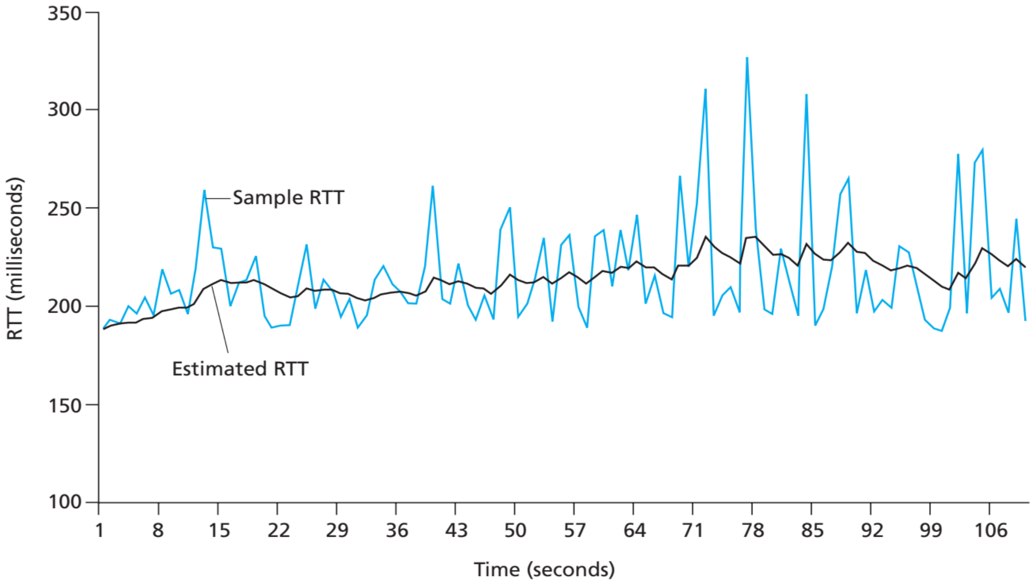
Module 3: Transport Layer
(Lecture – 3)

Dr. Nirnay Ghosh
Assistant Professor
Department of Computer Science & Technology
IIEST, Shibpur

TCP: Round Trip Time Estimation & Timeout

- TCP uses a timeout/retransmit mechanism to recover from lost segments
- Implementation challenge: how much larger the timeout should be than the connection's round trip-time (RTT)?
- Estimating RTT
  - *SampleRTT*: amount of time elapsed between when the segment is sent and the corresponding acknowledgement is received
  - It is measured for the segments which have been transmitted once
  - It fluctuates from segment to segment due to congestion in the routers and varying loads on the end systems
  - Estimation: take weighted combination of the previous value *of EstimatedRTT* and the new value of *SampleRTT*
  - Exponential Weighted Moving Average (EWMA): weight of a given *SampleRTT* decays exponentially fast as the updates proceed



RTT Samples and RTT Estimates

- *EstimatedRTT*: smoothens the variations in the *SampleRTT*
- *DevRTT*: Variability of the RTT
  - Estimates of how much *SampleRTT* typically deviates from the *EstimatedRTT*
  - EWMA of the difference between *SampleRTT* and *EstimatedRTT*

$$\text{DevRTT} = (1-\beta)\cdot\text{DevRTT} + \beta\cdot|\text{SampleRTT} - \text{EstimatedRTT}|$$

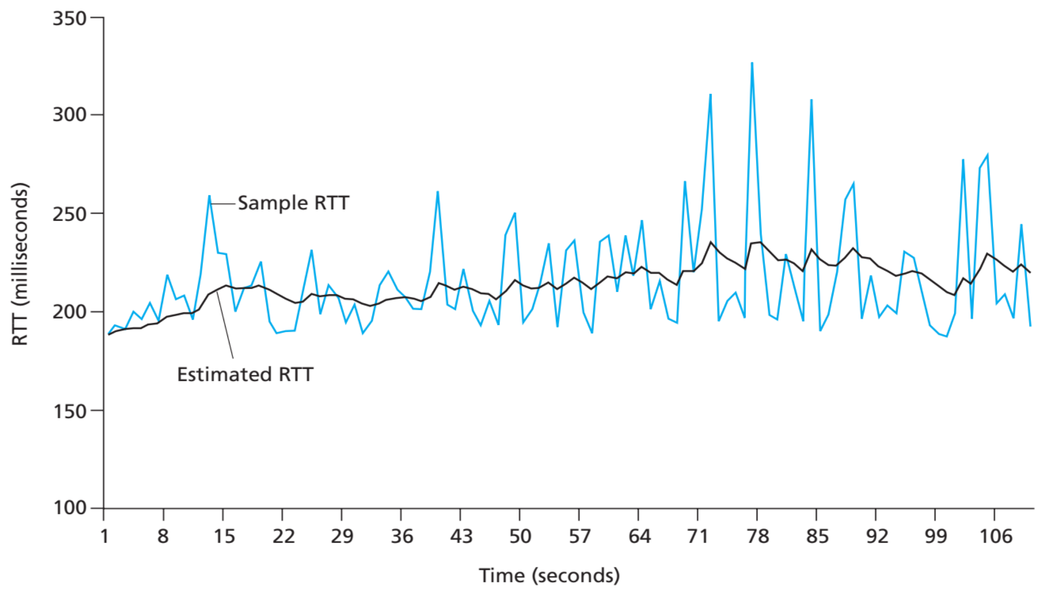
$$\text{EstimatedRTT} = (1-\alpha)\cdot\text{EstimatedRTT} + \alpha\cdot\text{SampleRTT}$$

- **Recommended value for β = 0.125**
- **Recommended value for α = 0.125**

TCP: Setting and Managing the Retransmission Timeout Interval

- TCP's timeout interval: should be greater than or equal to *EstimatedRTT*
- Otherwise: unnecessary retransmission would be sent
- Timeout interval: should not be too large than *EstimatedRTT*
  - TCP would not quickly retransmit the segment leading to larger data transfer delays
- Desirable to set the time equal to the *EstimatedRTT* plus some *margin*
- Margin:
  - Large: if there is a lot of fluctuation in the *SampleRTT* values
  - Small: if there is little fluctuation
- TCP's method for determining the retransmission timeout interval

```
TimeoutInterval = EstimatedRTT + 4 · DevRTT
```

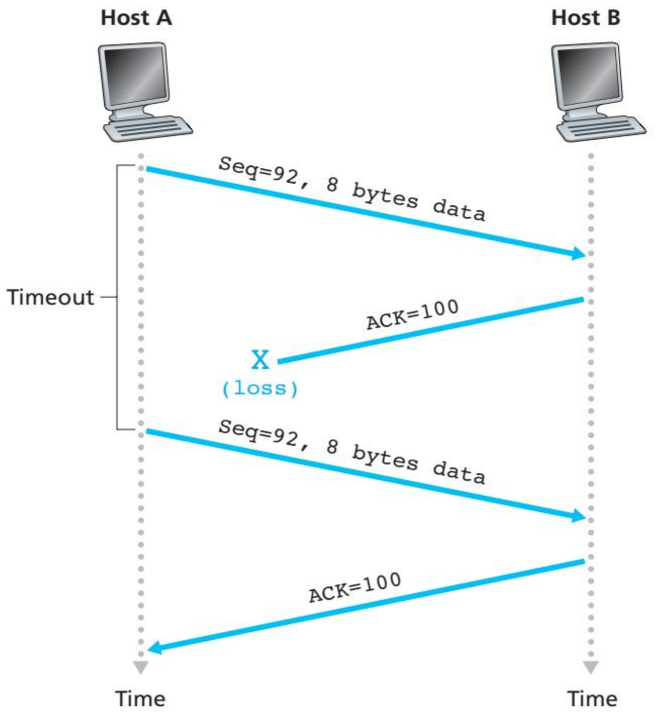


RTT Samples and RTT Estimates

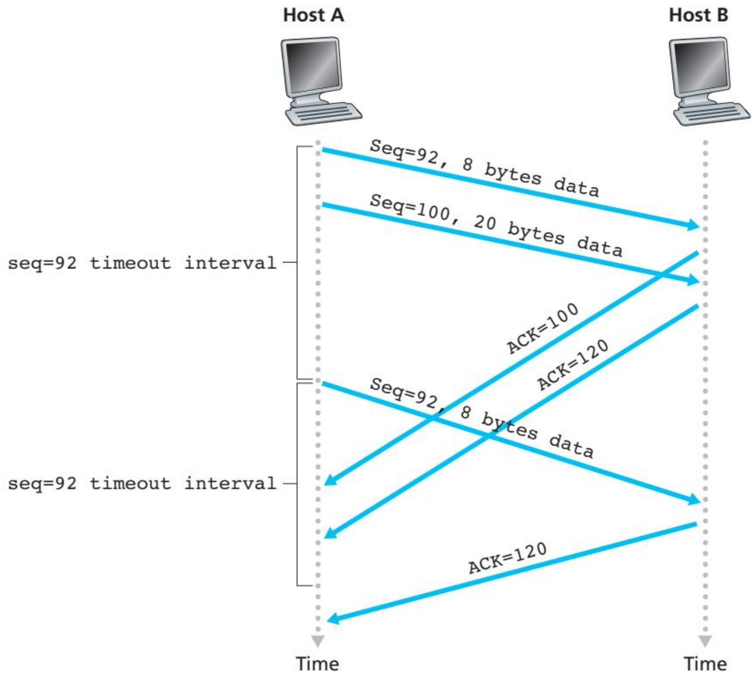
- Initial *TimeoutInterval*: 1 second (recommended)
- Subsequently the value is doubled to avoid premature timeout occurring for a segment that will soon be acknowledged
- As soon the segment is received, *EstimatedRTT* is updated
- *TimeoutInterval* is computed using the above formula

TCP: Reliable Data Transfer

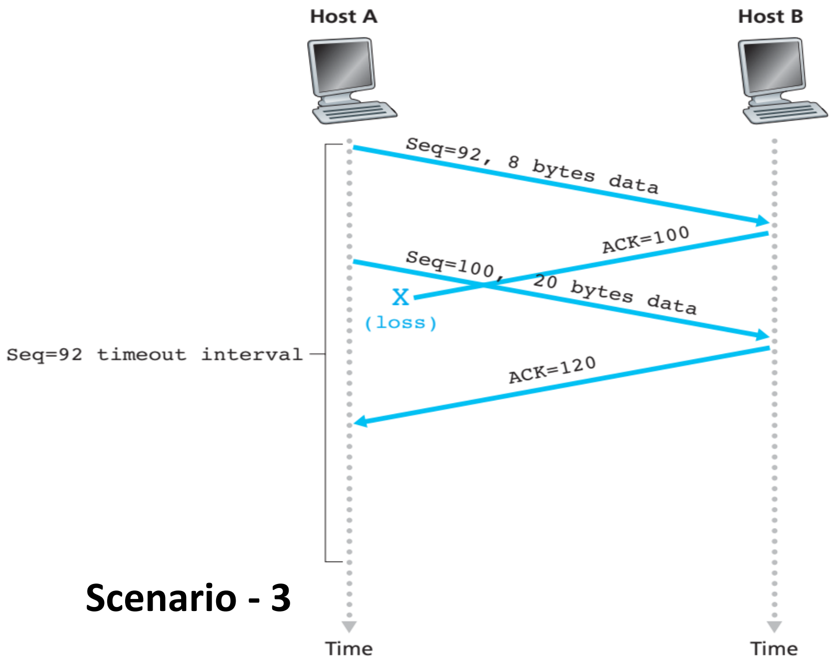
- Three major events related to data transmission and retransmission in the TCP sender are:
  - Data received from the upper-layer application
  - Timer timeout
  - ACK receipt
- Commonly occurring scenarios related to retransmission
  - 1. Retransmission due to lost acknowledgment
  - 2. Acknowledgement arrives after the timeout
  - 3. Cumulative acknowledgement avoiding retransmission



Scenario - 1



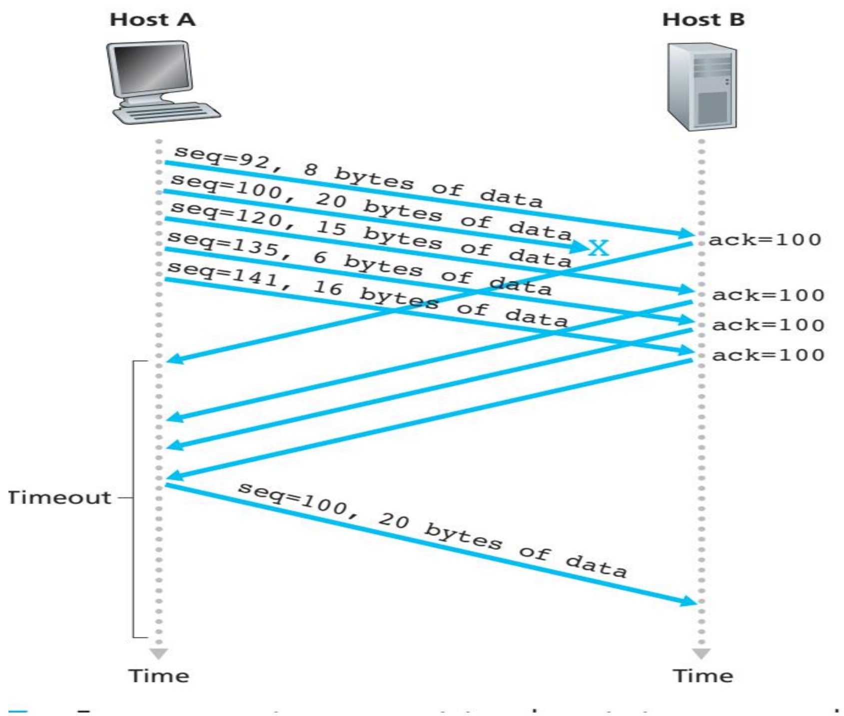
Scenario - 2



Scenario - 3

TCP: Reliable Data Transfer – Two approaches

- Doubling the *TimeoutInterval*
  - Time out event occurs first time: TCP retransmits the not-yet-acknowledged segment with smallest sequence number
  - For each retransmission, the *TimeoutInterval* is set to twice the previous value
  - Timer is started under two events: data received from application above and ACK received
    - *TimeoutInterval* – derived from the most recent values of *EstimatedRTT* and *DevRTT*
  - Provides a limited form of congestion control
  - Congestion: caused due to persistent retransmissions
  - TCP avoids congestion by retransmitting after longer and larger intervals

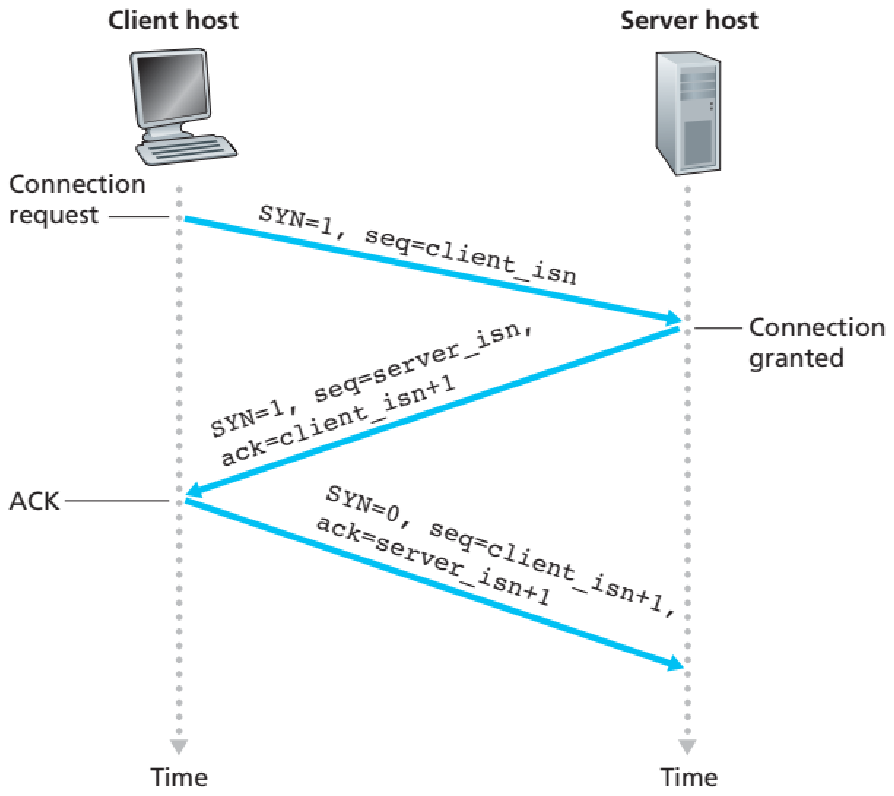


Fast Retransmit

- Fast Retransmit
  - Longer *TimeoutInterval* forces the sender to delay resending the lost packet
  - Increases the end-to-end delay
  - Duplicate ACK: used by senders to detect packet loss well before the timeout event occurs
  - TCP receiver: detects a gap in the data stream – result of segment loss or reordered within the network
    - Reacknowledges the missing segment multiple times (duplicate ACK) for which the sender has already received an earlier ACK
  - TCP sender: if receives three duplicate ACKs for the same data, performs a fast retransmission of the missing segments before the segment's timer expires

TCP: Connection Management

- Suppose a process running in one host (client) wants to initiate a connection with another host (server)
- Three-way handshake protocol: TCP connection establishment involves exchange of three segments:
    - Client-side TCP: sends segment (no application data) with SYN bit set to 1; randomly chooses an initial sequence number (*client_isn*); inserts *client_isn* in the sequence number field of the TCP segment
    - Server-side TCP: allocates TCP buffers and variables to the connection; sends a connection-granted segment (no application data) with - SYN bit: 1, acknowledgement number field: *client_isn + 1*; sequence number field: initial sequence number (*server_isn*)

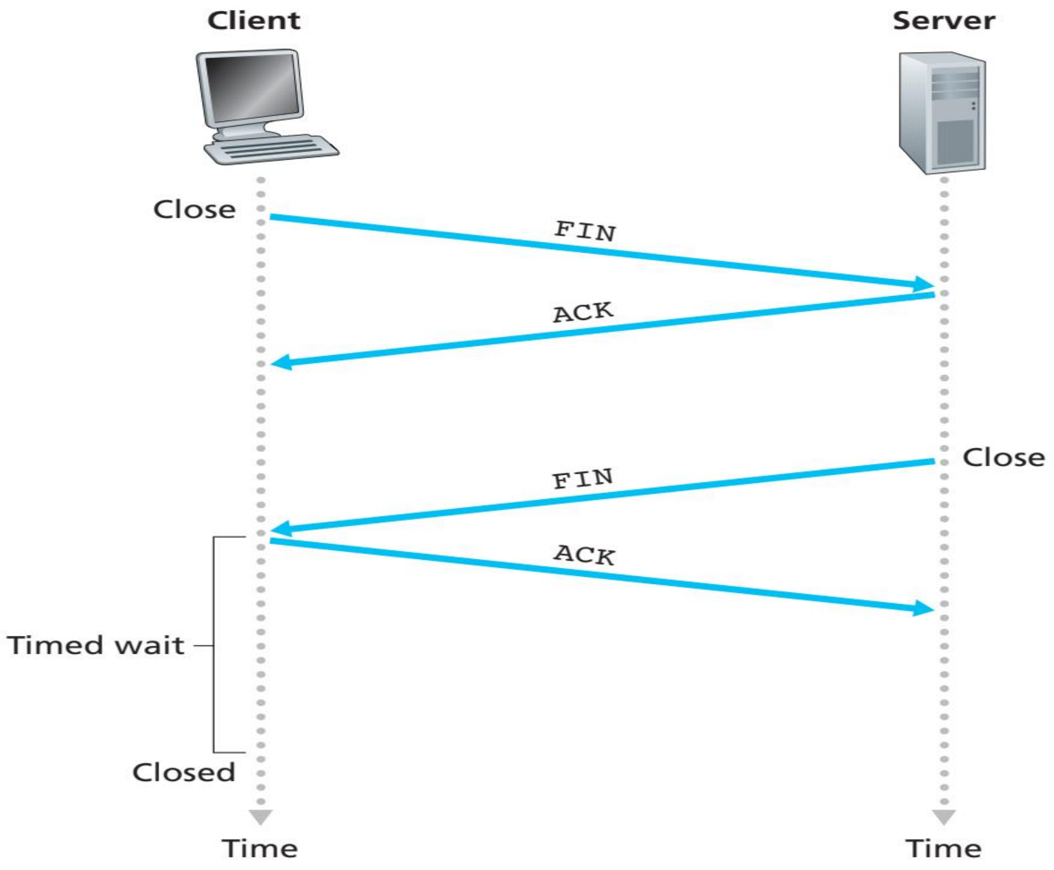


Connection Establishment: TCP Three-way Handshake Protocol

- Client-side TCP: allocates buffers and variables to the connection; sends another segment (may or may not carry application data) with - SYN bit: 0 (connection already established); acknowledgement number field: *server_isn + 1*, sequence number field: *client_isn + 1*

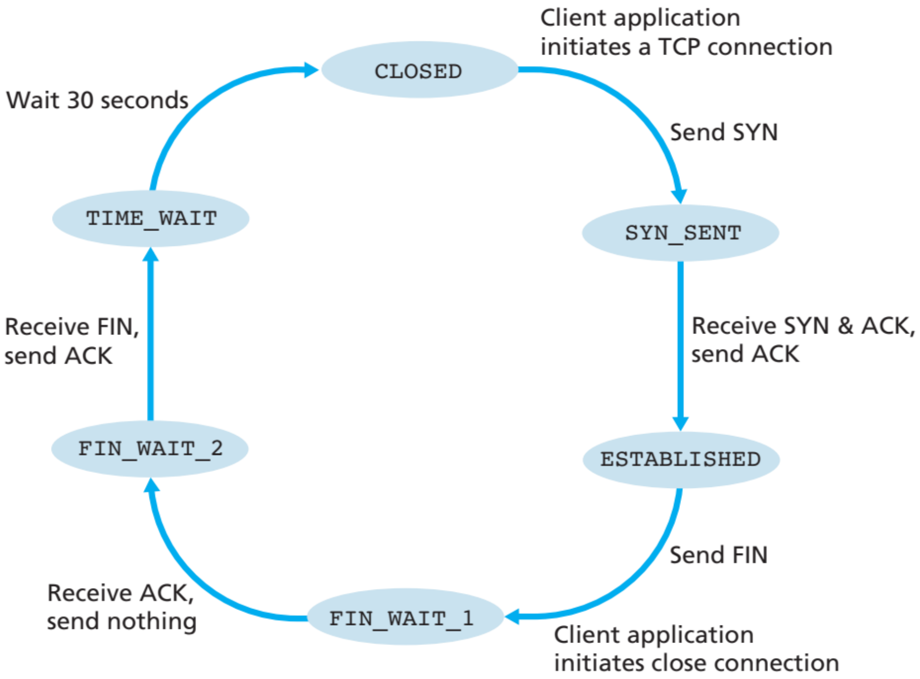
TCP: Connection Management

- Either of the two processes participating in a TCP connection can end the connection
- The resources (buffers and variables) in the hosts are deallocated.
- Steps involved in closing a TCP connection:
  - Client TCP sends a special TCP segment to the server with FIN bit set to 1.
  - The server TCP, on receiving this segment, returns an acknowledgement.
  - The server then sends its own shutdown segment, which has the FIN bit set to 1.
  - Finally, the client acknowledges the server's shutdown segment
- At this point, all the resources in the two hosts are deallocated.

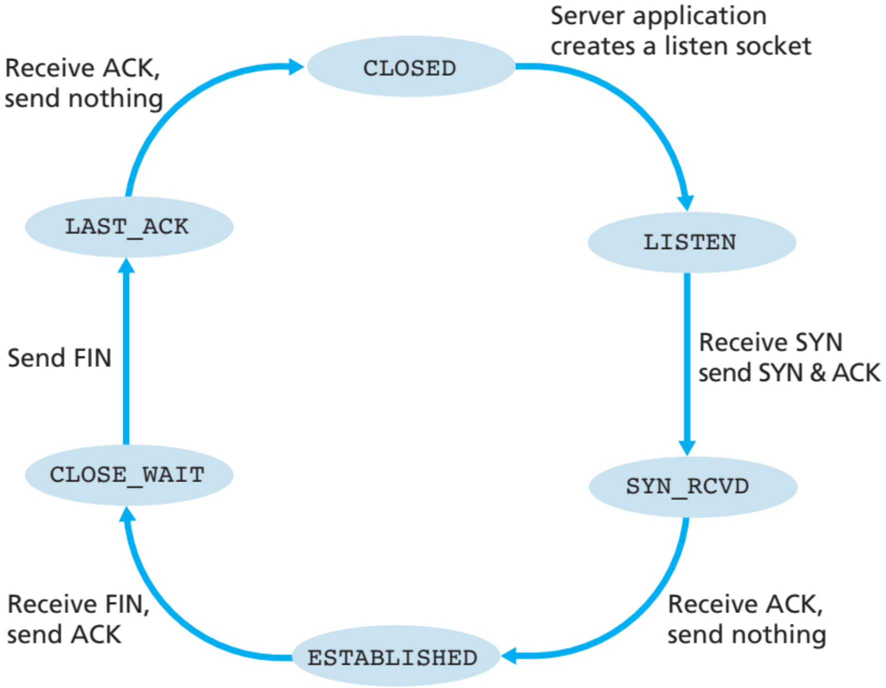


Closing a TCP Connection

TCP States



TCP States visited by a Client-TCP

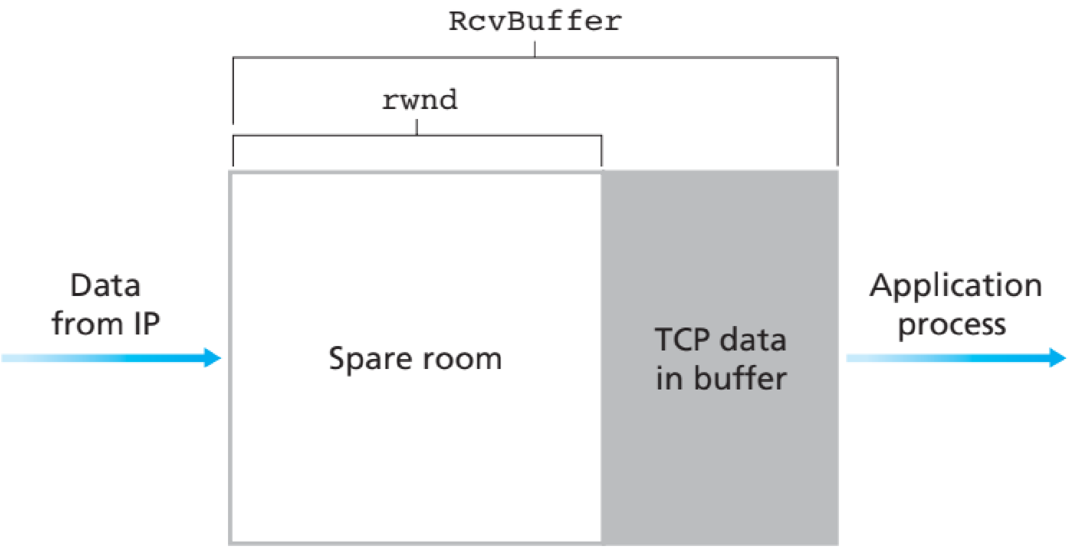


TCP States visited by a Server-TCP

- TCP states visited by a client TCP: CLOSED, SYN_SENT, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, TIME_WAIT
- TCP states visited by a server TCP: CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, CLOSE_WAIT, LAST_ACK
- If host receives a TCP segment whose port number does not match with any of the ongoing sockets in the host
  - Host sends a special TCP segment with the RST flag set to 1
  - Implications: SYN segment reached the target host, but the target host is not running and application with specific port number

TCP: Flow Control

- TCP connection uses send and receive buffers to obtain/pass the data from/to the upper-layer sender/receiver application processes
- The application at the receiver may be busy and read the data after longer duration
- A fast sender can very easily overflow the connection's receive buffer
- Flow control: eliminates the possibility of the sender overflowing the receiver's buffer
  - Matches the rate at which the sender is sending against the rate at which the receiving application is reading
- Sender maintains a variable – receive window (*rwnd*) – initially equal to the receive buffer
  - The current value of *rwnd* is inserted into every TCP segment sent by the receiver
  - Sender maintains a distinct receive window for each connection
  - It comes to know of how much free buffer space is available at the receiver

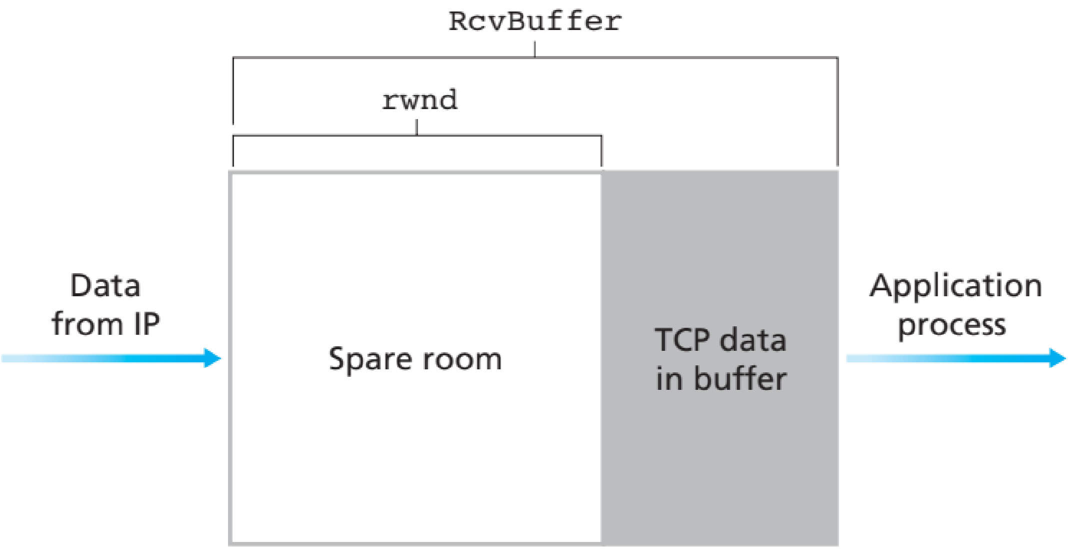


Receive Window (*rwnd*) and Receive Buffer (*RcvBuffer*)

- Flow control variables at the receiver end:
  - *RcvBuffer:* size of the buffer at the receiving host
  - *rwnd*: amount of spare space in the receive buffer; dynamic as the spare space in the buffer changes with time
  - *LastByteRead*: the number of the last byte in the data stream read from the receive buffer by the receiver application process
  - *LastByteRcvd*: the number of the last byte in the data stream that has been placed in the receive buffer at the receiver
- To prevent overflow of the allocated buffer:
  - $LastByteRcvd - LastByteRead \leq RcvBuffer$
  - $rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$

TCP: Flow Control

- Flow control variables at the sender end:
  - *LastByteSent*
  - *LastByteACKed*
- Amount of unacknowledged data that sender has sent into the connection:
  - *LastByteSent – LastByteACKed*
- TCP keeps the amount of unacknowledged data less than the value of *rwnd*
  - *LastByteSent – LastByteACKed ≤ rwnd*
  - This ensures that the sender does not overflow the receiver buffer
- Technical challenge with receive window
  - Receiver's receive buffer becomes full (*rwnd = 0*)
  - After advertising *rwnd = 0* to the sender, suppose the receiver has no data to sent (so no ACK segment)
  - TCP does not send new *rwnd* value to the sender even if the application process empties the receive buffer
    - Sender is blocked and cannot send more data



Receive Window (*rwnd*) and Receive Buffer (*RcvBuffer*)

- Solution:
  - Sender continues to send segments with one data byte when receiver's receive window is zero
  - Receiver acknowledges these segments and inserts the current value of the receive window
  - Eventually the buffer will begin to empty and the ACKs will contain a non-zero *rwnd* value