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

Source port # Dest port #
32 bits
<table>
<thead>
<tr>
<th>Length</th>
<th>Checksum</th>
</tr>
</thead>
</table>

Application data (message)

UDP segment format

High-Level TCP Characteristics

- Protocol implemented entirely on endpoints
  - 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
  - Backward compatibility is what makes it TCP
- Most changes related to:
  - Faster networks, efficiency
  - Congestion control

Evolution of TCP

1973
Three-way handshake
Raymond Tomkinson In SIGCOM #75

1974
TCP described by Vint Cerf and Bob Kahn In EEE Trans Comm

1983
BSD Unix 4.2 supports TCP/IP

1985
Congestion collapse observed

1986
Van Jacobson’s algorithms congestion avoidance and congestion control (most implemented in 4.3BSD Tahoe)

1990
4.3BSD Reno had retransmit delayed ACK’s

TCP Through the 1990s

1994
TCP Vegas (Brakmo et al) delay-based congestion avoidance

1994
ECN (Floyd) Explicit Congestion Notification

1996
How New Reno startup and loss recovery

1996
FACK TCP (Mathis et al) extension to SACK
TCP Through the 2000s

- 2000: TCP and its Header
- 2004: NewReno (Floyd et al.)
- 2008: Data Center TCP (too many authors)
- 2010: CUBIC
- 2011: Multi-Path TCP

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
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- 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
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 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 State Diagram: Connection Teardown
Outline

- Transport introduction
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- Congestion control
- Transport optimization and futures

A Naïve Protocol

- Sender simply sends to the receiver whenever it has packets.
- Potential problem: sender can outrun the receiver.
  - Receiver too slow, runs out of buffer space, ..
  - Even worse: packets can get lost!

Stop and Wait

- Send a packet, stop and wait until ACK arrives
  - Receiver sends acknowledgement (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period
  - Simplest “Automatic Repeat reQuest” protocol

Recovering from Error

- Time
- Packet lost
- Ack lost
- Early timeout
How to Recognize Retransmissions?

- Use sequence numbers
  - both packets and acks
- Sequence # in packet is finite
  → How big should it be?
  - For stop and wait?
- One bit – won’t send seq #1 until received ACK for seq #0

Window Flow Control

- Stop and wait flow control offers flow and error control, but it results in poor throughput for long-delay paths
- Solution: receiver provides sender with a window that it can fill with packets.
  - The window is backed up by buffer space on receiver
  - Receiver acknowledges a packet every time a packet is consumed and a buffer is freed
  - Need larger sequence numbers: \( W_s = 2^m - 1 \) window for \( m \) bits

Bandwidth-Delay Product

Max Throughput = \( \frac{\text{Window Size}}{\text{Roundtrip Time}} \)

Sliding Window Sender/Receiver State

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max ACK received</td>
<td>Next expected</td>
</tr>
<tr>
<td>Next sequence</td>
<td>Max acceptable</td>
</tr>
<tr>
<td>Sent &amp; Acked</td>
<td>Received &amp; Acked</td>
</tr>
<tr>
<td>OK to Send</td>
<td>Acceptable Packet</td>
</tr>
<tr>
<td>Sent Not Acked</td>
<td>Not Usable</td>
</tr>
<tr>
<td>Not Usable</td>
<td>Sent Not Acked</td>
</tr>
</tbody>
</table>

Receiver window

Sender window

Max acceptable

Received & Acked

Acceptable Packet

Not Usable

Not Usable
Window Sliding – Common Case

- On reception of new ACK
  - Increase max ACK received and send next packet
- On reception of new in-order data packet
  - 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?
    - How and when does the sender retransmit packets

ACKing Strategies

- Per-packet 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 sequence number
  - 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 (presumed to be) lost packet
  - Avoids the delay of a timeout but is not sufficient!

Selective Repeat Retransmissions

- Simple retransmission strategy for when receiver acknowledges correctly received packets individually
  - 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
  - Buffer space used at receiver consists of N consecutive seq #'s
  - Starts with an earliest unacknowledged packet
    - Some packets in the window may have been acknowledged but packet
      could not be given to application

Selective Repeat: Sender, Receiver Windows

- sender_base
- receiver_base
- window size = N
- already ack'ed
- sent, not
  - yet ack'ed
- usable, not
  - yet sent
  - not usable
- out of order
  - (buffered) but
  - already ack'ed
  - expected, not
    - yet received
  - acceptable
    - (within window)
  - not usable

(0) sender view of sequence numbers

(1) receiver view of sequence numbers
Go-Back-N Recovery

- Strategy when receiver sends cumulative ACKs
  - Send nothing for out of order packet – sender will timeout
  - Otherwise sends cumulative ACK
- Sender implements Go-Back-N recovery
  - Set timer upon transmission of packet
  - Retransmit all unacknowledged packets upon timeout
- Performance during loss recovery
  - Single loss can result in many packet retransmissions
  - Timeouts are expensive – add significant delay
  - Puts emphasis on simplicity of implementation
    - E.g., receiver can drop non-contiguous packets (resent anyway)

Outline

- Transport introduction
- TCP connection establishment
- Error recovery and flow control
- Making things work in TCP
- Congestion control
- Transport optimization and futures