CMPE 150:
Introduction to Computer Networks

Dr. Chane L. Fullmer
chane@cse.ucsc.edu
Homework Assignments

Homework assignment #4

Chapter three

Due May 29 – THIS THURSDAY!!
Homework Assignments

Homework assignment #5

Chapter two

Due June 5th – only one+ short week from today
(Optional) Class Project

- Network programming project
  - In lieu of taking final examination
  - Or, just a wildcard – for mid-term or final
- Goal:
  - Build an FTP client/server from scratch
  - Using ‘C’ language
- Details on web page.. now...

Due by next Wednesday (June 4th)
Class Final Exam

- Final exam
  - Three hours
  - 40 – 50 questions -- comprehensive
  - Multiple choice as the midterm
  - Scantron – bring your sheet and pencils

- **Wednesday – June 11th** 8:00 – 11:00 AM
  - Two weeks from tomorrow.. Tick, tick, tick...
CMPE 150: Introduction to Computer Networks
Set 16:
MORE -- End-to-End Transmission Control (TCP and UDP)
Flow Control vs. Congestion Control

- Congestion control
  - **Global issue**: concerns all routers and hosts on path from Source to Destination
  - make sure every subnet can handle the traffic
Flow Control vs. Congestion Control

- Flow Control
  - Involves two endpoints
  - Make sure sender doesn’t transmit faster than receiver can absorb packets

Diagram:

- Server 1Gbps
- File transfer
- PC 1Mbps
TCP Flow Control

- Transmission Window a.k.a. congestion window (\(cwnd\))
  - Sliding window
  - maintained by sender
  - conservation of packets
- Receiver’s window set through socket API
  - controlled by the receiver
  - advertised to the sender in field of TCP header (RAW)
Data Flow

- Conservation of packets
  - Inject new packets at the rate ACKs are returned by receiver
- New window: $cwnd$
  - initially $cwnd = 1$ segment
- **Sender’s window** = min($cwnd$, RAW)
TCP Congestion Control
Slow Start

- Due to Van Jacobson (SIGCOMM 88)
- Algorithm used at beginning of connection and after a timeout
- leads to **exponential growth** in the amount of outstanding data in network
- *cwnd* doubles **every** RTT !!
- Algorithm:
  - When an ACK is received before timeout:
    - \( cwnd = cwnd + 1 \) for each ack’d segment
TCP Congestion Control

- Slow start continues until BWDP is exceeded, then:
  - Routers drop packets -- losses occur
  - Need to stop exponential increase!
- Use *congestion avoidance* to deal with lost packets!
  - Slow down the transmission rate
  - Provide for **linear** increase of the transmission window
- In practice, *congestion avoidance* implemented together with **slow start**
Congestion Avoidance

CA is flow control imposed by the sender:

- Introduce a new variable, **ssthresh**
  - initialized to 65,535 (max. window)
- If timeout, set ssthresh = cwnd/2 and cwnd = 1
- Re-enter slow start, until cwnd = ssthresh
- When cwnd = ssthresh, then grow cwnd linearly until it reaches RAW:
  - ACK is received (before a timeout)
    then \( cwnd = cwnd + 1/cwnd \)
    - Hence, \( cwnd \) increases by 1 segment every RTT
Putting it together: Slow start and congestion avoidance

The algorithm:
- If $cwnd < ssthresh$
  - do slow start
- Else if $cwnd \geq ssthresh$
  - do congestion avoidance
Graph of CA and SS

Figure 21.8  Visualization of slow start and congestion avoidance.
TCP Congestion Control: Underdamped Feedback System!
Problems with CA and SS

- Slow start is an attempt to discover the network bandwidth
  - Discovery proceeds by filling network queues in intermediate routers.
  - Once queues are full, routers drop packets.
  - Once loss discovered, it’s too late!
  - TCP sender reduces window when loss is discovered
- Queue level oscillates between full and $cwnd/2$
  - What sort of problems does this introduce??
Fast Retransmit and Fast Recovery

- When third duplicate ACK is detected:
  - Retransmit missing segment
  - Set $ssthresh = \frac{cwnd}{2}$
  - Set $cwnd = ssthresh + 3$

- For each additional duplicate ACK:
  - increment $cwnd$ by one segment
  - transmit a new packet if possible
  - why? With each ACK, we know a data pkt left the network

- When new ACK arrives:
  - $cwnd = ssthresh$
**Nagle’s Algorithm**

- Problem with interactive applications such as Telnet or Rlogin ⇒ produce *tinygrams*
- Normally each keystroke generates a packet
  - 40 bytes overhead for one byte of data is unacceptable!
  - Plus, server echoes the data!

---

![Diagram showing the interaction between client and server with keystrokes, data bytes, acknowledgments, and echoes.]
Nagle’s Algorithm

- Solution proposed in RFC 896 in 1984
  - For *Silly Window Syndrome*

- Solution:
  - only one outstanding small segment is allowed
  - no additional small segments can be sent until the previous is ACKed
  - Once the ACK arrives, several small segments will have been buffered and can now be transmitted
  - Solution is self-clocking based on ACK arrival

- Sometimes Nagle’s alg. Should be turned off
  - X window system server
    - need to send mouse movements for real-time applications
Improvements to TCP

- **SACK**: include information in the ACK which indicates missing packets in the window
- **Vegas**: use rate control instead of arrival of ACKs to pace data into network
- **Santa Cruz**: use relative delay over forward path to anticipate queue buildup
TCP-SACK

- **Goal:** Improve TCP error recovery mechanism
- Selectively acknowledge lost data within the transmission window
- Uses sequence number ranges
  - example: $\text{ACK} = 1000, \text{SACK} = 1040:1080$
- Limited by max. size of TCP header to 3 distinct ranges
- Important when there are multiple losses per window
  - multiple losses often results in a timeout
- Significance performance improvements in wired networks
- Does this approach improve congestion control?
TCP Vegas

- Brakmo and O’Malley, Sigcomm’96
- Two main parts:
  - new and early retransmission mechanism
  - new congestion policy
- Improved RTT estimate:
  - sender records time for every packet sent
  - calculate RTT for every ACK received
Vegas - Retransmission Policy

- When a duplicate ACK is received:
  - if \((current\ time - timestamp ) > RTO\)
    - retransmit segment!
      - Note earlier retransmission than Reno
  - if elapsed time is not large enough
    - check again for 2nd duplicate ACK
    - finally retransmit on 3rd duplicate ACK

- Allows for faster retransmissions when duplicates arrive
Vegas - Retransmission Policy (cont.)

- When nonduplicate ACK arrives:
  - if 1st or 2nd after a retransmission
    - check timestamp on all outstanding packets
    - retransmit if necessary

- When retransmitted segment is ACKed
  - cut congestion window by 1/4
  - (Reno cuts in half)
  - slightly more aggressive policy
  - window reduced only once for each loss interval
Vegas - Congestion Avoidance

- Vegas attempts to be proactive to congestion
- Tries to avoid losses by preventing congestion
- Uses a rate-based approach instead of window-based
- Keeps transmission rate within a calculated range
Vegas - Important terms for CA

- **Base RTT**
  - RTT of a segment in uncongested network
  - minimum RTT ever observed
  - generally first RTT measurement taken
    - window small at beginning of connection

- **Expected Throughput**
  - cwnd/BaseRTT

- **Actual Throughput**
  - Time one segment
  - Note # of packets transmitted during 1 RTT interval
  - Actual Throughput = #packets/RTT
Vegas - CA algorithm

- Basic idea: Try to keep the *expected throughput* close to the *ideal throughput*
- Let \( \text{Diff} = \text{Expected} - \text{Actual} \)
- **Rules for window adjustment:**
  - \( \text{Diff} < \alpha \), increase linearly
  - \( \text{Diff} > \beta \), decrease linearly
  - \( \alpha < \text{Diff} < \beta \), hold window steady
    - Where \( \alpha \) & \( \beta \) are expressed in KB/s (ie, 10 KB/s and 30KB/s) less than the Expected Rate.
Vegas - CA Algorithm

- Region A: increase window ▲ not pushing hard enough
- Region B: keep window constant
- Region C: decrease window ▲ pushing system too hard

Figure 3: Congestion avoidance mechanism for TCP Vegas
Vegas - Modified Slow Start

- Slows down Reno’s slow start mechanism!
- Double window every other RTT
  - Keep window fixed during 2nd interval
- Once rate falls below $\alpha$, transition from slow start to congestion avoidance
Vegas - Modified Slow Start
Berkeley Sockets

- Set of transport-level primitives made available by Berkeley UNIX.

- Server side:
  - SOCKET: create new communication end point.
  - BIND: attach local address to socket (once server binds address, clients can connect to it).
  - LISTEN: listen for connection.
  - ACCEPT: accept new connection.
  - SEND, RECEIVE: send and receive data.
  - CLOSE: release connection.
Berkeley Sockets

- Client side:
  - SOCKET: create socket.
  - CONNECT: try to establish connection.
  - SEND, RECEIVE: send and receive data.
  - CLOSE: release connection.
That's All For Today
Slow Start, CA and Fast Recovery