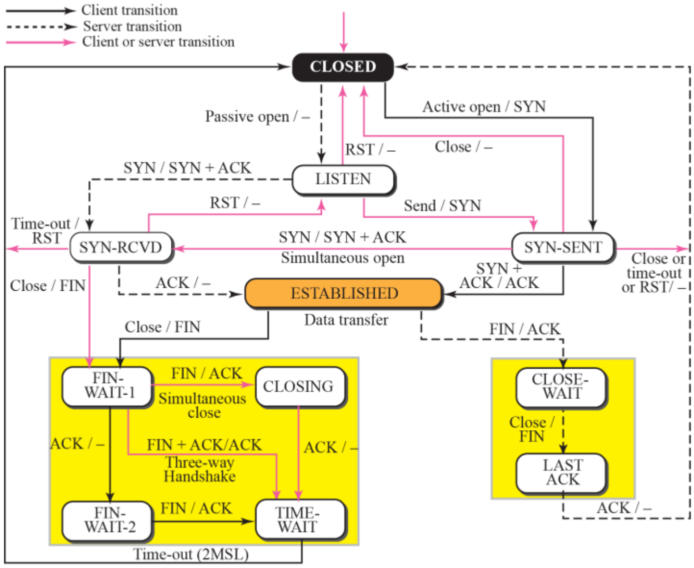
17

Half-Close



18

State transition diagram

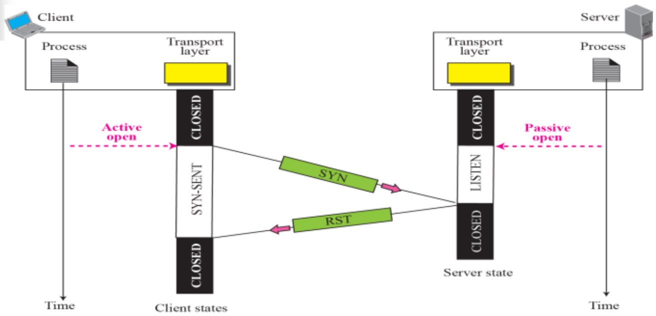


(half-close)

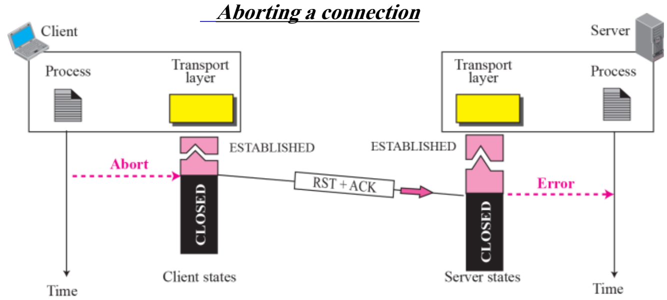
Table 15.2 *States for TCP*

State	*Description*
`CLOSED`	No connection exists
`LISTEN`	Passive open received; waiting for SYN
`SYN-SENT`	SYN sent; waiting for ACK
`SYN-RCVD`	SYN+ACK sent; waiting for ACK
`ESTABLISHED`	Connection established; data transfer in progress
`FIN-WAIT-1`	First FIN sent; waiting for ACK
`FIN-WAIT-2`	ACK to first FIN received; waiting for second FIN
`CLOSE-WAIT`	First FIN received, ACK sent; waiting for application to close
`TIME-WAIT`	Second FIN received, ACK sent; waiting for 2MSL time-out
`LAST-ACK`	Second FIN sent; waiting for ACK
`CLOSING`	Both sides decided to close simultaneously

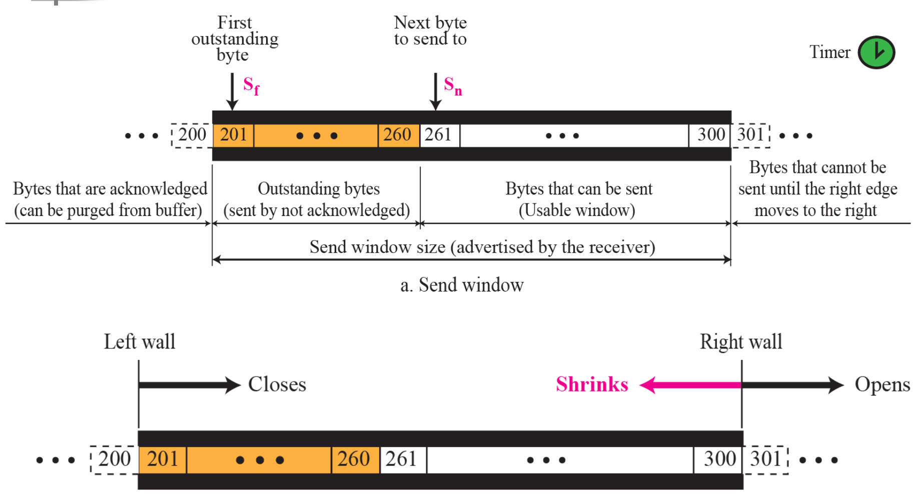
Denying a connection



Aborting a connection

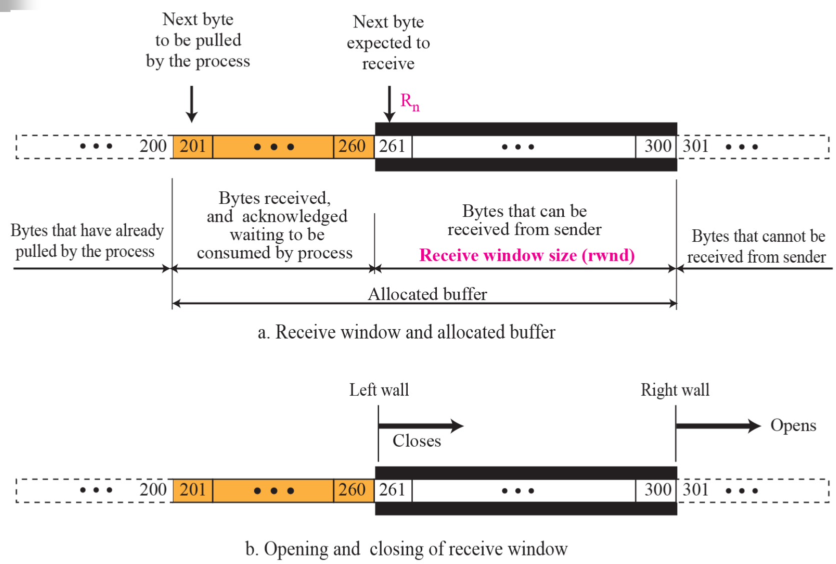


Send window in TCP



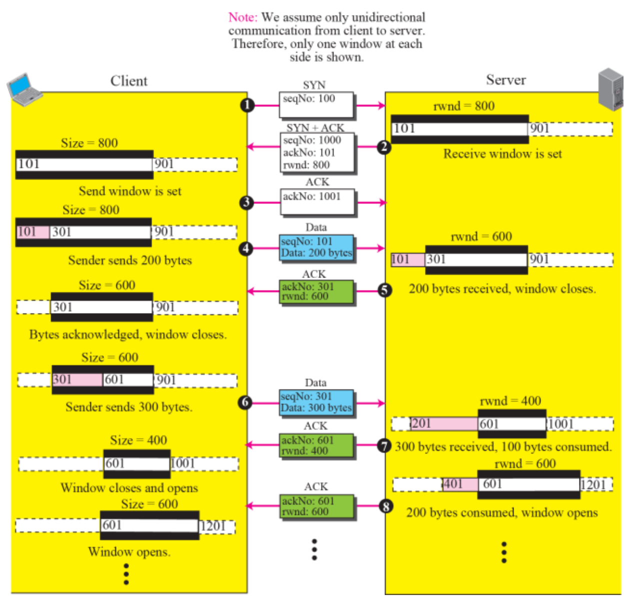
b. Opening, closing, and shrinking send window

Receive window in TCP



23

An example of flow control

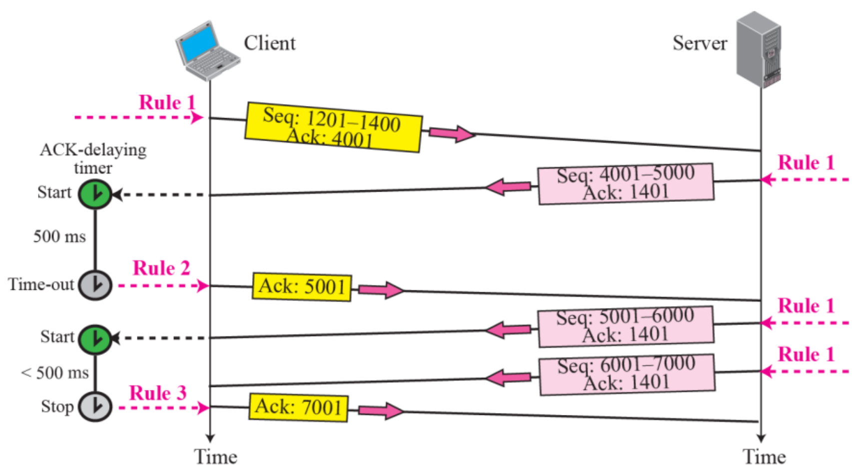


24

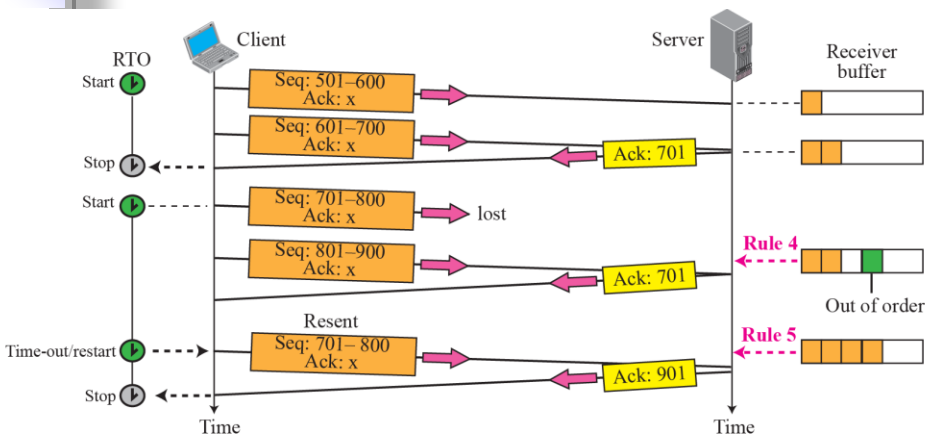
Rules for Generating ACK

1. When one end sends a data segment to the other end, it must include an ACK. That gives the next sequence number it expects to receive. (Piggyback)
2. The receiver needs to delay sending (until another segment arrives or 500ms) an ACK segment if there is only one outstanding in-order segment. It prevents ACK segments from creating extra traffic.
3. There should not be more than 2 in-order unacknowledged segments at any time. It prevent the unnecessary retransmission.
4. When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. (for fast retransmission)
5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected.
6. If a duplicate segment arrives, the receiver immediately sends an ACK.

Normal operation



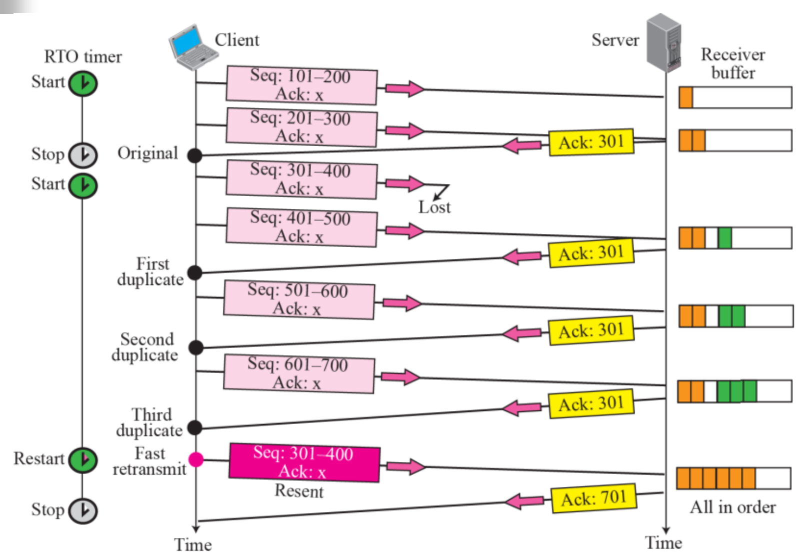
Lost segment



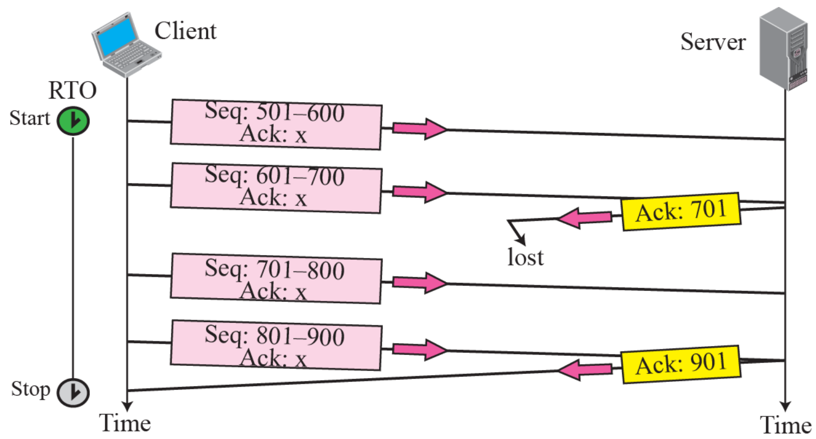
Note

The receiver TCP delivers only ordered data to the process.

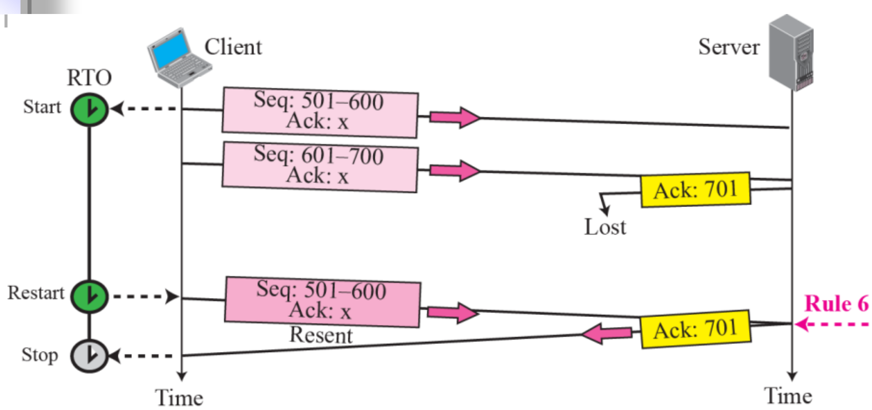
Fast retransmission



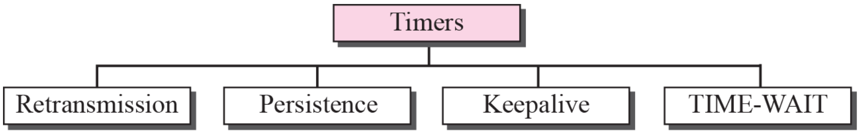
Lost acknowledgment



Lost acknowledgment corrected by resending a segment



TCP TIMERS



Retransmission timer is used when expecting an acknowledgment from the other end. This chapter looks at this timer in detail, along with related issues such as congestion avoidance.
Persistence timer keeps window size information flowing even if the other end closes its receive window.
Keepalive timer detects when the other end on an otherwise idle connection crashes or reboots.
TIME_WAIT (*2MSL)* timer measures the time a connection has been in the TIME_WAIT state..

31

Calculation of RTO

- **Smoothed RTT: RTT_S**
  - **Original → No value**
  - **After 1st measurement → $RTT_S = RTT_M$**
  - **2nd … → $RTT_S = (1-\alpha)*RTT_S + \alpha*RTT_M$**
- **RTT Deviation : RTT_D**
  - **Original → No value**
  - **After 1st measurement → $RTT_D = 0.5*RTT_M$**
  - **2nd … → R**
- **Retransmission Timeout (RTO)**
  - **Original → Initial value**
  - **After any measurement**
    → $RTO = RTT_S + 4RTT_D$
  - **$TT_D = (1-\beta)*RTT_D + \beta*|RTT_S - RTT_M|$**

32