Reliable Byte-Stream (TCP)

Outline
Connection Establishment/Termination
Sliding Window Revisited
Flow Control
Adaptive Timeout

End-to-End Protocols

- Underlying best-effort network
  - drop messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - limits packet (not message) to some finite size
  - delivers messages after an arbitrarily long delay
- Common end-to-end services
  - 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 between sender and receiver
  - allow the receiver to flow control the sender
  - support multiple application processes on each host
Simple Demultiplexor (UDP)

- Unreliable and unordered datagram service
- Adds multiplexing
- No flow control or error control
  - no need for sender-side buffer
- Endpoints identified by ports
  - servers listens at well-known ports!
  - see /etc/services on Unix
- Header format
  - Optional checksum
    - pseudo header (IP.src, IP.dsest, IP.proto, UDP.len) + UDP header + data

TCP Overview

- Connection-oriented
- Byte-stream
  - app writes bytes
  - TCP sends segments
  - app reads bytes
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network
Data Link Versus End-to-End Transport

- Potentially connects many different hosts
  - need explicit connection establishment and termination
- Potentially different RTT
  - need adaptive timeout mechanism
- Potentially long delay in network
  - need to be prepared for arrival of very old packets
- Potentially different capacity at destination
  - need to accommodate different node capacity
- Potentially different network capacity
  - need to be prepared for network congestion

Segment Format

```
<table>
<thead>
<tr>
<th>4</th>
<th>10</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequenceNum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
<td>Flags</td>
<td>AdvertisedWindow</td>
</tr>
<tr>
<td>Checksum</td>
<td>UrgPtr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Segment Format (cont)

- Each connection identified with 4-tuple:
  - \((\text{SrcPort, SrcIPAddr, DsrPort, DstIPAddr})\)
- Sliding window + flow control
  - acknowledgment, SequenceNum, AdvertisedWindow

  ![Segment Format Diagram]

- Flags
  - \text{SYN}, \text{FIN}, \text{RESET}, \text{PUSH}, \text{URG}, \text{ACK}
- Checksum
  - pseudo header + TCP header + data

Connection Establishment and Three-Way Handshake

<table>
<thead>
<tr>
<th>Active participant (client)</th>
<th>Passive participant (server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{SYN, SequenceNum #}</td>
<td></td>
</tr>
<tr>
<td>\text{SYN+ACK, SequenceNum=y}</td>
<td>\text{Acknowledgment=x+1}</td>
</tr>
<tr>
<td>\text{ACK, Acknowledgment=y+1}</td>
<td></td>
</tr>
</tbody>
</table>
State Transition Diagram

**event / action**

**event:** receiving a segment, or an operation invoked by application

**State Transition Diagram (cont)**

- Data transfer occur in the ESTABLISHED state
- Open a connection
  - Server listens and waits for SYN.
  - If the client’s ACK to the server is lost, connection is still established, due to cumulative ACKs
- Terminate a connection
  - Both sides can terminate
    - Case 1: one side closes first
    - Case 2: both sides close at the same time
  - TIME_WAIT to CLOSED: wait for 120 seconds
    - The other side might retransmit FIN while waiting for ACK
    - The next TCP connection might reuse the same port.
Connection Termination – One Side Closes First

Connection Termination – Both Sides Close
Sending Buffer and Receiving Buffer

- The receiver’s buffer has two purposes
  - Reorder segments received out of order
  - Hold data unread by the application
- The receiver sends \textit{AdvertisedWindow} in ACK
- The sender cannot send more than \textit{AdvertisedWindow} bytes of unacknowledged data at any given time (Flow Control).
- The receiver selects a suitable \textit{AdvertisedWindow} based on the available memory and application reading speed.

---

Sliding Window Revisited

- Sending side
  - \( \text{LastByteAcked} \leq \text{LastByteSent} \)
  - \( \text{LastByteSent} \leq \text{LastByteWritten} \)
  - buffer bytes between \( \text{LastByteAcked} \) and \( \text{LastByteWritten} \)
- Receiving side
  - \( \text{LastByteRead} < \text{NextByteExpected} \)
  - \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)
  - buffer bytes between \( \text{NextByteRead} \) and \( \text{LastByteRcvd} \)

\textbf{Implementation:} circular buffers
Flow Control

- **MaxSendBuffer** and **MaxRcvBuffer**
- Receiving side
  - $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$
  - $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$
- Sending side
  - $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
    - block sender if $(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSenderBuffer}$
  - $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
  - $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$ (how much more original data can be sent)
- Always send ACK in response to arriving data segment
- Persist when $\text{AdvertisedWindow} = 0$
  - Sender sends 1 byte of data every so often.

Adaptive Retransmission
(Original Algorithm)

- Measure **SampleRTT** for each segment / ACK pair
- Compute weighted average of RTT
  - $\text{EstRTT} = \alpha \times \text{EstRTT} + \beta \times \text{SampleRTT}$
  - where $\alpha + \beta = 1$, $0.8 \leq \alpha \leq 0.9$, $0.1 \leq \beta \leq 0.2$
  - Smooth noisy measurements
- Set timeout based on $\text{EstRTT}$
  - $\text{TimeOut} = 2 \times \text{EstRTT}$
  - 2: to be conservative
Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission
  - When the retransmitted segment is ACKed, timeout value is reduced to $2 \times \text{EstRTT}$

Jacobson/ Karels Algorithm

- Takes the variances of sampled RTT into account
  - if the var is small, no need to multiply $\text{EstRTT}$ by 2.
- $\text{Diff} = \text{SampleRTT} - \text{EstRTT}$
- $\text{EstRTT} = \text{EstRTT} + (\delta \times \text{Diff})$
  $$= (1 - \delta) \times \text{EstRTT} + \delta \times \text{SampleRTT}$$
- $\text{Dev} = \text{Dev} + \delta(\mid \text{Diff} \mid - \text{Dev})$
  $$= (1 - \delta) \times \text{Dev} + \delta \mid \text{Diff} \mid$$  
  - where $\delta$ is a factor between 0 and 1
- $\text{TimeOut} = \mu \times \text{EstRTT} + \phi \times \text{Dev}$
  - where $\mu = 1$ and $\phi = 4$
- Notes
  - algorithm only as good as granularity of timer (500ms on Unix, 100ms on Linux)
  - accurate timeout mechanism important to congestion control (later)
Silly Window Syndrome

• MSS (Max Segment Size) is set to (local MTU – TCP/IP header)
• The TCP sender may sends tiny segment into networks
  – if the effective window is less than MSS
  – if the application generates data one byte at a time
• Inefficient use of bandwidth: 4000% overhead of TCP/IP header
• Not aggregates afterwards due to the ACK self-clocking mechanism.

Nagle’s Algorithm

• How long does sender delay sending data?
  – too short: poor network utilization
  – too long: hurts interactive applications
  – how long? utilize ACK self-clocking to simulate a timer
• If there is unACKed data in transit: buffer it until ACK arrives; else send it
Message Boundaries

- **UDP socket API** is message-oriented (datagram sockets)
  - Individual datagrams (sent with separate calls) will be kept separate when they are received. A `recvfrom()` call on a datagram socket will only return the next datagram.
  - Applications pick the segment size.
    - Could be segmented by IP.
- **TCP socket API** is byte-oriented (stream sockets)
  - Message boundaries addressed by the application layer protocol.

Problem: Keeping the Pipe Full

- 16-bit **`AdvertisedWindow`** allows 64KB

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>18KB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122KB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>549KB</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>1.2MB</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>1.8MB</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>7.4MB</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>14.8MB</td>
</tr>
</tbody>
</table>

assuming 100ms RTT
Problem: Protection Against Wrap Around

• 32-bit **SequenceNum**
  – 16-bit **AdvertisedWindow**: $2^{32} \gg 2^{16}$

• Another byte with the same sequence number x could be sent once again, if window size is large enough (e.g 1GB)

<table>
<thead>
<tr>
<th>Bandwidth</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>

TCP Extensions

• Implemented as header options

• Store timestamp in outgoing segments
  – for fine-grained RTT measurements

• Extend sequence space with 32-bit timestamp (PAWS)
  – for packet differentiation
  – not for reordering or acknowledging

• Shift (scale) advertised window