End-to-End Protocols

- UDP (User Datagram Protocol)
- TCP (Transport Control Protocol)
  - Connection Establishment/Termination
  - Sliding Window Revisit
  - Flow Control
  - Adaptive Retransmission

 limitation of underlying network (e.g., Internet)
 - drop messages
 - re-orders messages
 - delivers duplicate copies of a given message
 - limits messages to some finite size
 - delivers messages after an arbitrarily long delay

Common Properties of the End-to-End Protocols
- guarantee message delivery
- deliver messages in the same order they are sent
- deliver at most one copy of each message
- support arbitrarily large messages
- support synchronization
- allow the receiver to flow control the sender
- support multiple application processes on each host

Simple Processes Demultiplexor (UDP)
- Unreliable and unordered datagram service
- Adds multiplexing
- No flow control
- Endpoints (target process) identified by ports
  - servers have well-known ports
  - see /etc/services on Unix (e.g., DNS: 53)
  - Unique on a single host
  - Implementation diff. on OSs
- Header format
  - Optional checksum (IPv4)
    - Over UDP header/payload, part of IP header (source/des IP address, protocol number)

Header format

<table>
<thead>
<tr>
<th>SrPort</th>
<th>DstPort</th>
<th>Length</th>
<th>Checksum</th>
<th>Data</th>
</tr>
</thead>
</table>

Ref: Data & Computer Communications by Stallings
### TCP Overview

- **Reliable, Connection-oriented**
- **Byte-stream**
  - app *writes* bytes
  - TCP *sends* segments
  - app *reads* bytes
- **Full duplex**

#### Flow control
- keep sender from overrunning receiver, an end-to-end issue ("how much")

#### Congestion control
- keep sender from overrunning network; concerned with how hosts and networks interact ("how fast")

TCP frees the application from the worry of missing or reordered data.

### End-to-End Issues: Data Link Vs Transport

- **Core of TCP:** sliding window algorithm
- Potentially connects many different hosts
- need explicit connection establishment and termination
- Potentially different RTTs
- need adaptive timeout mechanism
- Potentially long delay in network
- need to be prepared for arrival of very old packets (MSL:120s)
- Potentially different capacity at destination
- need to accommodate different amounts of buffering -> flow control
- Potentially different network capacity
- need to be prepared for network congestion (Chapter 6)
- X.25 use sliding window on a hop-by-hop basis

### Segment Format

- Although TCP provides "byte stream" service to application processes, TCP itself does not transmit individual bytes over the Internet, but a packet (segment) exchanged between TCP peers
- Trigger mechanisms: MSS without causing local IP to fragment; TCP sender "push" operation (e.g., telnet);

### Segment Format (cont)

- Each connection identified (TCP demux key) with 4-tuple:
  - (SrcPort, SrcIPAddr, DsrPort, DstIPAddr)
- Sliding window + flow control
- acknowledgment, SequenceNum, AdvertisedWindow

#### Flags
- SYN, FIN, RESET, PUSH, URG, ACK

#### Checksum
- pseudo header + TCP header + data
Connection Establishment and Termination

Three way handshake algorithm
(note: a timer is scheduled for each of the first 2 segments)

Q. Why exchange starting Seq. # instead of starting from default number such as 0?
(two incarnations)

Note: TCP Connection setup is asymmetric, while connection tear down is symmetric; Once a connection setup, it is a bidirectional connection

Revisit Sliding Window: Sender

- Assign sequence number to each frame (SeqNum)
- Maintain three state variables:
  - send window size (SWS): upper bound on the # of outstanding (un-ACKed) frames allowed.
  - sequence # of last acknowledgment received (LAR)
  - sequence # of last frame sent (LFS)
- Maintain invariant: LFS - LAR <= SWS at all time
- Advance/update LAR when ACK arrives to allow a new frame be sent
- Buffer up to SWS frames for retransmission if needed

Basic Idea:
- Allow sender to transmit multiple frames before receiving an ACK, thereby keeping the pipe full. There is an upper limit (called window) on the number of outstanding (un-ACKed) frames allowed.
  - Size of window sets amount of data that can be sent w/o waiting for ACK from the receiver

Sender
Receiver

Time: 1 2 3 4 5 6 7 8 9 10

Lit: normal path for client

Dashed line: normal path for server

Ref: book by Leon-Garcia
**Revisit Sliding Window: Receiver**

- Maintain three state variables
  - receive window size (RWS): upper bound on the # of out-of-order frames (Why?)
  - sequence # of largest acceptable frame (LAF)
  - sequence # of last frame received (LFR) in order

- Maintain invariant: LAF - LFR <= RWS

- Frame SeqNum arrives:
  - if LFR < SeqNum <= LAF accept
  - if SeqNum <= LFR or SeqNum > LFA discard

- Mechanism of Sending cumulative ACKs
  - LFR = SeqNum to Ack (largest seq # not yet acknowledged)
  - LAF = LFR + RWS

---

**TCP: Sliding Window Revisit**

- ARQ: Stop-and-wait -> Sliding Window
  - delay x bandwidth

- 1) reliable delivery:
- 2) in-order delivery:
- 3) flow control:
  - dynamic window size

- Sending side
  - LastByteAcked <= LastByteSent
  - LastByteSent <= LastByteWritten
  - buffer bytes between LastByteAcked and LastByteWritten

- Receiving side
  - LastByteRead < NextByteExpected
  - NextByteExpected <= LastByteRcvd + 1
  - buffer bytes between LastByteRead and LastByteRcvd

---

**Flow Control**

- Send buffer size: MaxSendBuffer
- Receive buffer size: MaxRcvBuffer

- Receiving side
  - LastByteRcvd - LastByteRead <= MaxRcvBuffer
  - AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - LastByteRead)

- Sending side
  - LastByteSent - LastByteAcked <= AdvertisedWindow
  - EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
  - LastByteWritten - LastByteAcked <= MaxSendBuffer
  - block sender process if (LastByteWritten - LastByteAcked) + y > MaxSenderBuffer

- Always send ACK (AWS + NextByteExpt) in response to arriving data segment
- Persist when AdvertisedWindow = 0 (1 byte probe segment)

---

**Protecting Against Wrap Around**

- TCP: 16bits for AdvertiseWindow, 32bits for SeqNum
- Sequence # space be at least twice of window size
- SeqNum should not wrap around within MSL (120 seconds)
  - Assume the full pipe
- 32-bit SequenceNum
  - \(2^{32} \times 8 / BW\)

<table>
<thead>
<tr>
<th>Bandwidth (BW)</th>
<th>Time Until Wrap Around</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>13 minutes</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>55 seconds</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>
Sequence Number Space (sliding window at data link layer)

- **SeqNum** field is finite; sequence numbers wrap around (reuse)
- Sequence number space must be larger than number of outstanding frames
- **SWS <= MaxSeqNum-1** is not sufficient
  - suppose 3-bit SeqNum field (0..7)
  - SWS=RWS=7
  - sender transmit frames 0..6
  - arrive successfully, but all ACKs lost
  - sender retransmits 0..6
  - receiver expecting 7,0..5, but receives second incarnation of 0..5
- **SWS < (MaxSeqNum+1)/2** is correct rule when SWS = RWS
  - if RWS=1, SWS <= MaxSeqNum-1 is sufficient

Keeping the Pipe Full

- **Delay x Bandwidth** product
- 16-bit **AdvertisedWindow** (assume RTT of 100ms)
  - Allows to advertise a window of only 64KB

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>16KB &lt; 64KB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122KB &gt; 64KB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>549KB &gt; 64KB</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>1.2MB &gt; 64KB</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>1.8MB &gt; 64KB</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>7.4MB &gt; 64KB</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>14.8MB &gt; 64KB</td>
</tr>
</tbody>
</table>

- TCP Extension (reading assignment: p408-409)

Adaptive Retransmission (Original Algorithm)

- In Internet, "range" & "variation" of RTT could be big
- TCP **retransmit** if an ACK is not received within ... time (as a function of RTT)
- **Measure** SampleRTT for each segment/ACK pair
- **Compute** weighted average of RTT
  - EstRTT = α x EstRTT + β x SampleRTT
  - where α + β = 1
    - α between 0.8 and 0.9
    - β between 0.1 and 0.2
- **Set timeout** based on EstRTT
  - TimeOut = 2 x EstRTT
- TCP Dynamics (vs. data link layer sliding window mech.)
  - Advertised window (how much data the source can send)
  - Adaptive Timeout (how fast data can be resent by a specific source)

Karn/Partridge Algorithm

- **Problem**: Ack indicates the receipt (not transmission) of data
- **Solution**: Do not sample RTT when retransmitting
  - Double timeout for each retransmission (exponential backoff): intuition?
Jacobson/Karels Algorithm

- Karn/Partridge algorithm does not take the variance into account
- New Calculations for average RTT
  \[ \text{EstRTT} = \alpha \times \text{EstRTT} + \beta \times \text{SampleRTT} \]
- Diff = SampleRTT - EstRTT
- \[ \text{EstRTT} = \text{EstRTT} + (\delta \times \text{Diff}) \]
- \[ \text{Dev} = \text{Dev} + \delta \left( |\text{Diff}| - \text{Dev} \right) \]
  where \( \delta \) is a factor between 0 and 1
- Consider variance when setting timeout value
- \[ \text{TimeOut} = \mu \times \text{EstRTT} + \phi \times \text{Dev} \]
  where \( \mu = 1 \) and \( \phi = 4 \)
- Notes
  - accurate timeout mechanism important to congestion control (later)