18-345: Introduction to Telecommunication Networks
Lectures 14: Transport Protocols

Peter Steenkiste

Spring 2015
www.cs.cmu.edu/~prs/nets-ece

Outline

• Transport introduction
• Error recovery and flow control
• Congestion control
• Transport optimization and futures

Transport Protocols

• Lowest level end-to-end protocol.
  – Header generated by sender is interpreted only by the destination
  – Routers view transport header as part of the payload

Functionality Split

• Network provides best-effort delivery only
• End-systems must implement many functions
  – Demultiplexing
  – Error detection
  – Error recovery
  – In-order delivery
  – Message boundaries
  – Connection abstraction
  – Congestion control
  – …
UDP: User Datagram Protocol
[ RFC 768 ]

- "No frills," "bare bones" Internet transport protocol
- Demultiplexing based on ports
- Optional checksum
  - One's complement add (weak)
- That's it!
- So why do we need UDP?
  - No connections: no delay, state
    - Remember DNS?
  - No congestion control: can lead to unpredictable delays
    - Problem for multimedia, games, ...
  - Good starting point for other transport protocols
    - Implemented at application level

High-Level TCP Characteristics

- Protocol implemented entirely at the ends
  - Fate sharing
- Protocol has evolved over time
  - Nearly impossible to change the header
  - Change processing at endpoints
  - Use options to add information to the header
    - These do change sometimes
  - Backward compatibility is what makes it TCP
- Most changes related to:
  - Faster networks, efficiency
  - Congestion control

Evolution of TCP

- 1974 TCP described by Vint Cerf and Bob Kahn in IEEE Trans Comm
- 1975 Three-way handshake
- 1982 TCP & IP
- 1983 BSD Unix 4.2 supports TCP/IP
- 1984 Nagel's algorithm to reduce overhead of small packets, predicts congestion collapse
- 1985 TCP & IP RFC 791 & 793
- 1986 Congestion collapse observed
- 1987 Kahn's algorithm to better estimate round-trip time
- 1988 Van Jacobson's algorithm to improve congestion avoidance and congestion control
  - Most implemented in 4.3BSD Tahoe
- 1990 4.3BSD Reno fast retransmit delayed ACK's

TCP Through the 1990s

- 1992 ECN (Floyd) Explicit Congestion Notification
- 1992 TCP Vegas (Brakmo et al) delay-based congestion avoidance
- 1994 T/TCP (Braden) Transaction TCP
- 1996 SACK TCP (Floyd et al) Selective Acknowledgement
- 1998 SACK TCP (Floyd et al) Selective Acknowledgement
TCP Through the 2000s

- 2004: NewReno (Floyd et al.) Partial ACK in Fast Recovery
- 2007: CUBIC (Rhee, Xu, Ha Convex-Concave Response Fn.)
- 2008: Data Center TCP (too many authors) ECN, proportional window scaling
- 2011: Multi-Path TCP (Barré, Bonaventure) TCP over multiple subflows

TCP and its Header

- The cadillac of transport protocols
- Demultiplexing
- Connections
  - Sequence numbers
- Reliable
  - Acks, checksum
- Flow control
  - Window
- Congestion control
  - Nothing?
- Bookkeeping ++

Outline

- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

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 packets.
  - Packet size is limited to the Maximum Segment Size
- Each packet has a sequence number.
  - Indicates where it fits in the byte stream

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequence number</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>
Establishing Connection: Three-Way handshake

- Each side notifies other of starting sequence number it will use for sending
  - Why not simply chose 0?
    - Must avoid overlap with earlier incarnation
    - Security issues
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- Can combine second SYN with first ACK

TCP Connection Setup Example

- Client SYN
  - SeqC: Seq. 4019802004, window 65535, max. seg. 1260
- Server SYN-ACK+SYN
  - Receive: #4019802005 (= SeqC+1)
  - SeqS: Seq. 3428951569, window 5840, max. seg. 1460
- Client SYN-ACK
  - Receive: #3428951570 (= SeqS+1)

TCP State Diagram: Connection Setup

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
TCP Connection Teardown

Example

Session
- Echo client on 128.2.222.198, server on 128.2.210.194
- Client FIN
- SeqC: 1489294581
- Server ACK + FIN
- Ack: 1489294582 (= SeqC+1)
- SeqS: 1909787689
- Client ACK
- Ack: 1909787690 (= SeqS+1)

TCP State Diagram:
Connection Teardown

Outline

- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

Review: Stop and Wait

- ARQ
  - Receiver sends acknowledgement (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period
- Simplest ARQ protocol
- Send a packet, stop and wait until ACK arrives
Problems with Stop and Wait

- Stop and wait offers provides flow and error control, but..
- How do we overcome the limitation of one packet per roundtrip time: Sliding window.
  - Receiver advertises a “window” of buffer space
  - Sender can fill the window -> fills the “pipe”
- How do we distinguish new and duplicate packets: Sequence numbers
  - 1 bit enough for stop and wait
  - More bits for larger windows (see datalink lecture)

Bandwidth-Delay Product

\[ \text{Max Throughput} = \frac{\text{Window Size}}{\text{Roundtrip Time}} \]

Sliding Window

Sender/Receiver State

- On reception of new ACK (i.e. ACK for something that was not acked earlier)
  - Increase sequence of max ACK received
  - Send next packet
- On reception of new in-order data packet (next expected)
  - Hand packet to application
  - Send an ACK that acknowledges the paper
  - Increase sequence of max acceptable packet
- But what do we do if packets are lost or reordered?
  - Results in a gap in the sequence of received packets
  - Raises two questions
    - What feedback does receiver give to the sender, and how?
    - How and when does the sender retransmit packets
ACKing Strategies

• ACKs acknowledge exactly one packet
  – Simple solution, but bookkeeping on sender is a bit messy
    • Must keep per packet state – not too bad
  – Inefficient: need ACK packet for every data packet
• Cumulative acks acknowledge all packets up to a specific packet
  – Maybe not as intuitive, but simple to implement
  – Stalls the pipe until lost packet is retransmitted and ACKed
• Negative ACKs allow a receiver to ask for a packet that is (presumed to be) lost
  – Avoids the delay associated with a timeout

Selective Repeat

• Receiver individually acknowledges correctly received packets
  – If packets out of order, receiver cannot hand data to application so window does not move forward
• Sender only resends packets for which ACK not received
  – Sender timer for individual unACKed packet
• Sender window calculation
  – N consecutive seq #’s
  – Starts with an earliest unacknowledged packet
    • Some packets in the window may have been acknowledged

Selective Repeat: Sender, Receiver Windows

Go-Back-N Recovery

• Receiver sends cumulative ACKs
  – When out of order packet - send nothing (wait for source to timeout)
  – Otherwise sends cumulative ACK
• Sender implements Go-Back-N recovery
  – Set timer upon transmission of packet
  – Retransmit all unacknowledged packets on timeout
• Performance during loss recovery
  – No longer have an entire window in transit
  – Can have much more clever loss recovery
    • Receiver can send cumulative ACK even for out of order packets - Why?
Basic Go-Back-N in Action

Outline

- Transport introduction
- Error recovery and flow control
  - Connection establishment
  - Review stop-and-wait and friends
  - ACK and retransmission strategies
  - Making things work (well) in TCP
  - Timeouts
- Congestion control
- Transport optimization and futures

TCP = Go-Back-N Variant

- Sliding window with cumulative acks
  - Receiver can only return a single “ack” sequence number to the sender.
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
  - Duplicate acks sent when out-of-order packet received
- But: sender only retransmits a single packet.
  - Reason???
    - Only one that it knows is lost
    - Network is congested \( \rightarrow \) shouldn’t overload it
- Error control is based on byte sequences, not packets.
  - Retransmitted packet can be different from the original lost packet — Why?
**Duplicate ACKs (Fast Retransmit)**

- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
  - Window update – advertisement of new flow control window
- Assume re-ordering is infrequent and not of large magnitude
  - Receipt of 3 or more duplicate acks viewed as sign of a loss
  - Don’t wait for timeout to retransmit packet
  - When does this fail?

**How about Multiple Losses?**

- Basic problem is that cumulative acks provide little information
- Selective acknowledgement (SACK) essentially adds a bitmask of packets received
  - Implemented as a TCP option
  - Encoded as a set of received byte ranges (max of 4 ranges/often max of 3)
- When to retransmit?
  - Still need to deal with reordering → wait for out of order by 3 pkts

**SACK**
Outline

- Transport introduction
  - Error recovery and flow control
    - Connection establishment
    - Review stop-and-wait and friends
    - ACK and retransmission strategies
    - Making things work (well) in TCP
    - Timeouts
  - Congestion control
  - Transport optimization and futures

Round-trip Time Estimation

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT estimate
    - unneeded retransmissions
  - High RTT estimate
    - poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- Spurious timeouts
  - “Conservation of packets” principle – never more than a window worth of packets in flight
  - Most timeouts set using coarse clock, e.g., 500 msec

Original TCP Round-trip Estimator

- Round trip times exponentially averaged:
  - New RTT = \( \alpha \) (old RTT) + (1 - \( \alpha \)) (new sample)
  - Recommended value for \( \alpha \): 0.8 - 0.9
    - 0.875 for most TCP’s
- Retransmit timer set to \((b \times RTT)\), where \(b = 2\)
  - Every time timer expires, RTO exponentially backed-off
- Not good at preventing spurious timeouts
  - Why?
Jacobson’s Retransmission Timeout

• Key observation:
  – At high loads, round trip variance is high
• Solution:
  – Base RTO on RTT and standard deviation
    • $RTO = RTT + 4 \times rttvar$
  – new_rttvar = $\beta \times$ dev + $(1 - \beta) \times$ old_rttvar
    • Dev = linear deviation
    • Inappropriately named – actually smoothed linear deviation

Important Lessons

• Transport service
  – UDP → mostly just IP service
  – TCP → congestion controlled, reliable, byte stream
• Types of ARQ protocols
  – Sliding window for high throughput
  – Go-back-n → can keep link utilized (except w/ losses)
  – Selective repeat → efficient loss recovery
• TCP uses go-back-n variant
  – Avoid unnecessary retransmission ..
  – … and gaps in the flow (fast retransmit/recovery, SACK)