COMP/ELEC 429/556
Introduction to Computer Networks

The TCP Protocol

Some slides used with permissions from Edward W. Knightly, T. S. Eugene Ng, Ion Stoica, Hui Zhang
Transport Layer

- Purpose 1: Demultiplexing of data streams to different application processes
- Purpose 2: Provide value-added services that many applications want
  - Recall network layer in Internet provides a “Best-effort” service only, transport layer can add value to that
    - Application may want reliability, etc
  - No need to reinvent the wheel each time you write a new application
A very simple transport protocol: User Datagram Protocol (UDP)

- Connectionless datagram
  - Socket: SOCK_DGRAM
- Port number used for demultiplexing
  - port numbers = connection/application endpoint
- Adds end-to-end error checking through optional checksum
  - some protection against data corruption errors between source and destination (links, switches/routers, bus)
  - does not protect against packet loss, duplication or reordering
Using Transport Layer Port Number to Demultiplex Traffic

![Diagram showing transport layer multiplexing and demultiplexing using port numbers.](image)
Usages of UDP

• Custom protocols/applications can be implemented on top of UDP
  – use the port addressing provided by UDP
  – implement specialized reliability, flow control, ordering, congestion control as the app sees fit

• Examples:
  – remote procedure call
  – multimedia streaming (real time protocol)
  – cluster computing communication libraries
Transmission Control Protocol (TCP)

- Reliable bidirectional in-order byte stream
  - Socket: SOCK_STREAM
- Connections established & torn down
- Multiplexing/ demultiplexing
  - Ports at both ends
- Error control
  - Users see correct, ordered byte sequences
- End-to-end flow control
  - Avoid overwhelming receiver at each end
- Congestion control
  - Avoid creating traffic jams within network
Connection Setup

• Why need connection setup?

• Mainly to agree on starting sequence numbers
  – Starting sequence number is randomly chosen
  – Reason: to reduce the chance that sequence numbers of old and new connections from overlapping
Important TCP Flags

• SYN: Synchronize
  – Used when setting up connection

• FIN: Finish
  – Used when tearing down connection

• ACK
  – Acknowledging received data
Establishing Connection

• Three-Way Handshake
  – Each side notifies other of starting sequence number it will use for sending
  – Each side acknowledges other’s sequence number
    • SYN-ACK: Acknowledge sequence number + 1
  – Can combine second SYN with first ACK
Tearing Down Connection

• Either Side Can Initiate Tear Down
  – Send FIN signal
  – “I’m not going to send any more data”

• Other Side Can Continue Sending Data
  – Half open connection
  – Must continue to acknowledge

• Acknowledging FIN
  – Acknowledge last sequence number + 1
Sequence Number Space

- Each byte in byte stream is numbered
  - 32 bit value
  - Wraps around
  - Initial values selected at start up time

- TCP breaks up the byte stream into segments
  - Each segment transmitted by a packet
  - Limited by the Maximum Segment Size
  - Set to prevent packet fragmentation

- Each segment has a sequence number
  - Indicates where it fits in the byte stream
Bidirectional Communication

- Each Side of Connection can Send and Receive
- What this Means
  - Maintain different sequence numbers for each direction
  - A single packet can contain new data for one direction, plus acknowledgement for the other, may also contain only data or only acknowledgement
  - Called duplicate ACKs
  - There are TCP variants that don’t do this – beyond our scope
Sequence Numbers

• 32 Bits, Unsigned

• Why So Big?
  – For sliding window, must have
    \(|\text{Sequence Space}| > 2^* |\text{Sending Window}|\)
    • Sending window size of basic TCP is at most \(2^{16}\) bytes
    • \(2^{32} > 2^{16};\) no problem
  – Also, want to guard against stray packets
    • With IP, assume packets have maximum segment lifetime (MSL) of 120s
      – i.e. can linger in network for upto 120s
    • Sequence number would wrap around in this time at 286Mbps
Error Control

• Checksum provides some end-to-end error protection
• Sequence numbers detect packet sequencing problems:
  – Duplicate: ignore
  – Reordered: reorder or drop
  – Lost: retransmit
• Lost segments retransmitted by sender
  – Use time out to detect lack of acknowledgment
  – Need estimate of the roundtrip time to set timeout
• Retransmission requires that sender keep copy of the data
  – Copy is discarded when ack is received
TCP Must Operate Over Any Internet Path

- Retransmission time-out should be set based on round-trip delay
- But round-trip delay different for each path!
- Must estimate RTT dynamically
Setting Retransmission Timeout (RTO)

- Time between sending & resending segment

**Challenge**
- Too long: Add latency to communication when packets dropped
- Too short: Send too many duplicate packets
- General principle: Must be > 1 Round Trip Time (RTT)
Round-trip Time Estimation

- Every Data/Ack pair gives new RTT estimate
- Can Get Lots of Short-Term Fluctuations
  - How to address this problem?
Exponential Smoothing Technique

• Round trip times estimated as a moving average:
  – Smoothed RTT = $\alpha$ (Smoothed RTT) + (1 - $\alpha$) (new RTT sample)
  – Recommended value for $\alpha$: 0.8 - 0.9
• Retransmission timeout (RTO) is a function of
  – Smoothed RTT (SRTT)
  – RTT variation (RTTVAR)
• RTO = SRTT + 4 * RTTVAR
  – Details in RFC 6298
RTT Sample Ambiguity

- Ignore sample for segment that has been retransmitted
TCP Speed Control

- Sliding window protocol
  - For window size $n$, can send up to $n$ bytes without receiving new acknowledgement
  - Speed proportional to $n/RTT$
  - When the data are acknowledged then the window slides forward

- Send window size set to minimum (advertised window, congestion window)
Window Flow Control: Send Side

- **Sent and acked**
- **Sent but not acked**
- **Not yet sent**

**Next to be sent**
**Must retain for possible retransmission**

window
Window Flow Control: Receive Side

- TCP receiver can delete acknowledged data only after the data has been delivered to the application.

- So, depending on how fast the application is reading the data, the receiver’s window size may change!!!
Solution

• Receiver tells sender the current advertised window size in every packet it transmits to the sender
• Sender uses this current advertised window size as an upper bound
  – send window size = minimum (advertised window, congestion window)

• Advertised window size is continuously changing
• Can go to zero!
  – Sender not allowed to send anything!
Setting Congestion Window

Send window size = minimum (advertised window, congestion window)

Phases of TCP congestion control
1. Slow start (getting to equilibrium)
   – Want to find this very very fast and not waste time
2. Congestion Avoidance
   – Additive increase - gradually probing for additional bandwidth
   – Multiplicative decrease – decreasing congestion window upon loss/timeout
Variables Used in Implementation

• **Congestion Window** *(cwnd)*
  Initial value is 1 MSS (=maximum segment size) counted as bytes

• **Actual sender window size used by TCP** = minimum *(advertised win, cwnd)*

• **Slow-start threshold Value** *(ss_thresh)*
  Initial value is the advertised window size
  
  • **slow start** *(cwnd < ssthresh)*
  • **congestion avoidance** *(cwnd >= ssthresh)*
TCP: Slow Start

• Goal: discover roughly the proper sending rate quickly

• Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
  • Initialize $cwnd = 1$ MSS (max segment size)
  • Each time a segment is acknowledged, increment $cwnd$ by one MSS ($cwnd++$).

• Continue until
  – Reach ss_thresh or
  – Packet loss
Slow Start Illustration

• The congestion window size grows very rapidly

• Observe:
  – Each ACK generates two packets
  – slow start increases rate exponentially fast (doubled every RTT)!
Congestion Avoidance

• Slow Start figures out roughly the rate at which the network starts getting congested
• Congestion Avoidance continues to react to network condition
  – Probes for more bandwidth, increase cwnd if more bandwidth available
  – If congestion detected, aggressive cut back cwnd
Congestion Avoidance: Additive Increase

- Slowly increase cwnd to probe for additional available bandwidth

- If $cwnd \geq ss\_thresh$ then
  each time a segment is newly acknowledged
  $cwnd += 1/cwnd$

- $cwnd$ is increased by one MSS only if all segments in the window have been acknowledged
  - Increases by 1 per RTT
Example of Slow Start + Congestion Avoidance

Assume that $ss_{thresh} = 8$
Detecting Congestion via Timeout

• If there is a packet loss, the ACK for that packet will not be received

• The packet will eventually timeout
  – No ack is seen as a sign of congestion
Congestion Avoidance: Multiplicative Decrease

• Each time when timeout occurs
  – ss_thresh is set to half the current size of the congestion window:
    \[ ss\text{-}thresh = \frac{cwnd}{2} \]
  – cwnd is reset to one:
    \[ cwnd = 1 \]
  – and slow-start is entered
Packet Pacing

- ACKs “self-pace” the data to avoid a burst of packets to be sent
- Observe: received ACK spacing ≈ bottleneck bandwidth
TCP (Tahoe variant) Illustration

- Slow Start
- Timeout
- Congestion Avoidance
- Timeout
- ss_thresh
Many Variants of TCP

• Common variants of TCP
  – TCP Tahoe - the basic algorithm (discussed previously)
  – TCP Reno - Tahoe + fast retransmit & fast recovery
    • Many end hosts today implement TCP Reno

• and many more:
  – TCP Vegas (use timing of ACKs to avoid loss)
  – TCP SACK (selective ACK)
TCP Reno

• Problem with Tahoe: If a segment is lost, there is a long wait until timeout
• Reno adds a **fast retransmit** and **fast recovery mechanisms**

• Upon receiving 3 duplicate ACKs, retransmit the presumed lost segment (**“fast retransmit”**)  
• But do not enter slow-start. Instead enter congestion avoidance (**“fast recovery”**)
Fast Retransmit

- Resend a segment after 3 duplicate ACKs
  - remember a duplicate ACK means that an out-of-sequence segment was received
  - ACK-n means packets 1, ..., n-1 all received

- Notes:
  - duplicate ACKs can be due to packet reordering!
  - if window is small, may not get 3 duplicate ACKs!
Fast Recovery

• After a fast-retransmit
  – cwnd = cwnd/2
  – ss_thresh = cwnd
  – i.e. starts congestion avoidance at new cwnd

• After a timeout
  – Same as TCP Tahoe
  – ss_thresh = cwnd/2
  – cwnd = 1
  – Do slow start
Fast Retransmit and Fast Recovery

- Retransmit after 3 duplicate ACKs
  - prevent expensive timeouts
- Slow start only once per session (if no timeouts)
- In steady state, $cwnd$ oscillates around the ideal window size.
TCP Reno Summary

• Slow-Start if cwnd < ss_thresh
  – cwnd++ upon every new ACK (exponential growth)
  – Timeout: ss_thresh = cwnd/2 and cwnd = 1

• Congestion avoidance if cwnd >= ss_thresh
  – Additive Increase Multiplicative Decrease (AIMD)
  – ACK: cwnd = cwnd + 1/cwnd
  – Timeout: ss_thresh = cwnd/2 and cwnd = 1

• Fast Retransmit & Recovery
  – 3 duplicate ACKs (interpret as packet loss)
  – Retransmit lost packet
  – cwnd=cwnd/2, ss_thresh = cwnd
TCP Reno Saw Tooth Behavior

- Initial Slow start
- Slow start to pace packets
- Fast Retransmit and Recovery
- Timeouts may still occur

Congestion Window

Time