**IP Hourglass (redux)**

Diagram showing various protocols and technologies:
- Telecollaboration
- NFS
- HTTP
- email
- rlogin
- RSVP
- RPC
- TCP
- UDP
- IP
- Ethernet
- modem
- ATM
- PPP
- SONET
- packet radio
- air
- 100BT

**User Datagram Protocol (UDP)**

UDP Header:
- Link-layer
- IP
- SrcPort
- DestPort
- Checksum
- Len
- Data...

UDP Header indicates where the data begins and ends.
Transmission Control Protocol (TCP)

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>10</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DestPort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequenceNum</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>000000</td>
<td>Flags</td>
<td>AdvertisedWindow</td>
<td></td>
</tr>
<tr>
<td>CheckSum</td>
<td>UrgPtr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable – max of 320 bits)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ACKs and Timeouts

- Time
- Timeout
- Packet
- ACK
- ACK lost
- Packet lost
- Early timeout/Delayed ACK
Sequence and ACK Numbers

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq = 123, Len = 50</td>
<td>Ack = 173</td>
</tr>
<tr>
<td>Seq = 173, Len = 40</td>
<td>Ack = 213</td>
</tr>
<tr>
<td>Seq = 213, Len = 60</td>
<td>Ack = 273</td>
</tr>
</tbody>
</table>

TCP Three-Way Handshake

- **Client**
  - SYN, sequence # = x
  - SYN+ACK, sequence # = y
  - ACK, Acknowledgement = y + 1

- **Server**
  - SYN, sequence # = x
  - ACK, Acknowledgement = x + 1
Filling the Pipe

- Each side of a TCP connection can independently close the connection.
- Thus, possible to have a half duplex connection.
- Possible problems?
- Solutions?

- Closing process sends a FIN message.
- Waits for ACK of FIN to come back.
- This side of the connection is now closed.

Reliability, First Cut: Stop and Wait

- Time
- Packet
- ACK
- Timeout
- Sender
- Receiver

- Reliability, two principal mechanisms:
- ACKs and timeouts
- Send a packet, stop and wait until acknowledgement arrives before sending next packet.

Problems?

- Recovering From Error

Packet
ACK
Timeout
Packet
ACK
Timeout
Packet
ACK
Timeout
Packet
ACK
Timeout
Packet
ACK
Timeout
ACK lost
Packet lost
Early timeout/
Delayed ACK

Problems with Stop and Wait

- How to recognize a duplicate transmission?
- Solution: Put sequence number in packet.

Performance

- Unless Latency-Bandwidth product is very small, sender cannot fill the pipe.
- Solution: Sliding window protocol with dynamically changing window size.

Keeping the Pipe Full

- Bandwidth-Delay product measures network capacity.
- How much data can you put into the network before the first byte reaches receiver?
- Stop and Wait: 1 data packet per RTT (round trip time).

Example: 10 Mbps link, 100 ms RTT:

- How much data is needed to keep pipe full?
- 10 x 10^6 bps * 100 x 10^-3 s = 1,000,000 bits = 125 KB

Lost Packets
TCP Sliding Window (1)

Sender
- 1 2 3
- 4 5 6
- 7 8 9
- 10 11 12

Sender
- sent and acknowledged
- sent, not ACKed
- can send ASAP
- can't send until window moves

Receiver
- 1 2 3
- 4 5 6
- 7 8 9
- 10 11 12

Receiver
- ACK’d and read
- Available bufs
- can't recv until window moves

TCP Sliding Window (2)

Sender
- 1 2 3 4 5
- 6 7 8 9
- 10 11 12

Sender
- sent and acknowledged
- sent, not ACKed
- can send ASAP
- can't send until window moves

Receiver
- 1 2 3
- 4 5
- 6 7 8 9
- 10 11 12

Receiver
- ACK’d and read
- Available bufs
- ACK’d, not read
- can't recv until window moves

Solution?
- Circular buffer at sender and receiver
- Sliding window with changing window size
- System will eventually block (through backpressure)
- Only finite space available at each location
- # packets in transit <= buffer size
- Receiver ACKs 1, 2, 3, 3, 3, 3

What happens when receiver receives 1, 2, 3, 5, 6, 7?

ACK 5, Window 4

Read

Reliable

Flow Control

NIC Buffer

OS Buffer

recv

TCP Sliding Window (3)

Sender

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>sent and acknowledged</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>advertised window</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Receiver

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK’d and read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>offered window</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Send 7, 8, 9

TCP Congestion Control

NOTE: Advertised window expressed in bytes, not packets.

How can we detect congestion?

Options for Sender Discovery of

Receiver App reads packets 4, 5, 6

TCP Sliding Window (4)

Sender

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>advertised window=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Receiver

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK’d and read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>offered window=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ACK 9, Window 0

Available bufs

of the window moves

can’t send until

can’t recv until

window moves

window moves

window moves

window moves

window moves
TCP Sliding Window (5)

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10 11 12</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>advertised window=0</td>
<td>offered window=3</td>
</tr>
<tr>
<td>sent and acknowledged</td>
<td>ACK'd and read</td>
</tr>
<tr>
<td>can't send until window moves</td>
<td>ACK'd, not read</td>
</tr>
</tbody>
</table>

Read bytes 4, 5, 6

How does TCP do it?

Basic Implementation

- **Sender**
  - Sends packets 1-6
  - Receiver acknowledges packets
  - Receiver sends duplicate ACK with a larger advertised window

- **Receiver**
  - Reads packets 4, 5, 6
  - Increases window to send faster

Congestion Control

- **Source**
  - 100 Mbps Ethernet

- **Router**
  - 45 Mbps T3 link

- **Sink**

How fast should we send initially?

- Slow start
- Increase window to send faster; decrease to send slower

How should we adjust our sending rate?

- Congestion control and flow control mechanisms
- Artificially constrain number of outstanding packets allowed in network
- Increase window to send faster; decrease to send slower

Options for Sender Discovery of Source Quench:

- Rate-based (many streaming media protocols)
- Pro: Smooth traffic
- Pro: Cheap to implement, good failure properties
- Con: Creates traffic bursts
- Art: Complicates receiver design
- Idea: decrease window

Questions:

- How can we fix it?
- Jacobson & Karels 88: Seminal paper in computer science
Additive Increase, Multiplicative Decrease (AIMD)

Goal: Adapt to changes in available bandwidth

Why called slow? It's exponential after all…

Algorithm:
- Quickly increase sending rate until congestion detected
- Goal: Quickly find the equilibrium sending rate
- Problem: Takes a long time to rate using AIMD

In steady state:
- Let $cwnd = 1$
- Increase sending rate by a constant (e.g. by 1500 bytes)
- Next time, if $cwnd = 2$
- If timeout then set $cwnd = 3$
- For each segment acknowledged, increment
- On new connection, or after timeout, set $cwnd = 4$

Slow Start Growth Example

Rough intuition for why this works
- Increase sending rate by a constant (e.g. by 1500 bytes)
- For each segment acknowledged, increment
- On new connection, or after timeout, set

$\text{Slow Start Avoidance}$

Avoidance

- Decrease quickly when there is loss (went too far!)
- Why called slow? It's exponential after all…
- Algorithm: Remember last rate that worked
- $cwnd = 2$
- $cwnd = 4$
- $cwnd = 8$
- Increase sending rate by a constant (e.g. by 1500 bytes)
- Next time, if $cwnd = N$
- If timeout then set $cwnd = N/2$
- For each segment acknowledged, increment
- On new connection, or after timeout, set $cwnd = 1$

$\text{Fast Recovery}$

Consequence: Queue size increases exponentially
- During congestion,
- Must reduce sending rate exponentially as well (hence

$\text{Additive Increase/Multiplicative Decrease}$)

Congestion Avoidance

- Decrease quickly when there is loss (went too far!)
- Why called slow? It's exponential after all…
- Algorithm: Remember last rate that worked
- $cwnd = 2$
- $cwnd = 4$
- $cwnd = 8$
- Increase sending rate by a constant (e.g. by 1500 bytes)
- Next time, if $cwnd = N$
- If timeout then set $cwnd = N/2$
- For each segment acknowledged, increment
- On new connection, or after timeout, set $cwnd = 1$

$\text{Fast Retransmit & Recovery}$

- If a packet has made it through -> we can send another one
- If there are still ACKs coming in, then no need for slow start
- Use 3 duplicate ACKs to indicate a loss; detect losses quickly
- When packet is lost, receiver still ACKs last in-order packet
- Timeouts are slow (1 second is fastest timeout on many TCPs)

$\text{Slow Start + Congestion Avoidance}$

Slow Start + Congestion Avoidance

- Timeout
- Congestion avoidance
- ssthresh
- Slow start

$\text{cwnd}$

round-trip times
**TCP Sawtooth Pattern**

Slow Start + Congestion Avoidance + Fast Retransmit + Fast Recovery = Sawtooth

**TCP Link Sharing**

Cheating TCP: ACK splitting

Rule: grow window by one

Growth factor proportional to M!
TCP/UDP Sharing

Fast Retransmit & Recovery

1

2

Ack

3

4

5

6

7

Sender

Receiver

Ack

Ack

Ack

Ack

Ack

Ack

3 Dup Acks

Fast  recovery

Fast Retransmit (don’t wait for timeout)

Fast Recovery in Action

TCP vs. UDP

TCP

• Connection oriented

• On-going conversation

• Heavy-weight

• Reliable delivery

• In-order delivery

• Connection setup and tear down required

• Flow & congestion control

• What apps need TCP?

UDP

• Connection-less

• No notion of conversation

• Light-weight

• No reliability

• No in-order delivery

• No connection setup or tear down

• No flow or congestion control

• What apps don’t need TCP (and can use UDP)?

What if Two TCP Connections Share Link?

• Reach equilibrium independent of initial bandwidth (assuming equal RTTs)

What if TCP and UDP Share Link?

• Independent of initial rates, UDP will get priority!

TCP will take what’s left.

Cheating TCP:

• Rule: grow window by one full-sized packet for each valid ACK received

• Send M ACKs for one pkt

• Growth factor proportional to M!