CSCI-1680
Transport Layer IV

Data over TCP

Nick DeMarinis

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

- Sender wants to send “abcdef”
- Max segment size (MSS) = 1
- Receiver’s window = 4

**ADVERTISED WINDOW**: How much space left in receiver’s buffer
- Upper bound on how much can be sent.
What to do when window is full?

- Zero window moving
- Sender sends 1-byte segment periodically
- Receiver will ACK which will indicate if its window has changed.
"Hi, I'd like to hear a TCP joke."
"Hello, would you like to hear a TCP joke?"
"Yes, I'd like to hear a TCP joke."
"OK, I'll tell you a TCP joke."
"Ok, I will hear a TCP joke."
"Are you ready to hear a TCP joke?"
"Yes, I am ready to hear a TCP joke."
"Ok, I am about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline."
"Ok, I am ready to get your TCP joke that will last 10 seconds, has two characters, does not have an explicit setting, and ends with a punchline."
"I'm sorry, your connection has timed out. ... Hello, would you like to hear a TCP joke?"
Administrivia

- Sign up for TCP milestone II: on/before next Thursday
- HW4 (short): out today, related to what you need for TCP milestone II
- Grading is in progress...
Topics for today

• Connection termination
• More on flow control
• Motivation for congestion control
Connection Termination

- **Both sides must agree when done sending**
  - When you have no more data to send, send a FIN
    - Sent by close() or shutdown()
  - Both sides close connection separately!

**TIME-WAIT:** Sender must keep connection open for $2 \times AKL$ before declaring TCB closed

**TIME-WAIT:** Configurable (AIM)
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
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)

\[\text{CLOSED} \rightarrow \text{TIME_WAIT} \rightarrow \text{FIN_WAIT}_2 \rightarrow \text{FIN_WAIT}_1 \rightarrow \text{CLOSE_WAIT} \rightarrow \text{CLOSED}\]
Helpful sending mechanics
(used in modern TCPs, not required for project)
When to Transmit?

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
EXAMPLE:

TELNET
OLD
SSH

Is
is
Is

PROBLEM
LOTS
NETWORK
OVELHEAD
SMALL
Samonte
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)
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

⇒ Lots of "knobs" for DIFF APPLICATIONS.
Handling Retransmissions
How do ACKs work?

• ACK contains next expected sequence number
• Sender: if one segment is missed but new ones received, send duplicate ACK
How do ACKs work?

- ACK contains next expected sequence number
- Sender: if one segment is missed but new ones received, send duplicate ACK
- Receiver retransmits when:
  - Receive timeout (RTO) expires
  - Possibly other conditions, for certain TCP variants (eg. 3 dup 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: measure expected RTT based on ACKs received
  – Use exponentially weighted moving average (EWMA)
  – RFC793 version ("smoothed RTT"): 
When to time out?

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

- **Problem:** RTT can be highly variable

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

- **Values:**
  - \(\alpha = "\text{Smoothing factor}": 0.8-0.9\)
  - \(\beta = "\text{Delay variance factor}": 1.3-2.0\)
  - \(\text{RTO}_{\text{Min}} = 1 \text{ second}\)

*RFC793, Sec 3.7*

RFC6298 (slightly more complicated, also measures variance)
Using the RTO timer

Recommended by RFC6298

• Maintain ONE timer per connection
• When segment is sent => set timer to expire after \( t_{RTO} \)
• When ACK is received with new data, reset the timer

When the timer expires:

• Retransmit earliest unacknowledged segment
• \( RTO = 2 \times RTO \) (up to some max) \( \rightarrow \text{Exponential Backoff} \)
• If no data after \( N \) retransmissions => give up, terminate connection
Using the RTO timer

Recommended by RFC6298

• Maintain ONE timer per connection
• When segment is sent => set timer to expire after $t_{RTO}$
• When ACK is received with new data, reset the timer
extra content
we will
cover later

(feel free to
read ahead

though!)
This is only the beginning…

- Problem 1: what if ACK is for a retransmitted segment?
This is only the beginning...

- Problem 1: what if ACK is for a retransmitted segment?
  - Solution: don’t update RTT if segment was retransmitted
This is only the beginning...

• Problem 1: what if ACK is for a retransmitted segment?
  – Solution: don’t update RTT if segment was retransmitted

• Problem 2: RTT can have high variance
  – Initial implementation doesn’t account for this (modern version, RFC6298)
• Problem 1: what if ACK is for a retransmitted segment?
  – Solution: don’t update RTT if segment was retransmitted

• Problem 2: RTT can have high variance
  – Initial implementation doesn’t account for this (modern version, RFC6298)
  – Congestion control: modeling network load
Congestion control: motivation
The story so far

- Flow control: reliable, in-order delivery
- Goal: send as much data as receiver can handle
  - Receiver’s *advertised window*: sent with every ACK
- Sliding window: increase throughput by having multiple packets in flight
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
A Short History of TCP

- 1974: 3-way handshake
- 1978: IP and TCP split
- 1983: January 1st, ARPAnet switches to TCP/IP
- 1984: Nagle predicts congestion collapses
- 1986: Internet begins to suffer congestion collapses
  - LBL to Berkeley drops from 32Kbps to 40bps
- 1987/8: Van Jacobson fixes TCP, publishes seminal paper*: (TCP Tahoe)
- 1990: Fast transmit and fast recovery added
  (TCP Reno)

* Van Jacobson and Michael Karels. Congestion avoidance and control. SIGCOMM ’88
Congestion Collapse
Nagle, rfc896, 1984
• Mid 1980’s: Problem with the protocol implementations, not the protocol!
Congestion Collapse
Nagle, rfc896, 1984

• Mid 1980’s: Problem with the protocol implementations, not the protocol!
• What was happening?
• Mid 1980’s: Problem with the protocol implementations, not the protocol!

• What was happening?

• If close to capacity, and, e.g., a large flow arrives suddenly…
  – RTT estimates become too short
  – Lots of retransmissions \(\rightarrow\) increase in queue size
  – Eventually many drops happen (full queues)
  – Fraction of useful packets (not copies) decreases
The problem

• https://witestlab.poly.edu/respond/sites/genitutorial/files/tcp-aimd.ogv
TCP Congestion Control

• 3 Key Challenges
  – Determining the available capacity in the first place
  – Adjusting to changes in the available capacity
  – Sharing capacity between flows

• Idea
  – Each source determines network capacity for itself
  – Rate is determined by window size
  – Uses implicit feedback (drops, delay)
  – ACKs pace transmission (self-clocking)
Congestion control has a long history

• Active research area for ~40 years

• I am nowhere close to being an expert

• My hope is to get you to understand the problems involved
Just a few TCP implementations

What’s the difference?

General usage
- Reno (1980s)
- Tahoe
- Vegas
- New Vegas
- Westwood
- Cubic
- BBR (2016)
- …
Dealing with Congestion

• Maintain two windows:
  – Advertised Window (from receiver)
  – Congestion window (cwnd)

\[
\text{Sending rate} = \min(\text{Advertised Window}, \; \text{cwnd})
\]

• Ideally, want to have sending rate:  \( \sim \) Window/RTT
Dealing with Congestion

- Assume losses are due to congestion
- After a loss, reduce congestion window
  - How much to reduce?
- Idea: conservation of packets at equilibrium
  - Want to keep roughly the same number of packets in network
  - Analogy with water in fixed-size pipe
  - Put new packet into network when one exits
Next time

- TCP Tahoe/Reno
- Overview of other CC schemes