Lecture 28: Flow Control, Reliable Delivery

**TCP Flow Control**

The receiver side of a TCP connection maintains a receiver buffer:

- Application process may be slow at reading from the buffer.
- **Flow control** ensures that sender won’t overflow receiver’s buffer by transmitting too much, too fast.

**Sliding Window**

TCP uses sliding window flow control: allows a larger amount of data “in flight” than has been acknowledged:
- Allows sender to get ahead of the receiver
- But not too far ahead

**TCP Receiver Window**

- Receiver window size ($rwnd$):
  - Amount that can be sent without acknowledgment
  - Receiver can buffer this amount of data
- Receiver continually advertises buffer space available to sender by including the current value of $rwnd$ in TCP header

Sender limits unACKed data to $rwnd$:
- Guarantees receiver buffer wouldn’t overflow

[Rexford]
TCP Header with rwnd

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S's port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D's port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgment Seq#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>window size</td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution to Silly-window Syndrome

Don’t advertise window until it opens “significantly” (> \( \frac{1}{2} \times \text{MSS} \) or \( \frac{1}{2} \times \text{rwnd} \))

Implementation alternatives:
- ACK with \( \text{rwnd} = 0 \):
  - sender probes after persistence timer goes off
- delayed ACK, but
  - delayed not more than 500 ms
  - or ACK every other segment (Why?)

TCP Flow Control Problems

Two flow-control problems:
1. receiver too slow (silly-window syndrome)
2. sender’s data comes in small amount (Nagle’s algorithm)

Silly-window syndrome:
receiver window opens only by a small amount, hence sender can send only a small amount of data at a time

Why is this not good?
1.
2.

Characteristics of Interactive Applications

User sends only a small amount of data, e.g., telnet sends one character at a time

Problem: 40-byte header for every byte sent!

Solution: “clumping,” sender clumps data together, i.e., sender waits for a “reasonable” amount of time before sending

How long is “reasonable”?

Data

Optimization: Don’t advertise window until it opens “significantly” (> \( \frac{1}{2} \times \text{MSS} \) or \( \frac{1}{2} \times \text{rwnd} \))

Implementation alternatives:
- ACK with \( \text{rwnd} = 0 \):
  - sender probes after persistence timer goes off
- delayed ACK, but
  - delayed not more than 500 ms
  - or ACK every other segment (Why?)

TCP Flow Control Problems

Two flow-control problems:
1. receiver too slow (silly-window syndrome)
2. sender’s data comes in small amount (Nagle’s algorithm)

Silly-window syndrome:
receiver window opens only by a small amount, hence sender can send only a small amount of data at a time

Why is this not good?
1.
2.

Characteristics of Interactive Applications

User sends only a small amount of data, e.g., telnet sends one character at a time

Problem: 40-byte header for every byte sent!

Solution: “clumping,” sender clumps data together, i.e., sender waits for a “reasonable” amount of time before sending

How long is “reasonable”?
Nagle Algorithm

- send first segment immediately
- accumulate data until ACK returns, or
- up to ½ sender window or ½ MSS

Advantages:
- bulk transfer is not held up
- data sent as fast as network can deliver (see next slide)

Can be disabled by `setsockopt(TCP_NODELAY)`

Original TCP Error Recovery

Originally, TCP uses Go-Back-N ARQ:
- sender transmits a window of packets, associating a timeout with each packet
- when a packet is lost, sender waits for ACK until timer expires
- and then retransmits from the lost packet onward

Current TCP Error Recovery

Sender:
- maintains only one active timer, for `snd_una`, restarting the timer after retransmission

Receiver:
- cumulative ACK acknowledges all packets received in-order
- out-of-order packets repeat the last ACK
- buffers out-of-order packets
- error recovery on current TCP is more complicated as it’s tied up with congestion control, but it still relies on retransmission timeout for correctness
TCP Go-back-N with Buffering

Receiver:
- buffers out-of-order packets
- cumulative ACK acknowledges all packets received in-order

For illustration only! Actual error recovery is tied up with congestion control

Retransmission Timeout

ARQ depends on retransmission to achieve reliability: sender sets a timeout waiting for an ACK

Retransmission timeout (RTO) computed from round-trip time (RTT)
- expect ACK to arrive after an RTT
- but on the Internet, RTT of a path varies over time, due to:

Varying RTT complicates the computation of:
1. retransmission timeout (RTO)
2. optimal sender’s window size

Implications of Bad RTO

RTO too small: unnecessary retransmissions:
- estimated_RTT = \alpha \times \text{sample}_\text{RTT} + (1 - \alpha \times \text{sample}_\text{RTT}

RTO too big: lower throughput:
- \alpha = 1: each sample changes the estimate only a little bit
- \alpha = 0: each sample influences the estimate heavily
- \alpha is typically \(\frac{1}{2}\) (1 – \(\frac{1}{2}\)), which allows for fast implementation (3 right shifts)

Estimating RTT

RTO must adapt to actual and current RTT

Estimate the RTT by watching returning ACKs
- compute a smoothed estimate by keeping a running average of the RTTs (a.k.a. Exponentially Weighted Moving Average (EWMA))
- \text{estimated}_\text{RTT} = \alpha \times \text{estimated}_\text{RTT} + (1 - \alpha \times \text{sample}_\text{RTT}

where
- \text{sample}_\text{RTT}: time between when a segment is transmitted and when its ACK is received
- \alpha is the weight:
  - \alpha = 1: each sample changes the estimate only a little bit
  - \alpha = 0: each sample influences the estimate heavily
Example RTT Estimation

How to Compute RTO?

First try: \( \text{RTO} = \beta \text{RTT} \), with \( \beta \) typically set to 2 or 3

Two problems:

1. an ACK acknowledges receipt of data, is not an ACK for transmission: which packet to associate with an ACK in the case of retransmission?
2. RTTs spread too wide

ACK Ambiguity

Which retransmitted packet to associate with an ACK?

1. original packet: RTO can grow unbounded
2. retransmitted packet: RTO shrinks

There is a feedback loop between RTO computation and RTT estimate

ACK Ambiguity: Karn’s Algorithm

Karn’s algorithm:

- adjust RTT estimate only from non-retransmitted samples
- however, ignoring retransmissions could lead to insensitivity to long delays
- so, back off RTO upon retransmission:
  \( \text{RTO}_{\text{new}} = \gamma \text{RTO}_{\text{old}} \), \( \gamma \) typically = 2
RTT Spread Too Wide

RTT estimate computed using EWMA only considers the mean, doesn’t take variance into account

![Graph showing RTT spread]

Jacobson’s algorithm:
• estimate deviation \((D)\) of sample\_RTT
  \[ D_{\text{new}} = \alpha D_{\text{old}} + (1 - \alpha) |\text{sample\_RTT} - \text{estimated\_RTT}| \]
• compute new estimated\_RTT as usual
• take the deviation in sample\_RTT \((D)\) into account when computing RTO
  \[ \text{RTO} = \text{estimated\_RTT} + 4D \]

Timers Used in TCP

1. **TIME\_WAIT**: 2*MSL
2. **persistence timer**
3. **RTO**
4. **keep-alive timer**: probe the other side if connection has been idle for “too long”
   • may be turned on/off
   • idle period may be set using `setsockopt()`