Transmission Control Protocol (TCP)

Linda Wu

(CMPT 471 • 2003-3)

Content

- TCP protocol & services
- Byte numbering
- Flow control
- Error control
- Congestion control
- Segment format
- Connection
- TCP state machine
- Silly window syndrome

Reference: chapter 13

The Need for Stream Delivery

- Unreliable packet delivery at lowest level
  - Packets can be lost, destroyed, delayed, or duplicated
- Application programs need to send large volumes of data reliably
  - Unreliable connectionless delivery system for large data requires programmer to design or modify software in order to provide reliability
  - Find general-purpose reliable stream protocol that all application programs can use

TCP Protocol

- Transmission Control Protocol
- Provide process-to-process communication
  - Port number
- Provide flow control
  - sliding window
- Provide error control
  - Acknowledgement, time-out, retransmission
- Provide connection mechanism
TCP Services

- Stream delivery service
  - Stream orientation
    - Data are delivered as a stream of bytes
    - The receiver receives the same sequence of bytes as they are sent out from the sender

TCP Services (cont.)

- Buffered transfer
  - Sender and receiver may not produce or consume data at the same speed
  - Each direction has 2 buffers: sending / receiving buffer
  - One way to implement a buffer: a circular array
- Segments
  - TCP groups a number of bytes together into a packet called segment, adds TCP header, and passes it to IP for transmission

TCP Services (cont.)

- Connection-oriented service
  - In order to transfer data between the processes on site A & B
    - A’s TCP informs B’s TCP to get approval
    - A’s TCP and B’s TCP exchange data in both directions
    - After both processes have no data left to send and the buffers are empty, the two TCPs destroy their buffers
  - * Virtual circuit connection, not physical connection

TCP Services (cont.)

- Full duplex service
  - Data can flow in both directions at the same time with no apparent interaction (sending / receiving buffer)
  - Flow in one direction can be terminated while data continues to flow in the other direction: half duplex
- Reliable service
  - Use acknowledgement mechanism to check the safe and sound arrival of data
TCP Services (cont.)

- A simple acknowledgement mechanism

Sender

Send packet 1

Receive packet 1

Receive ACK 1

Send packet 2

Receive packet 2

Receive ACK 2

* The sender waits for ACK (acknowledgement) before sending out next packet

Receiver

Send ACK 1

Send ACK 2

Byte Numbering

- Byte numbers
  - TCP numbers all bytes transmitted in a connection
  - TCP numbers the data when it receives bytes from the process and store them in the sending queue
  - The numbering starts from a random number between 0 and $2^{32} - 1$
  - Numbering is independently in each direction
  - Byte numbering is used for flow and error control

Byte Numbering (cont.)

- Sequence number
  - A sequence number is assigned to each segment that is being sent
  - Sequence number = the number of the first byte in that segment

- Acknowledgement number
  - For receiver to confirm the the bytes it has received
  - Acknowledge number = the number of the next byte expected to receive = the number of the last byte received + 1

Flow Control

- Sliding window protocol
  - Allows the sender to transmit multiple segments before an acknowledgement arrives
  - Both sender and receiver use a window for each connection
  - The window spans a portion of the sending buffer
  - The host can send the data within the window before obtaining acknowledgement from the receiver
  - The window can slide over the buffer
  - Operates at byte level, not segment level
Flow Control (cont.)

- Sender buffer
  - Empty, to be filled
  - Can be sent immediately
  - Sent, not acknowledged
  - Sent, acknowledged, recycled
  - Size = receiver window

- Receiver window
  - N: Total size of receiving buffer
  - M: number of occupied locations in the receiving buffer
  - N – M more bytes can be received
  - N – M: size of the receiver window

* Without sliding window, the sender can send all the bytes (up to 41) without regard to the condition of receiver

Notes-9 CMPT 471 • 2003-3 13

Flow Control (cont.)

- Sender window includes:
  - The bytes sent and not acknowledged
  - The bytes that can be sent
- We have flow control if sender window ≤ receiver window

Notes-9 CMPT 471 • 2003-3 14

Flow Control (cont.)

- Sender window sliding
  - Size = receiver window

Notes-9 CMPT 471 • 2003-3 15

Flow Control (cont.)

- Size = receiver window

Notes-9 CMPT 471 • 2003-3 16
Flow Control (cont.)

- **Sender window expanding**
  - The receiving process consumes data faster than it receives → The size of receiver window expands → Sender window size expands
  - Receiver acknowledged receipt of 2 more bytes (expecting byte 35) and increased receiver window to 10
  - Sender created 4 more bytes and sent 5 bytes

Flow Control (cont.)

- **Sender window shrinking**
  - The receiving process consumes data slower than it receives → The size of receiver window decreases → the receiver informs the sender to shrink its window size
  - Receiver received 5 bytes (35 ~ 39), but consumed only 1. It acknowledged bytes 35 ~ 39, and informed the sender to shrink its window size
  - Sender sent 2 more bytes and created 3 more bytes

Flow Control (cont.)

- **Sender window closing**
  - When receiver buffer is totally full
  - Receiver window size is 0
  - Receiver informs sender about the full situation
  - Sender closes its window by letting the left and right wall overlaps
  - Sender cannot send any byte until receiver announces a non-zero receiver window size

Error Control

- **Error control**: to provide reliability
  - Error control includes mechanism for:
    - Detecting corrupted, lost, out-of-order and duplicated segments
    - Correcting errors
    - Error detection
      - Checksum
        - Check for corrupted segment
      - Acknowledgement
        - Confirm receipt of segment
      - Time-out
        - Considered as corrupted or lost segment
Error Control (cont.)

- Error correction
  - Source TCP starts one timer for each segment sent
  - Each timer is checked periodically
  - If a timer expires, the corresponding segment is considered corrupted or lost, and the segment is retransmitted

Error Control (cont.)

- Checksum
  - Follows the same procedure as UDP
  - Checksum inclusion is mandatory in TCP
  - Pseudo-header
    - Source IP address
    - Destination IP address
    - ZERO
    - PROTO (6)
    - TCP length
  - Padding
    - To make segment a multiple of 16 bits
    - Pseudo-header and padding are not included in the segment length, nor transmitted with the segment

Error Control (cont.)

- Corrupted segment handling
  - The sender sends out a segment and starts a timer for it
  - The receiver finds error in the segment after checksum recalculation
  - The receiver discards the corrupted segment; no acknowledgement will be sent back
  - After the timer for the corrupted segment expires, the sender resends the segment

Error Control (cont.)

- Corrupted segment handling
  - The sender sends out a segment and starts a timer for it
  - The receiver finds error in the segment after checksum recalculation
  - The receiver discards the corrupted segment; no acknowledgement will be sent back
  - After the timer for the corrupted segment expires, the sender resends the segment
Error Control (cont.)

- Lost segment handling
  - Handled in the same way as corrupted segment

  [Diagram showing segment 1 with Seq#: 1201; 200 bytes, segment 2 with Seq#: 1401; 200 bytes, ACK#: 1401, and segment 2 retransmitted with Seq#: 1401; 200 bytes, ACK#: 1601.]

- First segment lost
  - Example: a file of 5000 bytes is transferred in 5 segments. Each segment has 1000 bytes. First byte number is 101.

  If the first segment is lost while the other 4 arrive at the receiver
  - The acknowledgements for the 4 received segments will specify 101, the next contiguous byte expected to receive
  - There is no way for the receiver to tell the sender that the last 4 segments arrived
  - The sender must retransmit either all the 5 segments, or, only the first segment: both are not efficient

Error Control (cont.)

- Duplicate segment handling
  - A duplicate segment is created by the sender when the acknowledgement does not arrive before the time-out
  - When the receiver finds a duplicate segment which has the same sequence number as another received segment, the receiver simply discards it

- Out-of-order segment handling
  - Segments may arrive at the receiver out of order
    - Segments are encapsulated in IP datagram
    - Segments are transmitted by IP
  - The receiver uses sequence number to reorder segments
  - The receiver reconstructs bytes contiguously from the beginning of the stream
Error Control (cont.)

- The receiver does not acknowledge an out-of-order segment until it receives all the preceding segments.
- If the acknowledgement for the out-of-order segment is delayed and the timer at the sender side expires, the sender will resend the segment, which will be discarded by the receiver.

Lost acknowledgement handling

- Acknowledgement is not acknowledged in TCP.
- Lost acknowledgement may not be noticed by the receiver.

TCP Timers

- 4 timers
  - Retransmission timer
  - Persistence timer
  - Keepalive timer
  - Time-waited timer

TCP Timers (cont.)

- Retransmission timer
  - To record the waiting time for an acknowledgement of a segment.
  - A retransmission timer is created when a segment is sent.
  - If the acknowledgement is received before the timer expires, the timer is destroyed.
  - If the timer expires before the acknowledgement arrives, the segment is retransmitted, and the timer is reset.
TCP Timers (cont.)

- Retransmission time calculation
  - Different retransmission time should be used for different connections
  - Retransmission time should not be fixed for one single connection
  - Adaptive retransmission time in TCP

  \[ \text{timeout} = \beta \times \text{RTT} \]

* RTT is the estimated round trip time
* \( \beta \): constant weighting factor (> 1)
  - \( \beta \) close to 1: detect packet loss quickly
  - \( \beta = 1 \): any small delay will cause an unnecessary retransmission
  - \( \beta = 2 \): recommended by the original specification

TCP Timers (cont.)

- RTT calculation
  1. Initial RTT = acknowledgement arriving time - segment sending time
  2. Whenever a new round trip time is obtained, adjust RTT

  \[ \text{RTT} = \alpha \times \text{old_RTT} + (1 - \alpha) \times \text{new_RTT} \]

* \( \alpha \): constant weighting factor, \( 0 \leq \alpha < 1 \)
  - \( \alpha \) close to 1: RTT is immune to short time change
  - \( \alpha \) close to 0: RTT responses to changes quickly
  - \( \alpha \) is usually 0.9

TCP Timers (cont.)

- Complications in RTT calculation
  - Acknowledgement ambiguity
    - In case of retransmission, there is no way to know which segment corresponds to the acknowledgement
    - Associate original segment with the acknowledgement: RTT grows without bound in cases where segments are lost
    - Associate retransmitted segment with the acknowledgement: RTT may stabilize at a value which is shorter than the correct RTT

TCP Timers (cont.)

- Karn’s algorithm
  - Avoid ambiguous acknowledgement
  - Adjust RTT only for unambiguous acknowledgement, which is for segment transmitted once
    - ** Ignoring time for retransmitted segments can also lead to failure
  - Timer backoff strategy
    - When a segment is retransmitted, increase the timeout
      \[ \text{new_timeout} = \gamma \times \text{timeout} \]
      (typically, \( \gamma = 2 \))
    - Retain the timeout value for subsequent packets until a valid sample is obtained
TCP Timers (cont.)

- **Persistence timer**
  - Possible deadlock caused by zero window-size advertisement
    - The receiving TCP announces a window size of zero
    - The sending TCP stops transmission until the receiving TCP sends acknowledgement for a non-zero window size
    - If this acknowledgement is lost, both TCPs wait for each other forever → deadlock

TCP Timers (cont.)

- To correct the deadlock
  - When the sending TCP receives a zero window-size advertisement, it starts a persistence timer
  - When the persistence timer expires, the sending TCP sends a probe segment to alert the receiving TCP that acknowledgement was lost and should be resent
  - If no response is received from the receiver, another probe is sent and the timer value is doubled and reset. This continues till the timer reaches a threshold (usually 60s)
  - After that, the sender sends a probe every 60s till the window is open

TCP Timers (cont.)

- **Keepalive timer**
  - Used in some implementations
  - To prevent a long idle connection between 2 TCPs
  - Each time the server hears from a client, it resets the keepalive timer (usually 2h)
  - If the server does not hear from the client after 2 hours, it sends a probe segment
  - If there is no response after 10 probes, terminate the connection

TCP Timers (cont.)

- **Time-waited timer**
  - Used during connection termination
  - When TCP closes a connection, it does not consider the connection really closed
  - The connection is held for a time-waited period to allow duplicate FIN segments to be discarded
  - The value of the timer is usually two times of the expected lifetime of a segment
Congestion Control

- Congestion collapse
  - Congestion occurs when the router receives packets faster than it can process
  - Some packets are dropped in congested network
  - The sender retransmits lost packets
  - More congestion and more dropping of packets
  - More retransmission and more congestion
  - The whole system collapses and no more data can be sent: congestion collapse

Congestion Control (cont.)

- Congestion avoidance
  - Congestion window
    - The sender’s window size is determined by:
      - Receiver-advertised window size
      - Network congestion
    - Allowed_window = min
      (receiver_advertisement, congestion_window)
  - Two strategies
    - Slow-start
    - Multiplicative decrease

Congestion Control (cont.)

- Slow-start
  - When start traffic on a new connection, or, increase traffic after a period of congestion:
    - Start the congestion window at the size of a single segment
    - Increase the congestion window by one segment each time an acknowledgement arrives
    - It takes $\log_2 N$ round trips before TCP can send $N$ segments: the start is not slow at all!

Congestion Control (cont.)

- Congestion avoidance phrase
  - Once the congestion window reaches a threshold (half of its original size before congestion, or half of allowed maximum window size), TCP enters a congestion avoidance phrase
    - Slows down the increment rate: increase the congestion window size by 1 only if all segments in the window have been acknowledged
    - The increment continues as long as ACKs arrive before their time-out, or, the congestion window size reaches the receiver window size
Congestion Control (cont.)

Multiplicative congestion avoidance
- In the steady state on a non-congested connection:
  - The congestion window is the same size as the receiver window
- Upon loss of a segment:
  - Set the congestion window threshold to half of the last congestion window size
  - Start the congestion window size from one segment
  - For the segments that remain in the allowed window, double the retransmission timer

Congestion Control (cont.)

Router policies for congestion
- Tail-drop policy
  - If the queue is full when a datagram arrives, discard the datagram
  - Once the queue is full, the router discards the "tail"
  - May cause global synchronization
    - The router may discard one segment from N connections rather than N segments from one connection
    - Simultaneous loss causes all N instances of TCP to enter slow start

Random early discard (RED) policy
- $T_{\text{min}}, T_{\text{max}}$: 2 thresholds to mark positions in the queue
  - If the queue contains fewer than $T_{\text{min}}$ datagrams, add new datagram to the queue
  - If the queue contains more than $T_{\text{max}}$ datagrams, discard new datagram
  - If the queue contains between $T_{\text{min}}$ and $T_{\text{max}}$ datagrams, randomly discard the datagram according to a probability $p$
  - The router slowly and randomly drops datagram as congestion increases
Congestion Control (cont.)

- Choice of $T_{\text{min}}$, $T_{\text{max}}$, $p$
  - $T_{\text{min}}$: must be large enough to ensure that the output link has high utilization
  - $T_{\text{max}}$: must be larger than $T_{\text{min}}$ by more than the typical increase in the queue size during one TCP round trip time
- Probability $p$
  - The value of $p$ is computed for each datagram
  - The value depends on the relationship between the current queue size and the thresholds
  - The value should not lead to the discarding of datagrams from a burst

Segment Format

- Segment: TCP transfer unit
- Segments are exchanged to:
  - Establish connection
  - Transfer data
  - Send acknowledgement
  - Advertise window size
  - Close connection
- Acknowledgement transmission
  - The ACK from A to B may travel in the same segment as data from A to B: piggybacking

Segment Format (cont.)

- Segment format

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>10</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td>Sequence Number</td>
<td>Acknowledgement Number</td>
<td>HLen</td>
</tr>
</tbody>
</table>

Segment Format (cont.)

- Header (20 ~ 60 bytes)
  - Source / destination port
    - TCP port numbers that identify the application programs at the ends of the connection
  - Sequence number
    - The number of the first byte in this segment
    - Refer to the stream in the same direction as the segment
  - Acknowledgement number
    - The number of the byte the receiver expects to receive next
    - Refer to the stream in the opposite direction from the segment
Segment Format (cont.)

- HLEN
  - The header length measured in 32-bit
  - Possible value: 5 ~ 15
- Code Bits
  - Define 6 different control bits
  - Used to determine the purpose and contents of the segment
  - One or more bits can be set at a time

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning if bit set to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>URG</td>
<td>urgent pointer field is valid</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement field is valid</td>
</tr>
<tr>
<td>PSH</td>
<td>This segment requests a push</td>
</tr>
<tr>
<td>RST</td>
<td>Reset the connection</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronize the sequence number</td>
</tr>
<tr>
<td>FIN</td>
<td>Sender has reached end of its byte stream (terminate the connection)</td>
</tr>
</tbody>
</table>

Segment Format (cont.)

- URG bit
  - Used when the sender wants to send data out of band, without waiting for the receiver to consume data already in the stream
  - The sender specifies data as urgent
  - The receiver will be notified of the urgent data's arrival as quickly as possible, regardless of its position in the stream
  - When the urgent data is found, the receiving TCP should notify whatever application program is associated with the connection to go into urgent mode
  - After all urgent data is consumed, TCP tells the application program to return to normal operation

- Window
  - Used by the TCP software to specify how much data it is willing to accept
  - This field accompanies the segments that carry data and that carry only ACK: piggybacking

- Checksum

- Urgent pointer
  - Valid only if URG bit is set
  - Specify the position in the segment where urgent data ends

- Options
  - Up to 40 bytes of optional information
Segment Format (cont.)

- Options
  - End of option
  - No operation
  - Maximum segment size
  - Window scale factor
  - Timestamp

Segment Format (cont.)

- End of option
  - 1 byte, code 0 (00000000)
  - Used for padding at the end of the option field
  - Can used only as the last option
  - Only one "end of option" can be used
  - If more than 1 byte is needed to align the option fields, use some no-operation options before an end of option

<table>
<thead>
<tr>
<th>Options</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-Op</td>
<td>(Used for padding)</td>
</tr>
</tbody>
</table>

Segment Format (cont.)

- No operation
  - 1 byte, code 1 (00000001)
  - Used as a filler between options

<table>
<thead>
<tr>
<th>No-op</th>
<th>3-byte option</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>to align the beginning of an option</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3-byte option</th>
<th>No-op</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-byte option</td>
<td></td>
</tr>
<tr>
<td></td>
<td>to align the next option</td>
</tr>
</tbody>
</table>
Segment Format (cont.)

- Maximum segment size (MSS)
  - 4 bytes, code: 2 (00000010)
  - Define the maximum segment size that can be accepted on a connection
  - Only count bytes of data in the segment; not count TCP or IP header
  - Determined only at the time a connection is established
  - The value can be 0 ~ 65,535 bytes
  - Default value: 536 bytes
  - MSS announcement is sent from the data receiver to the sender

<table>
<thead>
<tr>
<th>Code</th>
<th>Length</th>
<th>Maximum segment size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>1 byte</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

Segment Format (cont.)

- MSS choosing
  - Two end points are on the same physical network
    - IP datagram carrying segment with MSS size should match the network MTU
  - Endpoints are not on the same physical network:
    - Discover the minimum MTU along the path between them, or,
    - Choose the default value 536
  - It is hard to find a good MSS in a general internet environment

Segment Format (cont.)

- Window scale factor (3 bytes)
  - To specify a larger size for the sliding window than the “window” field in the header
  - new window size = window size defined in the header * 2\(^{\text{window scale factor}}\)
  - Maximum factor in TCP/IP: 16
  - Determined during the connection establishment

<table>
<thead>
<tr>
<th>Code</th>
<th>Length</th>
<th>Scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>1 byte</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

Segment Format (cont.)

- Timestamp (10 bytes)
  - Timestamp value is filled by the sender when the segment leaves
  - The receiver receives the segment and stores the timestamp value
  - When the receiver sends back an ACK, it enters the previously stored value in the echo reply field
  - When the sender receives the ACK, it checks the current time versus the echo reply value. The difference is the round trip time

<table>
<thead>
<tr>
<th>Code</th>
<th>Length</th>
<th>Timestamp value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Length</th>
<th>Timestamp echo reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Connection

- Connection abstraction
  - TCP uses the connection, not the protocol port, as its fundamental abstraction
  - The connection is identified by a pair of endpoints
  - Endpoint is a pair of integer (host, port)
    - Host: IP address of a host
    - Port: a TCP port on that host; use both static and dynamic port binding
      - e.g., 2 endpoints (18.26.0.36, 1069) and (128.10.2.3, 25) defines a connection
  - Multiple connections can share an endpoint

Connection (cont.)

- Connection establishment
  - Both ends of the connection must agree that the connection is desired
    1. Server makes a request for a passive open: to tell its TCP that it is ready to accept a connection
    2. Client that wishes to connect to the server makes a request for an active open: to tell its TCP that it needs to be connected to the server

Connection (cont.)

- TCP starts three-way handshake
  1. The client sends the first segment: SYN
     - SYN bit
     - Source and destination port numbers
     - Client initial sequence number (ISN)
     - MSS & window scale factor (if needed)
     - NO ACK number OR window size
     * The first segment defines the wish of the client to make a connection with certain parameters
  2. The server sends the second segment: SYN + ACK
     - SYN & ACK bits
     - ACK number (= client ISN + 1)
     - Window size
     - Server initial sequence number (ISN)
     - MSS & window scale factor (if needed)
     * The second segment confirms the receipt of the first segment and continues the handshake
Connection (cont.)

3. The client sends the third segment: ACK
   - ACK bit
   - ACK number (= server ISN + 1)
   - Window size

- The third segment informs the server that both sides agree that a connection has been established
- TCP ignores additional requests for connection after a connection has been established
- Data can be sent with initial sequence numbers in the handshake segments

Simultaneous open

- Two applications both perform an active open to each other at the same time
  - Possible, although improbable
  - Each end must transmit a SYN segment
  - Each end has a local port number that is well known to the other end
  - TCP should handle simultaneous open:
    - One single connection is established between them
    - Data flows in both directions equally well; there is no master or slave
Connection (cont.)

- Connection closing
  - Both sides can close the connection
  - The connection may be closed in one direction, but left open in the other direction
  - TCP refuses to accept data from the closed direction, but data can still be sent in the other direction
  - ACK can still flow in the closed direction

Connection (cont.)

- Client makes a request for an active close
  - To tell its TCP it has finished sending data and wishes to terminate the connection
  - Client TCP closes connection in the client-server direction
  - When the server finishes sending data, it makes a request for a passive close
  - To tell its TCP to close the connection in the server-client direction
  - When both directions are closed, TCP at each endpoint deletes its record of the connection

Connection (cont.)

- Four-way handshake
  - The client sends FIN segment (FIN bit set)
  - The server sends ACK segment to confirm the receipt of the FIN segment from the client
  - The server continues sending data in the server-client direction. When it has no more data to send, it sends FIN segment
  - The client sends ACK segment to confirm the receipt of the FIN segment from the server

4-way handshake
Connection (cont.)

- Simultaneous close
  - Usually one side (often, but not always, the client) performs the active close
  - It is also possible for both sides to perform an active close
  - TCP protocol allows for simultaneous close

Connection (cont.)

- Connection reset
  * Used for abnormal disconnections
  - One side initiates termination by sending a RST segment (RST bit is set)
  - The other side responds to a reset segment immediately by aborting the connection; TCP informs the application that a reset occurred
  - A reset is an instantaneous abort
    - Transfer in both directions ceases immediately
    - Buffers are released

TCP State Machine

- TCP software is implemented as a finite state machine
  - The machine goes through a limited number of states
  - The machine is in one of the states at any time
  - An event can take the machine to a new state, or make it perform some actions

TCP State Machine (cont.)

- TCP States (Fig 13.15 in the textbook)

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>There is no connection</td>
</tr>
<tr>
<td>LISTEN</td>
<td>The server is waiting for calls from the client</td>
</tr>
<tr>
<td>SYN-SENT</td>
<td>A connection request is sent; waiting for ACK</td>
</tr>
<tr>
<td>SYN-RCVD</td>
<td>A connection request is received</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>Connection is established</td>
</tr>
<tr>
<td>FIN-WAIT-1</td>
<td>One side has requested to close the connection</td>
</tr>
<tr>
<td>FIN-WAIT-2</td>
<td>The other side has accepted the closing of the connection</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Both sides have decided to close simultaneously</td>
</tr>
<tr>
<td>TIME-WAIT</td>
<td>Waiting for retransmitted segments to die</td>
</tr>
<tr>
<td>CLOSE-WAIT</td>
<td>The server is waiting for the application to close</td>
</tr>
<tr>
<td>LAST-ACK</td>
<td>The server is waiting for the last ACK</td>
</tr>
</tbody>
</table>
TCP State Machine (cont.)

- Client state transition
  (in normal operation)
  - \( \text{TIME-WAIT} \) Time-out
  - \( \text{CLOSED} \) Active open / SYN
  - \( \text{SYN-SENT} \)
  - \( \text{FIN-WAIT-2} \)
    - ACK Close / FIN
  - \( \text{FIN-WAIT-1} \)
    - ACK Close / FIN
  - \( \text{ESTABLISHED} \)

TCP State Machine (cont.)

- Server state transition
  (in normal operation)
  - \( \text{LISTEN} \) Passive open
  - \( \text{CLOSED} \)
  - \( \text{LAST ACK} \)
  - \( \text{SYN-RCVD} \)
  - \( \text{ACK} \)
  - \( \text{FIN / ACK} \)
    - \( \text{ESTABLISHED} \)

Silly Window Syndrome

- Silly window syndrome (SWS)
  - Sending and receiving processes operate at different speeds
  - Small segments are produced and transferred
  - Waste network bandwidth
    - The ratio of header to data is large
  - Introduce computation overhead
    - Checksum calculation
    - Data extraction
    - Sequence number checking
    - Packet routing
    - ......

Silly Window Syndrome (cont.)

- Sender-side silly window
  - How silly window occurs at sender side
    - Data are created slowly by the sending TCP, e.g. 1 byte at a time
    - 1 byte is written into the sending buffer at a time
    - Small segment with 1 byte of data is created
    - A lot of 41-byte segments travel through an internet
  - Solution
    - Prevent the sending TCP from sending data byte by byte
    - Force sending TCP to wait as it collects data to send in a larger block
Silly Window Syndrome (cont.)

- Nagle algorithm
  1. Sending TCP sends the first piece of data it receives from the sending process even if it is only 1 byte
  2. Sending TCP accumulates data in the buffer and waits until:
     - receiving TCP sends an acknowledgement, or,
     - enough data has accumulated to fill a maximum-size segment; then, sending TCP sends the segment
  3. Step 2 is repeated for the rest of the transmission

Silly Window Syndrome (cont.)

- Elegance of Nagle algorithm
  - Adaptive – the delay (sender’s waiting time before sending next segment) depends on the speed of sending process and the performance of network
  - Simple – little computation overhead

Silly Window Syndrome (cont.)

- Receiver-side silly window
  - How silly window occurs at receiver side
    - Receiving process consumes data slowly, e.g. 1 byte at a time
    - Receiving buffer is full
    - Receiving TCP announces receiver window size as 1 byte after 1 byte is consumed
    - Sending TCP sends segment with only 1 byte
    - The procedure continues: 1 byte is consumed and 1 byte is sent again

Silly Window Syndrome (cont.)

- Clark’s solution
  The receiving TCP:
  - sends an acknowledgement as soon as the data arrives
  - announces receiver window size of zero until: there is enough space to accommodate a maximum-size segment, or
  - half of the buffer is empty
Silly Window Syndrome (cont.)

- Delayed acknowledgement
  - A segment is not acknowledged immediately after it arrives at receiver
  - Receiving TCP sends acknowledgement when there is enough space in the buffer

* Advantage: reduce traffic
* Disadvantage: delayed acknowledgement may force retransmission of unacknowledged segment
* To avoid potential problem: ACK cannot be delayed for more than 500 ms; (recommended) receiver acknowledges at least every other segment

TCP Operation

- Encapsulation & decapsulation
  - Application
  - Message
  - TCP header
  - TCP data
  - IP header
  - IP data
  - Frame header
  - Frame data

TCP Operation (cont.)

- Buffering
  - Sending buffer and receiving buffer are created for each direction

- Multiplexing & demultiplexing
  - One TCP
  - Possibly several applications that want to use TCP services

- Urgent data
  - URG bit is set to indicate urgent data

TCP Operation (cont.)

- Pushing data
  - TCP accumulates enough bytes in the buffer to make segments reasonably long
  - TCP provides a push operation to accommodate interactive users
    - Set PSH bit in the segment header
    - Force delivery of bytes in the stream without waiting for the buffer to fill

* TCP operates efficiently over the global Internet