TCP Protocol

- **TSAPs**
  - Use <host, port> combination
  - Well known ports provide services
    - first 256 ports (port # 0-255)
    - SMTP 25, Telnet 23, Ftp 21, HTTP 80
- Provides a **byte stream**
  - not a message stream (i.e., no message boundary)
  - a message (single call to send) may be split, merged, etc.
- **Urgent Data field**
  - provides cut through delivery *within* a transport connection
  - used to send control interrupts or other high priority info
- Every byte on a TCP connection has its own 32-bit sequence number
TCP Segment Format

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sequence Number**: (for the first byte contained in this segment)
- **Acknowledgment Number**: (for the next byte expected)
- **Window Size**: (for dynamic buffer allocation)
- **Checksum**: (check both the header and data)
- **Urgent Pointer**: (byte offset at which to find urgent data)
- **0 Or More Options**: Such as specifying the maximum segment size a host is willing to accept, specifying selective repeat instead of go back N, allowing 32-bit window size instead of 16-bit
- **Data (Optional)**: 32 bits

- **ACK bit**: set means an ACK is in this segment/piggybacked => use the ACK # field
  - What happens if ACK bit is on but Window Size = 0?
- **URG bit**: set means there are urgent data in this segment => use the urgent pointer field
- **PSH bit**: set means to push the receiver to deliver data to the application immediately
- **RST bit**: set means to reset a connection, say, due to a host crash
- **SYN bit**: to set up a connection
  - SYN set/ACK clear: it is a Connection Request TPDU
  - SYN set/ACK set: it is a Connection Accepted TPDU
- **FIN bit**: set means that the sender has no more data to send although it can still receive data
TCP Connection Management

- **Three-way Handshake to establish a connection**
  - host 1 SYN followed by host 2 SYN_ACK followed by host 1 ACK

- **clock-based scheme for assigning initial sequence numbers**
  - the clock ticks once every 4 $\mu$sec
  - use a simple scheme to make sure all old packets of a connection have died off
    - a crashed host waits for T (120 seconds) before a reboot
    - when releasing a connection, the TCP waits for T too

- **Connection Closure**
  - host 1 sends a FIN, host 2 sends a FIN_ACK, and vice versa
    - normally four TCP segments are needed to release a connection
    - however, if the first FIN_ACK and the 2nd FIN can be combined, then only 3 TCP segments are needed
      - A FIN times out after 2T (240 seconds): the connection is released
      - Keep-alive timers are used to timeout half-open dead connections
TCP Connection Establishment

- A SYN segment consumes 1 byte of sequence space (due to the clock-driven scheme) so it can be acknowledged unambiguously.

Just one connection is established, not two because connections are identified by their end points. In this case, both setups result in the same connection identified by \((x,y)\).

Fig. 6-26. (a) TCP connection establishment in the normal case. (b) Call collision.
TCP Connection/Disconnection Steps

- Can be represented by a finite state machine with 11 states

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>No connection is active or pending</td>
</tr>
<tr>
<td>LISTEN</td>
<td>The server is waiting for an incoming call</td>
</tr>
<tr>
<td>SYN RCVD</td>
<td>A connection request has arrived; wait for ACK</td>
</tr>
<tr>
<td>SYN SENT</td>
<td>The application has started to open a connection</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>The normal data transfer state</td>
</tr>
<tr>
<td>FIN WAIT 1</td>
<td>The application has said it is finished</td>
</tr>
<tr>
<td>FIN WAIT 2</td>
<td>The other side has agreed to release</td>
</tr>
<tr>
<td>TIMED WAIT</td>
<td>Wait for all packets to die off</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Both sides have tried to close simultaneously</td>
</tr>
<tr>
<td>CLOSE WAIT</td>
<td>The other side has initiated a release</td>
</tr>
<tr>
<td>LAST ACK</td>
<td>Wait for all packets to die off</td>
</tr>
</tbody>
</table>

(client side)  (server side)

Fig. 6-27. The states used in the TCP connection management finite state machine.
TCP Connection/Disconnection State Machine

A state transition is marked with an event/action pair

Events:
- A user-initiated call
  - Connect
  - Listen
  - Send
  - Close
- A segment arrival
  - SYN
  - FIN
  - ACK
  - RST
- A timeout after T

Actions (sending a control segment):
- SYN (connect)
- FIN (disconnect)
- RST (reject)
- -- (nothing)
TCP Transmission Policy

- **Use Variable-sized sliding window protocols**
  - ACK # field indicates the next byte # (seq. #) expected
  - Window size indicates the current window size allowed

- **Receiver can send a window of 0**
  - to indicate that it wants to pause for a while
  - urgent data need not follow this rule - the URG bit is on

- **Window size of 16 bits is too small**
  - only 64K bytes can be in transit
  - this is only a small fraction of the in-flight bytes when
    - bandwidth is high
    - delay is high
  - solution: window shift option - use the OPTION field
    - bit shift window up to 16 bits
    - permits up to $2^{32}$ byte windows
TCP Window Management

This is essentially a sliding window protocol with selective repeat, with the receiver’s and sender’s window size being dynamically adjusted depending on the buffer availability of the receiving host.

If lost, the sender can timeout asking for the current window size with a 1-byte segment with the URG bit turned on.

Fig. 6-29. Window management in TCP.
Problems with TCP Window Management

- A sender transmits one byte at a time to a remote editor
  - overhead is 41 (for the 1 byte data sent) + 40 (ACK) + 40 (window size update ACK after the character is read) + 41 (for echoing back)
  - For each character typed, overhead is 4 segments with 162 bytes

- A receiver reads data 1 byte at a time
  - this is called the silly window syndrome
  - receiver’s window size initially is 0
  - when one byte is read in, a window update segment is sent
  - the sender transmits another byte
  - the receiver sets the window size to 0 again

Solution:
- Nagle’s algorithm - a tiny segment is not sent
  - accumulate enough bytes at the sender’s side before sending
- Clark’s algorithm - a small window size update is not sent back
  - accumulate enough bytes at the receiver’s side before passing them to the application, so each window size update segment will not just advertise a small window size

Ex. 6-17: Disadvantage of Nagle’s algorithm?
Ans: several characters typed may not be seen for a while and then suddenly appear all at once
TCP Congestion Control

- **Detecting Congestion**
  - Consider why a segment might be dropped
    - link error - but Internet links are reliable now (not true for radio)
    - router buffer overflow due to congestion
  - Internet TCP assumes that timeouts are caused by congestion

- **Two sources of problems: network capacity and receiver capacity**

Fig. 6-31. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.
**TCP Congestion Control (cont.)**

- **Two separate windows to deal with congestion**
  - a window due to receiver capacity limitation is already in existence
  - add a second window called the congestion window
  - limit # of bytes sent any time is the **minimum** of these two windows

- **Jacobson’s slow start algorithm to control the size of the congestion window**
  - initially the congestion window size is the maximum segment size
  - grow the congestion window **exponentially** until a threshold is reached, after which grow the congestion window only **linearly**
    - initial congestion window size is one max segment (say 1Kbyte)
    - on ACKs for all segments sent in burst without a timeout
      - if window < threshold, double the congestion window size
      - else increment the congestion window by the max. segment
  - stop until either a timeout occurs or the receiver’s window is reached
  - on timeout
    - set threshold to one half of the current congestion window size
    - restart the exponential-then-linear growth process above
Jacobson’s Slow Start Algorithm

TCP helps IP do congestion control by treating an ICMP Source Quench packet arrival the same way as a timeout.

Fig. 6-32. An example of the Internet congestion algorithm.
An Exercise on Slow Start Algorithm

- Ex 6-19: Suppose that the TCP congestion window is set to 18KB and a timeout occurs. How big will the window be when the 5th burst is sent if the next four bursts are all successful? Assume that the maximum segment size is 1KB.

- Ans: a timeout occurs when the congestion window size is 18KB, so the threshold is 9KB. After that, the congestion window size doubles for each successful burst sent until it reaches the threshold.

\[
1KB \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 9
\]

Repeat the same question: window size when the 6th is sent with all first 5 being successful? Ans: 10.
TCP Timer Management

- Problem: How to pick a proper timeout value?
  - need to estimate the round-trip time ($RTT$) with a low variance

- Jacobson’s Algorithm: dynamically estimating $RTT$
  - $RTT = \alpha RTT + (1 - \alpha) M$
    - $M =$ time taken for an ACK to return (no retransmission)
    - $\alpha =$ a weight given to the old value, typically 7/8
  - Need to pick a retransmission timer value
    - old policy, use $\text{Timeout} = RTT \times \beta$, with $\beta = 2$
    - new policy: make $\beta$ proportional to standard deviation of $RTT$
    - estimate standard deviation of $RTT$ using mean deviation
      - $D = \alpha D + (1 - \alpha) / RTT - M /
      - $\text{Timeout} = RTT + 4D$
  - Should we update $RTT$ on a retransmission?
    - When an ACK comes in, it is not clear if it is for the 1st or 2nd?
    - Karn’s algorithm (in use): double Timeout on a retransmission

Jacobson’s Algorithm:

$$RTT = \alpha RTT + (1 - \alpha) M$$

- $M =$ time taken for an ACK to return (no retransmission)
- $\alpha =$ a weight given to the old value, typically 7/8

New Policy:

- $\beta =$ proportional to standard deviation of $RTT$
- Estimate standard deviation of $RTT$ using mean deviation
  
  $$D = \alpha D + (1 - \alpha) / RTT - M /$$
  
  $$\text{Timeout} = RTT + 4D$$

Karn’s Algorithm:

- Double Timeout on a retransmission
Other TCP Timers

- **Persistence Timer**
  - Prevents deadlock due to lost segments which inform current window-size.
  - This is a problem if the window was set to 0.
  - On timeout, a probe is sent to the receiver asking for the current window size.

- **Keep-alive Timer**
  - Prevents half-open connections.
  - On timeout, a probe is sent to the other side checking if it is here; if no response, the connection is terminated.
  - May consume bandwidth.
  - May kill live connections when the network hiccups.

- **TIMED WAIT timer**
  - Makes sure that all packets created by a connection have died off.
  - The timeout value can be set to the max packet lifetime (T).
UDP (User Data Protocol)

- A connectionless transport protocol supported by Internet -- no connection establishment is needed
  - Suitable for client-server applications for which one request and one response are needed
  - UDP segment: an 8-byte header + data
  - encapsulated with the IP payload field

Ex. 6-23: Why is UDP needed? Couldn’t a user process just send raw IP packets? IP is only to a destination host; UDP is to a process (port) on the host machine
Wireless TCP

- In theory, TCP should be independent of the underlying network layer but in practice it has been implemented based on wired networks, not wireless networks
  - TCP assumes that timeouts are caused by congestion, not by lost packets
  - this assumption is not valid when IP is running over radio
  - wired (IP running on fiber)
    - a timeout means congestion => slow down transmission
  - wireless (IP running on radio)
    - a timeout means lost packets => speed up transmissions or the throughput drops further

- End-to-end path is heterogeneous in wireless TCP/UDP
  - 1000km wired path followed by a 1km wireless path
  - so when a timeout occurs, what does it mean?
Wireless TCP (cont.)

- **indirect TCP**: split the TCP into two separate connections
  - advantage: both connections are homogeneous: timeouts on the 1st connection can slow the sender down (due to congestion), while timeouts on the 2nd can speed it up (due to link error)
  - disadvantage: the semantics of TCP is violated: receipt of an ACK by the sender does not mean the receiver really got it!

- **Balakrishnan’s solution**: still maintain the semantics of TCP
  - modifications to the network layer code in the base station
  - snooping - monitoring in and out segments and performing retransmissions transparently
    - a segment into mobile but no ACK back: retransmit to the mobile
    - duplicate ACKs from the mobile: discard and retransmit to the mobile
    - a missing segment from the mobile in seq. #: generate a request to the mobile
  - disadvantage: false congestion over the wired networks on timeout
ATM AAL Layer Protocols

- **AAL (ATM Adaptation Layer) is not a true transport layer**
- **Radically different from TCP - Rapid delivery is more important than accuracy in transmitting voice and video**
- **Structure: Two sublayers**
  - **CS (Convergence Sublayer)**
    - provides the interface to applications
    - concerns with messages: message boundaries are preserved
    - accepts bit streams or messages from the applications
    - adds its own header and trailer to each message (1-65535B) for error control and message framing (needed for a bit stream)
  - **SAR (Segmentation And Reassembly Sublayer)**
    - breaks a CS message into 48-byte cells at the sending end
      - may add its own header and trailer to each cell for cell sequencing and error control
    - reassembles cells into messages at the receiving end
    - concerns with cells
ATM AAL Layer Protocols (cont.)

- The ATM reference model showing the ATM AAL layer, ATM layer and the physical layer

CS deals with messages while SAR deals with cells

Both the CS and SAR sublayers can do error control and framing

The text is not correct in saying that CS breaks messages into cells

Fig. 6.37. The ATM model showing the ATM adaptation layer and its sublayers.
ATM AAL Layer Protocols (cont.)

- Flow Diagram from the application layer to the CS/SAR sublayers in the ATM AAL layer and then to the ATM layer
- CS sublayer’s overhead per message is 8 bytes for both AAL 3/4 and AAL 5
- SAR’s overhead per cell is 4 bytes for AAL 3/4, but 0 byte for AAL 5

Fig. 6-38. The headers and trailers that can be added to a message in an ATM network.
AAL 5

- One of the 4 AALs (AAL-1, AAL-2, AAL 3/4 and AAL 5)
  - Believed to be the future AAL to be used by the computer industry
- Unlike AAL 1-4, there is no overhead per cell in AAL 5
  - the CS sublayer pads the message such that the entire message, including padding and the 8-byte trailer, is a multiple of 48 bytes
  - the SAR sublayer does not add any headers or trailers; it just breaks the message into 48-byte units and passes them to ATM
  - how does SAR reassembles cells if no fields are added at the SAR layer?
    - No sequence number: use 32-bit CRC at the CS layer to detect errors
    - No “I am the last cell” field: use the Payload Type in an ATM cell header
- Can support both message mode and stream mode
- Similar to UDP in Internet that supports datagram services to pass a datagram with up to 65,535 bytes

---

For message sequencing
Length of the true payload, not counting the padding

---

Fig. 6-44. AAL 5 convergence sublayer message format.
Performance Issues

- **Synchronous overload**
  - Broadcast storms
    - 10,000 receivers send back an error notification to a broadcast TPDU caused by a bad parameter
  - Reboot storms
    - hundreds of machines reboot after a power failure
      - issue RARP (reverse address resolution protocol) queries
      - request a file server to get a copy of the OS

- **Lack of proper tuning**
  - improper scheduling/buffer allocation algorithms
    - lack of buffer space even though memory is available
  - improper timeout period for the retransmission timer
    - too short - unnecessary retransmissions
    - too long - unnecessary delays for a lost TPDU
  - improper piggybacking policy - how long to wait to piggyback data?

- **Gigabit Networks - can old protocols still be used?**
How to Measure Performance

- Ensure sample size is large enough - repeat measurement
- Make sure samples are representative
  - consider time of day, location, day of week, etc.
- Watch for clock resolution/accuracy
  - if the clock resolution is poor, aggregate the measure over multiple iterations and then take the average
  - Ex. 6-33: measuring the time to receive a TPDU by reading the clock value as a timer interrupt occurs and also once when the TPDU is fully processed; suppose you read 0 msec for 270,000 time and 1 msec for 730,000 times (on interrupts), what is the average receiving time?
    - Ans: $(0*270,000 + 1*730,000)/1,000,000 = 730 \mu \text{sec}$
- Watch for effect of caching/buffering on measurements
  - file transfer/web page times: is a cache going to distort results?
- Careful not to extrapolate too far
  - results generally hold for an operating region, not for all values, e.g., response time ($R$) vs. load ($\rho$) for M/M/1 is linear for $\rho < 0.4$ but raised sharply for $\rho > 0.4$ since $R = (1- \rho)^{-1}$
How to Design for Better Performance

- **Software speed is more important than link speed**
  - protocol software overhead dominates transmission time, e.g., a RPC on Ethernet takes about 1500 µs of which only 102 µs is due to transmission
  - use simple algorithms to reduce overhead in the software

- **Reduce packet count**
  - there is a large interrupt and processing overhead per packet
  - so big packets amortize this overhead over more bytes

- **Minimize context switches**
  - user/kernel boundary crossings are expensive
    - require many cache misses, pipeline stalls, etc.
  - buffer data into a large TPDU before transmit at the sending end and collect small TPDU before passing to the user at the receiving end

- **Minimize copying by the OS**
  - each copy is extra time: hardware -> kernel -> network layer -> transport
  - memory copy operations are often 3 times slower than other instructions
  - intelligent buffer management helps -- e.g., TCP and IP can each use a prototype header to fill in new fields and IP uses header and data pointers passed in by TCP directly without doing copying
How To Design for Better Performance (cont.)

- **Bandwidth is growing, but latency isn’t shrinking as fast**
  - fundamental limit is the round-trip delay
  - need to improve the protocol software, OS, and network interface to reduce the latency

- **Congestion avoidance beats recovery**
  - getting the network out of a bad state will take time
  - better to prevent getting into it in the first place

- **Avoid timeouts**
  - use NACKs to get info back
  - use conservative (longer) values for timeouts
  - timeouts result in:
    - interrupts (wasted CPU resources)
    - retransmission (wasted bandwidth and extra load on routers)

- **Make the common case run fast**
  - data transmission is more common than connect
  - minimizing processing time when everything goes right is 1st; minimizing processing time when an error occurs is secondary
Gigabit Network Problems

- Can result in low line efficiency because the round-trip propagation delay may dominate the transmission time
  - e.g., 1 Gbps; propagation delay=40 msec; receiver’s buffer=64KB
    - time to send data out of the host = 64K*8b/1Gbps = 0.5 msec
    - time to get the acknowledgement back = 40msec
    - transmission line efficiency = 0.5/40 = 1.25%
  - **Delay-bandwidth product** = line bandwidth * round-trip delay time
    - amount of data put on the line to keep the line going full speed (efficiency= 100%) until the first acknowledgement comes back
    - you can think of it as the pipe capacity, e.g., for the same example above it is 1Gbps * 40msec=40Mb

- CPU speed now cannot catch up with Gbps communication speed
  - e.g., 100-MIPS machine exchanging 4-KB packets over a Gbps line => about 30,000 packets/sec in => 1500 instructions/packet (half of the CPU is reserved), or 15 µs per packet processing time => protocols must be simple to meet this requirement

Ex 6-36: This will be even worse for 128B packets: => only 47 inst/packet
Gigabit Network Problems (cont.)

- **32-bit sequence number is not enough**
  - TCP uses sequence numbers for bytes => 32 seconds to use up all sequence numbers over 1 Gbps line (to send $2^{32}$ bytes or $2^{35}$ bits)
  - but in Internet packets can live for 120 sec
  - this limits the sender’s data rate
- **Ex 6-35: how about 64-bit sequence numbers?**
  - theoretical data rate of an optical fiber is 75Tbps
  - assume every byte has its own sequence number, as TCP does
  - it will take $(2^{64} \times 8)/75\text{Tbps}$ or about 3 weeks to recycle the sequence number, so the maximum life time of a packet is 3 weeks
- **Some gigabit applications** such as multimedia require Jitter-free, so it requires uniform delivery rate to keep standard deviation of packet arrivals low
  - slow but uniform is better than fast but jumpy
Dealing with Gigabit Network Problems

- **Design for speed, not for bandwidth optimization**
  - concentrate on minimizing processing time for the normal case to make the protocol run faster
  - minimize packet copying - use assembly code to do copy for optimization

- **No feedback on flow control**
  - Use *rate-based*, rather than credit-based, flow control
  - avoid the long delay for the receiver to feedback the sender
  - dynamic sliding window based on Jacobson’s slow start algorithm is not appropriate for Gb networks

- **Use connection-oriented, bandwidth-reservation designs to reduce jitter**

- **Use a new packet layout with gigabit networks in mind**
  - header should contain as few fields as needed to reduce processing time
  - large enough sequence number field to allow a large window
  - data and header should be separately checksummed
  - large data size should be allowed to minimize the header overhead