CC231

Introduction to Networks

Dr. Ayman A. Abdel-Hamid

CCIT - AASTMT

Transmission Control Protocol (TCP)
Outline

• Transmission Control Protocol
Transport Layer 1/2

Application layer

Gives services to

Transport layer

Packetizing

Addressing

Connection control

Reliability

Receives services from

Network layer

Quality of service

Congestion control

TCP © Dr. Ayman Abdel-Hamid, CC231
Transport Layer

Node-to-node: Data link layer
Host-to-host: Network layer
Process-to-process: Transport layer

Process-to-process delivery
Transport Layer Addressing

Addresses

• Data link layer $\rightarrow$ MAC address
• Network layer $\rightarrow$ IP address
• Transport layer $\rightarrow$ *Port number* (choose among multiple processes running on destination host)
Port Numbers

- Port numbers are 16-bit integers (0 → 65,535)
  - Servers use *well known ports*, 0-1023 are privileged
  - Clients use *ephemeral* (short-lived) ports
- *Internet Assigned Numbers Authority* (IANA) maintains a list of port number assignment
  - *Well-known ports* (0-1023) → controlled and assigned by IANA
  - *Registered ports* (1024-49151) → IANA registers and lists use of ports as a convenience (49151 is ¾ of 65536)
  - *Dynamic ports* (49152-65535) → ephemeral ports
  - For well-known port numbers, see /etc/services on a UNIX or Linux machine
Socket Addressing

• Process-to-process delivery needs *two* identifiers
  ➢ IP address and Port number
  ➢ Combination of IP address and port number is called a socket address (a socket is a communication endpoint)
  ➢ Client socket address uniquely identifies client process
  ➢ Server socket address uniquely identifies server process

• Transport-layer protocol needs a *pair* of socket addresses
  ➢ Client socket address
  ➢ Server socket address
  ➢ For example, socket pair for a TCP connection is a 4-tuple
    ✓ Local IP address, local port, and
    ✓ foreign IP address, foreign port
Multiplexing and Demultiplexing

**Multiplexing**

Sender side may have several processes that need to send packets (albeit only 1 transport-layer protocol)

**Demultiplexing**

At receiver side, after error checking and header dropping, transport-layer delivers each message to appropriate process
Transmission Control Protocol

• TCP must perform typical transport layer functions:
  ➢ Segmentation → breaks message into packets
  ➢ End-to-end error control → since IP is an unreliable Service
  ➢ End-to-end flow control → to avoid buffer overflow
  ➢ Multiplexing and demultiplexing sessions

• TCP is [originally described in RFC 793, 1981]
  ➢ Reliable
  ➢ Connection-oriented → virtual circuit
  ➢ Stream-oriented → users exchange streams of data
  ➢ Full duplex → concurrent transfers can take place in both directions
  ➢ Buffered → TCP accepts data and transmits when appropriate (can be overridden with “push”)
Transmission Control Protocol 2/13

• Reliable
  ➢ requires ACK and performs retransmission
  ➢ If ACK not received, retransmit and wait a longer time for ACK. After a number of retransmissions, will give up
  ➢ How long to wait for ACK? (dynamically compute RTT for estimating how long to wait for ACKs, might be ms for LANs or seconds for WANs)

  \[
  \text{RTT} = \alpha \times \text{old RTT} + (1 - \alpha) \times \text{new RTT}
  \]
  where \( \alpha \) usually 90%

  ➢ Most common, Retransmission time = 2 \times \text{RTT}
  ➢ Acknowledgments can be “piggy-backed” on reverse direction data packets or sent as separate packets
Transmission Control Protocol 3/13

• Sequence Numbers

➤ Associated with every byte that it sends
➤ To detect packet loss, reordering and duplicate removal
➤ Two fields are used sequence number and acknowledgment number. Both refer to byte number and not segment number
➤ Sequence number for each segment is the number of the first byte carried in that segment
➤ The ACK number denotes the number of the next byte that this party expects to receive (cumulative)

✔ If an ACK number is 5643 → received all bytes from beginning up to 5642
✔ This acknowledges all previous bytes as received error-free
Transmission Control Protocol

- Sending and Receiving buffers
  - Senders and receivers may not produce and consume data at the same speed
  - 2 buffers for each direction (sending and receiving buffer)
Transmission Control Protocol 5/13

- TCP uses a sliding window mechanism for flow control
- Sender maintains 3 pointers for each connection
  - Pointer to bytes sent and acknowledged
  - Pointer to bytes sent, but not yet acknowledged
    - Sender window includes bytes sent but not acknowledged
  - Pointer to bytes that cannot yet be sent

![Diagram showing TCP sliding window mechanism with byte numbers 3 to 17, and pointers indicating sent, acknowledged, and window size=8.]
Transmission Control Protocol 6/13

• Flow Control

➢ Tell peer exactly how many bytes it is willing to accept (advertised window $\rightarrow$ sender can not overflow receiver buffer)
   ✓ Sender window includes bytes sent but not acknowledged
   ✓ Receiver window (number of empty locations in receiver buffer)
   ✓ Receiver advertises window size in ACKs

➢ Sender window $\leq$ receiver window (flow control)
   ✓ Sliding sender window (without a change in receiver’s advertised window)
   ✓ Expanding sender window (receiving process consumes data faster than it receives $\rightarrow$ receiver window size increases)
   ✓ Shrinking sender window (receiving process consumes data more slowly than it receives $\rightarrow$ receiver window size reduces)
   ✓ Closing sender window (receiver advertises a window of zero)
Transmission Control Protocol

• Error Control

- Mechanisms for detecting corrupted segments, lost segments, out-of-order segments, and duplicated segments
- Tools: checksum (corruption), ACK, and time-out (one time-out counter per segment)
  - *Lost segment or corrupted segment* are the same situation: segment will be retransmitted after time-out (no NACK in TCP)
  - *Duplicate segment* (destination discards)
  - *Out-of-order segment* (destination does not acknowledge, until it receives all segments that precede it)
  - *Lost ACK* (loss of an ACK is irrelevant, since ACK mechanism is cumulative)
Transmission Control Protocol 8/13

• Congestion Control

➢ TCP assumes the cause of a lost segment is due to congestion in the network

➢ If the cause of the lost segment is congestion, retransmission of the segment does not remove the problem, it actually aggravates it

➢ The network needs to tell the sender to slow down (affects the sender window size in TCP)

➢ Actual window size = Min (receiver window size, congestion window size)

✓ The congestion window is flow control imposed by the sender
✓ The advertised window is flow control imposed by the receiver
Congestion Control 9/13

- **Slow start**
  
  - At start of connection, set congestion window size to maximum segment size
  
  - For each segment ACKed, increase congestion window size by 1 maximum segment size *until it reaches a threshold of one-half allowable window size*
  
  - Exponential increase in size
    
    ✓ Send 1 segment, receive 1 ACK, increase size to 2 segments
    
    ✓ Send 2 segments, receive 2 ACKs, increase size to 4 segments
    
    ✓ Send 4 segments, receives 4 ACKs, increase size to 8 segments
Congestion Control

• Additive Increase (Congestion Avoidance phase)
  ➢ After size reaches threshold, size is increased one segment for each
    ACK, even if ACK is for several segments (this continues as long as
    ACKs arrive before time-outs, or congestion window reaches the
    receiver window value)

• Multiplicative Decrease
  ➢ If a time-out occurs, threshold set to one-half of last congestion
    window size, and congestion window size starts from 1 (return to
    slow start)
  ➢ Threshold reduced to one-half current congestion window size
    every time a time-out occurs (exponential reduction)
  ➢ Exponential growth stops when the threshold is hit
  ➢ Afterwards, successful transmissions grow congestion window
    linearly

• Such congestion control often referred to as TCP Tahoe
Transmission Control Protocol

- Congestion Control

![Graph showing congestion window size vs. transmission number](image_url)
TCP Variants 12/13

• TCP Tahoe (first implemented in 4.3 BSD, 1988)

  ➢ Slow start + Congestion avoidance and fast retransmit

  ➢ Fast Retransmit

    ✓ Triggers the transmission of a dropped segment if three *dup ACKs* for a segment are received before the occurrence of the segment's timeout

    ✓ TCP required to immediately generate a dup ACK if an out-of-order segment is received (on receiving a dup ACK, cant tell if the reason is a reorder of segments or a lost segment, hence the wait to receive a number of dup ACKs)

    ✓ Fast Retransmit was incorrectly followed by slow start
• Full-Duplex

➤ send and receive data in both directions.
➤ Keep sequence numbers and window sizes for each direction of data flow
TCP Connection Establishment

SYN: Synchronize
ACK: Acknowledge
TCP Options

Each SYN can contain TCP options

• MSS Option
  ➢ maximum segment → the maximum amount of data it is willing to accept in each TCP segment
  ➢ Sending TCP uses receiver’s MSS as its MSS

• Window Scale Option
  ➢ maximum window is 65,535 bytes (corresponding field in TCP header occupies 16 bits)
  ➢ it can be scaled (left-shifted) by 0-14 bits providing a maximum of $65,535 \times 2^{14}$ bytes (one gigabyte)
  ➢ Option needed for high-speed connections or long delay paths
  ➢ In this case, the other side must send the option with its SYN
TCP MSS and output

- TCP MSS is \(= (\text{interface MTU} - \text{fixed sizes of IP and TCP headers (20 bytes)})\)
  
  \(\Rightarrow\) MSS on an Ethernet (IPv4) = 1460 bytes (1500 why?) - 40

- Successful return from write implies you can reuse application buffer
TCP Connection Termination

- FIN: Finish
- Step 1 can be sent with data
- Steps 2 and 3 can be combined into 1 segment
Typical TCP states visited by a TCP client
Typical TCP states visited by a TCP server
## State Transition Diagram

<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 acknowledgment.</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>The application has requested the closing of 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>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 acknowledgment.</td>
</tr>
</tbody>
</table>

Can use *netstat* command to see some TCP states
Packet Exchange

**Piggybacking feature**

Send 1-segment request and receive 1-segment reply
TIME_WAIT State

• *The end that performs the active close goes through this state*

• Duration spent in this state is twice the *maximum segment life* (2 MSL)
  
  ➢ *MSL*: maximum amount of time any given IP can live in the network

• Every TCP implementation must choose a value for MSL
  
  ➢ Recommended value is 2 minutes (traditionally used 30 seconds)

• TIME_WAIT state motives
  
  ➢ allow old duplicate segments to expire in the network (relate to *connection incarnation*)
  
  ✓ TCP will not initiate a new incarnation of a connection that is in TIME_WAIT state
  
  ➢ Implement TCP’s full-duplex connection termination reliably
  
  ✓ The end that performs the active close might have to resend the final ACK
# TCP Segment Format

**TCP Header**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit source port number</td>
<td>16 bits</td>
<td>Source port number</td>
</tr>
<tr>
<td>16-bit destination port number</td>
<td>16 bits</td>
<td>Destination port number</td>
</tr>
<tr>
<td>32-bit sequence number</td>
<td>32 bits</td>
<td>Sequence number</td>
</tr>
<tr>
<td>32-bit acknowledgment number</td>
<td>32 bits</td>
<td>Acknowledgment number</td>
</tr>
<tr>
<td>4-bit header length</td>
<td>4 bits</td>
<td>Header length</td>
</tr>
<tr>
<td>reserved (6 bits)</td>
<td>6 bits</td>
<td>Reserved</td>
</tr>
<tr>
<td>URG</td>
<td>1 bit</td>
<td>Urgent</td>
</tr>
<tr>
<td>ACK</td>
<td>1 bit</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>PSH</td>
<td>1 bit</td>
<td>Push</td>
</tr>
<tr>
<td>RST</td>
<td>1 bit</td>
<td>Reset</td>
</tr>
<tr>
<td>SYN</td>
<td>1 bit</td>
<td>Synchronize</td>
</tr>
<tr>
<td>FIN</td>
<td>1 bit</td>
<td>Finish</td>
</tr>
<tr>
<td>16-bit window size</td>
<td>16 bits</td>
<td>Window size</td>
</tr>
<tr>
<td>16-bit TCP checksum</td>
<td>16 bits</td>
<td>TCP checksum</td>
</tr>
<tr>
<td>16-bit urgent pointer</td>
<td>16 bits</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>options (if any)</td>
<td>variable</td>
<td>Options</td>
</tr>
<tr>
<td>data (if any)</td>
<td>variable</td>
<td>Data</td>
</tr>
</tbody>
</table>

20 bytes
TCP Header Fields

• **Source Port and Destination Port**
  - Identify processes at ends of the connection

• **Control bits**
  - URG urgent (urgent data present)
  - ACK acknowledgment
  - PSH push request
    - Inform receiver TCP to send data to application ASAP
  - RST reset the connection
  - SYN synchronize sequence numbers
  - FIN sender at end of byte stream
TCP Header Fields 2/2

• **Sequence Number**: position of the data in the sender’s byte stream

• **Acknowledgment Number**: position of the byte that the source expects to receive next (valid if ACK bit set)

• **Header Length**: header size in 32-bit units. Value ranges from [5-15]

• **Window**: advertised window size in bytes

• **Urgent**
  - defines end of urgent data (or “out-of-band”) data and start of normal data
  - Added to sequence number (valid only if URG bit is set)

• **Checksum**: 16-bit CRC (Cyclic Redundancy Check) over header and data

• **Options**: up to 40 bytes of options