CSCI-1680
Transport Layer II
Data over TCP

Rodrigo Fonseca
• Introduction to TCP
  – Header format
  – Connection state diagram
• Today: sending data
First Goal

• We should not send more data than the receiver can take: *flow control*
• Data is sent in MSS-sized segments
  – Chosen to avoid fragmentation
• *Sender can delay sends to get larger segments*
• *When to send data?*
• *How much data to send?*
Flow Control

• Part of TCP specification (even before 1988)
• Goal: not sent more data than the receiver can handle
• Sliding window protocol
• Receiver uses window header field to tell sender how much space it has
Flow Control

- **Receiver**: $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected}-1) - \text{LastByteRead})$

- **Sender**: $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$

**EffectiveWindow** = $\text{AdvertisedWindow} - (\text{BytesInFlight})$

$\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
Flow Control

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

- **Sender keeps sending 1-byte segments until window comes back > 0**
When to Transmit?

• Nagle’s algorithm
• **Goal: reduce the overhead of small packets**
  
  If available data and window \( \geq \) MSS
  
  Send a MSS segment
  
  else
  
  If there is unAced data in flight
  
  buffer the new data until ACK arrives
  
  else
  
  send all the new data now

• **Receiver should avoid advertising a window \( \leq \) 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 *very* badly with Nagle
  – Temporary deadlock
  – Can disable Nagle with TCP_NODELAY
  – Application can also avoid many small writes
Limitations of Flow Control

- Network may be the bottleneck
- Signal from receiver not enough!
- Sending too fast will cause queue overflows, heavy packet loss
- Flow control provides correctness
- Need more for performance: congestion control
Second goal

• We should not send more data than the network can take: *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. Congestion avoidance and control. SIGCOMM ’88
Congestion Collapse
Nagle, rfc896, 1984

• Mid 1980’s. Problem with the protocol implementations, not the protocol!
• What was happening?
  – Load on the network → buffers at routers fill up → round trip time increases
• If close to capacity, and, e.g., a large flow arrives suddenly…
  – RTT estimates become too short
  – Lots of retransmissions → increase in queue size
  – Eventually many drops happen (full queues)
  – Fraction of useful packets (not copies) decreases
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)
Dealing with Congestion

• TCP keeps *congestion* and flow control windows
  – Max packets in flight is lesser of two
• **Sending rate: ~Window/RTT**
• The key here is how to set the congestion window to respond to congestion signals
Starting Up

• **Before TCP Tahoe**
  – On connection, nodes send full (rcv)window of packets
  – Retransmit packet immediately after its timer expires

• **Result:** *window-sized bursts of packets in network*
Bursts of Packets

Figure 3: Startup behavior of TCP without Slow-start

Each dot is a 512 data-byte packet. The x-axis is the time the packet was sent. The y-axis is the sequence number in the packet header. Thus a vertical array of dots indicate back-to-back packets and two dots with the same y but different x indicate a retransmit. ‘Desirable’ behavior on this graph would be a relatively smooth line of dots extending diagonally from the lower left to the upper right. The slope of this line would equal the available bandwidth. Nothing in this trace resembles desirable behavior.

The dashed line shows the 20 KBps bandwidth available for this connection. Only 35% of this bandwidth was used; the rest was wasted on retransmits. Almost everything is retransmitted at least once and data from 54 to 58 KB is sent five times.

First-hop gateway sees a burst of eight packets delivered at 200 times the path bandwidth. This burst of packets often puts the connection into a persistent failure mode of continuous retransmissions (figures 3 and 4).

Conservation equilibrium: round trip timing

Once data is flowing reliably, problems (2) and (3) should be addressed. Assuming that the protocol implementation is correct, (2) must represent a failure of sender’s retransmit timer. A good round trip time estimator, the core of the retransmit timer, is the single most...

Graph from Van Jacobson and Karels, 1988
Determining Initial Capacity

• **Question: how do we set w initially?**
  – Should start at 1MSS (to avoid overloading the network)
  – Could increase additively until we hit congestion
  – May be too slow on fast network

• **Start by doubling w each RTT**
  – Then will dump at most one extra window into network
  – This is called *slow start*

• **Slow start, this sounds quite fast!**
  – In contrast to initial algorithm: sender would dump entire flow control window at once
Startup behavior with Slow Start

![Graph showing startup behavior with Slow Start](image-url)
Slow start implementation

• Let \( w \) be the size of the window in bytes
  – We have \( w \)/MSS segments per RTT

• We are doubling \( w \) after each RTT
  – We receive \( w \)/MSS ACKs each RTT
  – So we can set \( w = w + \text{MSS} \) on every ACK

• At some point we hit the network limit.
  – Experience loss
  – We are at most one window size above the limit
  – Remember window size (ssthreah) and reduce window
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 network
  - Analogy with water in fixed-size pipe
  - Put new packet into network when one exits
How much to reduce window?

• **Crude model of the network**
  – Let $L_i$ be the load (# pkts) in the network at time $i$
  – If network uncongested, roughly constant $L_i = N$

• **What happens under congestion?**
  – Some fraction $\gamma$ of packets can’t exit the network
  – Now $L_i = N + \gamma L_{i-1}$, or $L_i \approx \gamma^i L_0$
  – Exponential increase in congestion (for $\gamma > 1$

• **Sources must decrease offered rate exponentially**
  – i.e, multiplicative decrease in window size
  – TCP chooses to cut window in half
How to use extra capacity?

• Network signals congestion, but says nothing of underutilization
  – Senders constantly try to send faster, see if it works
  – So, increase window if no losses… By how much?

• Multiplicative increase?
  – Easier to saturate the network than to recover
  – Too fast, will lead to saturation, wild fluctuations

• Additive increase?
  – Won’t saturate the network
  – Remember fairness (third challenge)?
Chiu Jain Phase Plots

Fair: \( A = B \)

Efficient: \( A + B = C \)

Goal: fair and efficient!
Chiu Jain Phase Plots

Fair: $A = B$

Efficient: $A + B = C$

Flow Rate B

Flow Rate A
Chiu Jain Phase Plots

Flow Rate A

Flow Rate B

Fair: A = B

Efficient: A + B = C

AIAD
Chiu Jain Phase Plots

- Fair: $A = B$
- Efficient: $A + B = C$

Diagram shows points clustering along the fair and efficient lines, labeled as AIMD.
AIMD Implementation

• **In practice, send MSS-sized segments**
  – Let window size in bytes be $w$ (a multiple of MSS)

• **Increase:**
  – After $w$ bytes ACKed, could set $w = w + \text{MSS}$
  – Smoother to increment on each ACK
    • $w = w + \text{MSS} \times \text{MSS}/w$
    • (receive $w$/MSS ACKs per RTT, increase by $\text{MSS}/(w/\text{MSS})$ for each)

• **Decrease:**
  – After a packet loss, $w = w/2$
  – But don’t want $w < \text{MSS}$
  – So react differently to multiple consecutive losses
  – Back off exponentially (pause with no packets in flight)
AIMD Trace

- AIMD produces sawtooth pattern of window size
  - Always probing available bandwidth
Putting it together

• TCP has two states: Slow Start (SS) and Congestion Avoidance (CA)

• A window size threshold governs the state transition
  – Window <= threshold: SS
  – Window > threshold: congestion avoidance

• States differ in how they respond to ACKs
  – Slow start: \( w = w + \text{MSS} \)
  – Congestion Avoidance: \( w = w + \frac{\text{MSS}^2}{w} \) (1 MSS per RTT)

• On loss event: set \( w = 1 \), slow start
How to Detect Loss

• **Timeout**
• **Any other way?**
  – Gap in sequence numbers at receiver
  – Receiver uses cumulative ACKs: drops => duplicate ACKs
• **3 Duplicate ACKs considered loss**
Putting it all together

\[
\begin{align*}
\text{cwnd} & \\
\text{Time} & \\
\text{Slow Start} & \\
\text{Timeout} & \\
\text{AIMD} & \\
\text{ssthresh} & \\
\text{Timeout} & \\
\text{AIMD} & \\
\text{Slow Start} & \\
\end{align*}
\]
RTT

• We want an estimate of RTT so we can know a packet was likely lost, and not just delayed

• **Key for correct operation**

• **Challenge**: RTT can be highly variable
  – Both at long and short time scales!

• **Both average and variance increase a lot with load**

• **Solution**
  – Use exponentially weighted moving average (EWMA)
  – Estimate deviation as well as expected value
  – Assume packet is lost when time is well beyond reasonable deviation
Originally

- \( \text{EstRTT} = (1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT} \)
- \( \text{Timeout} = 2 \times \text{EstRTT} \)
- **Problem 1:**
  - in case of retransmission, ack corresponds to which send?
  - Solution: only sample for segments with no retransmission
- **Problem 2:**
  - does not take variance into account: too aggressive when there is more load!
Jacobson/Karels Algorithm (Tahoe)

- EstRTT = $(1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT}$
  - Recommended $\alpha$ is 0.125
- DevRTT = $(1 - \beta) \times \text{DevRTT} + \beta |\text{SampleRTT} - \text{EstRTT}|$
  - Recommended $\beta$ is 0.25
- Timeout = EstRTT + 4 DevRTT
- For successive retransmissions: use exponential backoff
Old RTT Estimation

Figure 5: Performance of an RFC793 retransmit timer

Packet RTT (sec.)

0 10 20 30 40 50 60 70 80 90 100 110

0 2 4 6 8 10 12

Trace data showing per-packet round trip time on a well-behaved Arpanet connection. The x-axis is the packet number (packets were numbered sequentially, starting with one) and the y-axis is the elapsed time from the send of the packet to the sender's receipt of its ack. During this portion of the trace, no packets were dropped or retransmitted. The packets are indicated by a dot. A dashed line connects them to make the sequence easier to follow. The solid line shows the behavior of a retransmit timer computed according to the rules of RFC793.

The parameter $\tau$ accounts for RTT variation (see [5], section 5). The suggested $\tau = 2$ can adapt to loads of at most 30%. Above this point, a connection will respond to load increases by retransmitting packets that have only been delayed in transit. This forces the network to do useless work, wasting bandwidth on duplicates of packets that will eventually be delivered, at a time when it's known to be having trouble with useful work. I.e., this is the network equivalent of pouring gasoline on a fire.

We developed a cheap method for estimating variation (see appendix A) and the resulting retransmit timer essentially eliminates spurious retransmissions. A pleasant side effect of estimating $\tau$ rather than using a fixed value is that low load as well as high load performance improves, particularly over high delay paths such as satellite links (figures 5 and 6).

Another timer mistake is in the backoff after a retransmit: If a packet has to be retransmitted more than once, how should the retransmits be spaced? For a transport endpoint embedded in a network of unknown topology and with an unknown, unknowable and constantly changing population of competing conversations, only one scheme has any hope of working—exponential backoff—but a proof of this is beyond the scope of this paper.

We are far from the first to recognize that transport needs to estimate both mean and variation. See, for example, [6]. But we do think our estimator is simpler than most.

See [8]. Several authors have shown that backoffs 'slower' than exponential are stable given finite populations and knowledge of the global traffic. However, [17] shows that nothing slower than exponential behavior will work in the general case. To feed your intuition, consider that an IP gateway has essentially the same behavior as the 'ether' in an ALOHA net or Ethernet. Justifying exponential retransmit backoff is the same as
Figure 6: Performance of a Mean+Variance retransmit timer

To finesse a proof, note that a network is, to a very good approximation, a linear system. That is, it is composed of elements that behave like linear operators — integrators, delays, gain stages, etc. Linear system theory says that if a system is stable, the stability is exponential. This suggests that an unstable system (a network subject to random load shocks and prone to congestive collapse) can be stabilized by adding exponential damping (exponential timer backoff) to its primary excitation (senders, traffic sources).

If the timers are in good shape, it is possible to state with some confidence that a timeout indicates a lost packet and not a broken timer. At this point, something can be done about (3). Packets get lost for two reasons: they are damaged in transit, or the network is congested and somewhere on the path there was insufficient buffer capacity. On most network paths, loss due to damage is rare (≪1%) so it is probable that a packet loss is due to congestion in the network.

Because a packet loss empties the window, the throughput of any window flow control protocol is quite sensitive to damage loss. For an RFC793 standard TCP running with window $w$ (where $w$ is at most the bandwidth-delay product), a loss probability of $p$ degrades throughput by a factor of $(1 + 2pw)^{-1}$. E.g., a 1% damage loss rate on an Arpanet path (8 packet window) degrades TCP throughput by 14%.

The congestion control scheme we propose is insensitive to damage loss until the loss rate is on the order of the window equilibration length (the number of packets it takes the window to regain its original size after a loss). If the pre-loss size is $w$, equilibration takes roughly $w^2/3$ packets so, for the Arpanet, the loss sensitivity
Slow start every time?!

- Losses have large effect on throughput
- Fast Recovery (TCP Reno)
  - Same as TCP Tahoe on Timeout: \( w = 1 \), slow start
  - On triple duplicate ACKs: \( w = w/2 \)
  - Retransmit missing segment (fast retransmit)
  - Stay in Congestion Avoidance mode
Fast Recovery and Fast Retransmit

- Slow Start
- Fast retransmit

AI/MD
3 Challenges Revisited

- Determining the available capacity in the first place
  - Exponential increase in congestion window
- Adjusting to changes in the available capacity
  - Slow probing, AIMD
- Sharing capacity between flows
  - AIMD
- Detecting Congestion
  - Timeout based on RTT
  - Triple duplicate acknowledgments
- Fast retransmit/Fast recovery
  - Reduces slow starts, timeouts
Next Class

- More Congestion Control fun
- Cheating on TCP
- TCP on extreme conditions
- TCP Friendliness
- TCP Future