CSCI-1680
Transport Layer II

Data over TCP: Flow Control

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
Adminstrivia

- Make sure you have signed up for IP grading
- TCP milestone I: Thurs, Apr 14
- TCP gear-up early next week, look for details
Topics for today

- Flow control: Sliding window
- Computing RTO
- Connection termination
Sliding window: in abstract terms

• Window of size \( w \)
• Can send at most \( w \) packets before waiting for an ACK
• Goal
  – Network “pipe” always filled with data
  – ACKs come back at rate data is delivered
    => “self-clocking”
Sender example
Receiver example
Flow Control: Sender

Invariants

- LastByteSent – LastByteAcked <= AdvertisedWindow
- EffectiveWindow = AdvertisedWindow – (BytesInFlight)
- LastByteWritten – LastByteAcked <= MaxSendBuffer

Useful Sliding Window Terminology:
RFC 793, Sec 3.3
Flow control: receiver

- Can accept data if space in window
- Available window =BufferSize - (NextByteExpected - 1) - LastByteRead

On receiving segment for byte S
  - if s is outside window, ignore packet
  - if s == NextByteExpected:
    • Deliver to application (Update LastByteReceived)
    • If next segment was early arrival, deliver it too
  - If s > NextByteExpected, but within window
    • Queue as early arrival

Send ACK for highest contiguous byte received, available window
Flow Control

- Advertised window can fall to 0
  - How?
    - Sender eventually stops sending, blocks application

- Resolution: zero window probing: sender sends 1-byte segments until window comes back > 0
Unfilled buffer
Data received, but not acknowledged
Data received, acknowledged and delivered to application
Data received, acknowledged, but not yet delivered to application
Initial sequence number
Receiver's window (Allocation buffer) Up to 2^{16}-1 slots
Window shifts
Sequence numbers (Circumference = 0 to 2^{32} slots)
Some Visualizations

• Normal conditions: https://www.youtube.com/watch?v=zY3Sxvj8kZA

• With packet loss: https://www.youtube.com/watch?v=lk27yiIToU
How do ACKs work?

- **ACK contains** next expected sequence number
- If one segment is missed but new ones received, send duplicate ACK
- Retransmit when:
  - Receive timeout (RTO) expires
  - Receive 3 Duplicate ACKs

- How to set RTO?
When to time out?

Should expect an ACK within one Round Trip Time (RTT)

- Problem: RTT can be highly variable

- Strategy: expected RTT based on ACKs received
  - Use exponentially weighted moving average (EWMA)
  - RFC793 version ("smoothed RTT"):
    \[
    \text{SRTT} = (\alpha \times \text{SRTT}) + (1 - \alpha) \times \text{RTT}_{\text{Measured}}
    \]
    \[
    \text{RTO} = \max(\text{RTO}_{\text{Min}}, \min(\beta \times \text{SRTT}, \text{RTO}_{\text{Max}}))
    \]

\(\alpha\) = "Smoothing factor": .8-.9
\(\beta\) = "Delay variance factor": 1.3—2.0

RFC793, Sec 3.7
This is only the beginning...

- Problem 1: what if segment is a retransmission?
  - Solution: don’t update RTT if segment was retransmitted

- Problem 2: RTT can have high variance
  - Initial implementation doesn’t account for this
  - Congestion control: modeling network load
Nagle’s algorithm

- Goal: reduce the overhead of small packets
  
  if (there is data to send) and (window >= MSS)
  
  Send a MSS segment

  else

  if there is unAcked data in flight
  
  buffer the new data until ACK arrives

  else

  send all the new data now

- Receiver should avoid advertising a window <= MSS after advertising a window of 0
Delayed Acknowledgments

• Goal: Piggy-back ACKs on data
  – Delay ACK for 200ms in case application sends data
  – If more data received, immediately ACK second segment
  – Note: never delay duplicate ACKs (if missing a segment)

• Warning: can interact badly with Nagle for some applications
  – Nagle waits for ACK until send => Temporary deadlock
  – App can disable Nagle with TCP_NODELAY
  – App should also avoid many small writes
Summary: flow control

• Flow control provides correctness: reliable, in order delivery
• Need more for performance
  – What if the network is the bottleneck?
• Sending too fast will cause queue overflows, heavy packet loss
• Need more for performance: congestion control
Connection Termination

• When you have no more data to send, send a FIN
  – Sent by close() or shutdown()

• Both sides close connection separately!

• TIME_WAIT: initiating side should wait for 2*MSL before deleting TCB
  – MSL = Longest time a segment might be delayed (configurable, ~1min)
TCP State Diagram

- **CLOSED**
  - **(Start)**
  - LISTEN
  - CONNECT/SYN
    - (Step 1 of the 3-way-handshake)
  - CLOSE/-

- **LISTEN**
  - SYN/SYN+ACK
    - (Step 2 of the 3-way-handshake)
  - CLOSE/-
  - SEND/SYN
    - (simultaneous open)
  - SYN+ACK/ACK
    - (Step 3 of the 3-way-handshake)

- **SYN RECEIVED**
  - SYN/SYN+ACK
    - (simultaneous open)
  - RST/-
  - ACK/-

- **SYN SENT**
  - SYN/SYN+ACK
    - (Step 2 of the 3-way-handshake)
  - FIN/ACK

- **ESTABLISHED**
  - Data exchange occurs
  - SYN+ACK/ACK
    - (Step 3 of the 3-way-handshake)
  - ACK/-

- **SENDING**
  - Data exchange occurs

- **ESTABLISHED**
  - Data exchange occurs

- **RECEIVING**
  - Data exchange occurs

- **CLOSED**
  - **(Go back to start)**
  - CONNECT/SYN
    - (Step 1 of the 3-way-handshake)
  - CLOSE/-

- **TIME WAIT**
  - Timeout

- **LAST ACK**
  - ACK/-

- **CLOSE WAIT**
  - CLOSE/FIN

- **FIN WAIT 2**
  - FIN+ACK/ACK

- **FIN WAIT 1**
  - FIN+ACK

- **CLOSING**
  - FIN+ACK/ACK

- **ACTIVE CLOSE**
  - Passive CLOSE

- **PASSIVE CLOSE**
  - Active CLOSE

- **ERROR**
  - unusual event
  - client/receiver path
  - server/sender path

- **TIME OUT**
  - Go back to start

- **SEND/ACK**
  - Data exchange occurs