The Transport Layer

Michael Brodskiy

Professor: E. Bernal Mor

October 23, 2023

- Transport Services and Protocols
  - Provide logical communication between application processes running on different hosts
  - Transport protocols actions in end systems:
    - Sender: breaks application messages into segments, passes to Network layer
    - Receiver: reassembles segments into messages, passes to Application layer
  - Two transport protocols available to internet applications
    1. TCP
    2. UDP
- Transport vs. Network Layer
  - Network layer: logical communication between two hosts
  - Transport layer: logical communication between processes
    - Relies on, enhances, network layer services
- Two Internet Transport Protocols
  - TCP: Transmission Control Protocol
    - Reliable, in-order delivery
    - Congestion Control
    - Flow Control
    - Connection set-up
  - UDP: User Datagram Protocol
    - Unreliable, unordered delivery
    - No-frills extension of "best-effort" IP

  - Services not available:
    - Delay guarantees
    - Throughput guarantees

- Multiplexing/Demultiplexing
  - Multiplexing at sender: Handle data from multiple sockets, add transport header (later used for demultiplexing)
  - How demultiplexing works
    - Host receives IP packets
      - Each packet has source IP address, destination IP address
      - Each packet carries one transport-layer segment
      - Each segment has source, destination port number
    - Host uses IP address and port numbers to direct segment to appropriate socket
  - Connectionless Demultiplexing
    - Create a socket in the client, the Transport layer automatically assigns a host-local port number to the socket
    - When data is sent into UDP socket, must specify
      - Destination IP address
      - Destination port number
    - When a host receives UDP segment, the Transport layer:
      - Checks destination port number in segment
      - Directs UDP segment to socket with that port number
    - IP datagrams with same destination port number but different source IP addresses and/or source port numbers will be directed to same socket at destination
  - Connection-Oriented Demultiplexing
    - TCP socket identified by 4-tuple:
      - Source IP address
      - Source port number
      - Destination IP address
      - Destination port number
    - Demultiplexing receiver users all four values to direct segment to appropriate socket
    - A server may support simultaneous TCP sockets:
      - Each socket identified by its own 4-tuple
      - Each socket associated with a different connecting client

    - Note: the TCP server has a welcoming socket
        - Each time a client initiates a TCP connection to the server, a new socket is created for this connection
        - To support n simultaneous connections, the server would need $n + 1$ sockets

- Connectionless Transport: UDP
  - "No frills", "bare bones" Internet transport protocol
  - "Best effort" service, UDP segments may be:
    - Lost
    - Delivered out-of-order to application
  - Connectionless:
    - No handshaking between UDP sender, receiver
    - Each UDP segment handled independently of others
  - Why is there a UDP?
    - No connection establishment (which can add RTT delay)
    - Simple: no connection state at sender, receiver
    - Small header size
    - No congestion control
      - UDP can blast away as fast as desired
      - It can function in the face of congestion
  - UDP used in:
    - Streaming multimedia apps (loss tolerant, rate sensitive)
    - DNS
    - HTTP/3
  - If reliable transfer or other services needed over UDP (like in HTTP/3)
    - Add needed reliability at application layer
    - Add congestion control at application layer

- Internet Checksum
  - Goal: detect "errors" (*e.g.* flipped bits) in received segment — optional if UDP segment is encapsulated in IPv4 packet: carries all-zeros if unused
  - Sender:
    - Treat segment contents, including UDP header fields and IP addresses, as sequence of 16-bit words
    - Checksum: ones complement of the ones complement sum of all 16-bit words

        - Sender puts checksum value into UDP checksum field
    - Receiver — error detected?
        - All 16-bit words are added, including checksum
        - Result = 1111111111111111 → no error
            - But maybe errors, nonetheless?
        - Result ≠ 1111111111111111 → Error
            - Do not recover from the error: discard segment or pass damaged data with a warning

- Principles of Reliable Data Transfer
    - Consider only unidirectional data transfer
    - But control information will flow in both directions
    - Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)

- Components of Error Recovery
    - Checksum to detect bit errors
    - Acknowledgments (ACKs): receiver explicitly tells sender the packet received is OK
    - Negative Acknowledgments (NAKs): receiver explicitly tells sender that pack had errors
    - Sender retransmits packet on receipt of NAK
    - What happens if ACK/NAK is corrupted?
        - Sender doesn't know what happened at receiver
        - Can't just retransmit, possible duplicate
    - Sender:
        - Sequence number added to packet
        - Two sequence numbers will suffice
        - Must check if received ACK/NAK corrupted
        - Sender must "remember" whether last sent packet had sequence number of 0 or 1
    - Receiver:
        - Must check if received packet is duplicate
            - Receiver must remember whether 0 or 1 is expected packet sequence number
        - If duplicate, send ACK to force sender to move on
        - Note: receiver can not know if its last NAK/ACK received ok at sender

- A similar protocol without NAKs may be developed:
  - Instead of NAK, receiver sends ACK for last packet received OK
    - Receiver must explicitly include in the ACK the sequence number of packet being ACKed
  - Duplicate ACK at sender results in same action as NAK: retransmit current packet
  - TCP uses this approach to be NAK-free
- This protocol may be further developed:
  - Approach: sender waits "reasonable" amount of time for ACK
  - Only retransmit if no expected ACK received in this time
  - If packet (or ACK) just delayed (not lost):
    - Retransmission will be duplicate, but sequence number already handles this
    - Receiver must specify sequence number of packet being ACKed
  - Requires countdown timer to interrupt after "reasonable" amount of time (timeout)

- Utility
  - For a "Stop and Wait" approach, the utility can be defined as:

$$U_{sender} = \frac{L/R}{RTT + (L/R)}$$

  - Thus, the performance of stop and wait would not be food if the RTT is high
  - Stop and wait limits performance of underlying infrastructure

- Pipelined Protocols
  - Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets
    - Range of sequence numbers must be increaseL k-bit sequence number in packet header
    - Buffering at sender and/or receiver
  - Two generic forms of pipelined protocols: Go-Back-N (GBN) and selective repeat

- Go-Back-N
  - A "sliding window" protocol
  - Sender window: N consecutive sequence numbers allowed for sent, unacknowledged packets
    - Sender window size is N

- Cumulative ACK: ACK(n) $\rightarrow$ Acknowledges all packets up to N, including sequence number n
  * On receiving ACK(n) $\rightarrow$ move window forward to begin at $n + 1$

- Selective Repeat
  - Sender Window
    * N consecutive sequence numbers
    * Limits sequence numbers of sent, unacknowledged packets
    * Sender window size is N
  - Sender maintains a timer for each unACKed packet
  - Sender times-out and retrsnmits individually unACKed packets
  - Receiver Window
    * N consecutive sequence number
    * Limits sequence numbers of received packets that are accepted
    * Receiver window size is N
  - Buffers packets, as needed for eventual in-order delivery to upper layer
  - Receiver individually acknowledges all correctly received packets

- Connection-Oriented Transport: TCP
  - Point-to-point: one sender, one receiver
  - Reliable, in-order byte stream:
    * No "message boundaries"
  - Full duplex data:
    * Bi-directional data flow in same connection
    * MSS: maximum segment size
  - Cumulative ACKs
  - Pipelining:
    * TCP congestion and flow control set the window size
  - Connection-oriented
    * Handshaking (exchange of control messages) initializes sender and receiver state before data exchange
  - Flow control
    * Sender will not overwhelm receiver

- TCP Sequence Numbers and ACKs

- Sequence Numbers:
  * Byte stream "number" of first byte in segment data
- Acknowledgments:
  * Piggbacked ACK on the other-direction data segment
  * Sequence number of the next byte expected from other side
  * Cumulative ACK
- How receiver handles out-of-order segments?
  * TCP specifications does not say — up to implementer
  * In practice: buffer out-of-order segments

- TCP RTT and Timeout
  - How to set TCP timeout value?
    1. Longer than RTT
       * But RTT varies
    2. Too short: premature timeout, unnecessary retransmissions
    3. Too long: slow reaction to segment loss
  - How to estimate RTT?
    * Sample RTT: measured time from segment transmission until ACK receipt
      · Ignore retransmissions
      · Sample RTT may have high variability
    * Estimated RTT: average of several recent Sample RTT measurements
      · Estimates a typical RTT
      · "Smoother" variability
  - Timeout interval: Estimated RTT plus "safety margin"
    * Large variation in Estimated RTTL want a larger safety margin

$$Timeout = EstimatedRTT + 4 \cdot DevRTT$$

  - Dev RTT: EWMA of Sample RTT deviation from Estimated RTT:

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$$

    * β is typically .25

- TCP Sender (Simplified)
  - Event: data received from application
  - Create a segment with sequence number
  - Sequence number is byte-stream number of first data byte in segment

  - Start timer if not already running
    * Single retransmission timer
    * Think of timer as for oldest unACKed segment
    * Expiration interval: TimeOutInterval
  - Event: timeout
    * Retransmit segment that caused timeout
    * Restart timer: twice the previous value
  - Event: ACK received
    * If ACK acknowledges previously unACKed segments
      · Update what is known to be ACKed

- TCP Receiver: ACK Generation

Event at Receiver	TCP Receiver Action
Arrival of in-order segment with expected sequence number, All data up to expected sequence number already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected sequence number. Another segment has ACK pending.	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-that-expeted sequence number: gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte
Arrival of segment that partially or completely fills a gap.	Immediately send ACK, if segment starts at lower end of gap.

- TCP Fast Retransmit
  - Time-out period often relatively long:
    * Long delay before resending lost packet
  - Detect lost segments via duplicate ACKs
    * Sender often sends many segments back-to-back
    * If segment is lost, there will likely be many duplicate ACKs
  - If sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest sequence number

- TCP Flow Control
  - What happens if network layer delivers data faster than application layer removes data from socket buffers? Overflow

- Flow control — Receiver controls sender, so sender will notoverflow receiver's buffer by transmitting too much, too fast
- RcvBuffer size set via socket options (typically 4096 bytes)
  - Many operating systems auto adjust RcvBuffer
- TCP receiver "advertises" free buffer space in rwnd field in TCP header
- TCP Sender limits amount of unACKed ("in-flight") data to received rwnd
- Guarantees receive buffer will not overflow

- TCP Connection Management
  - Before exchanging data, sender/receiver "handshake":
    - Agree to establish connection (each knowing the other is willing to establish connection)
    - Agree on connection parameters (*e.g.* starting sequence numbers)

- Principles of Congestion Control
  - Informally: "too many sources sending too much data too fast for the network to handle"
  - Manifestations:
    - Lost packets (buffer overflow at routers)
    - Long delays (queueing in router buffers)
  - Different from flow control (one sender sending too fast for one receiver)
  - A top-10 problem
  - Simplest scenario:
    - One router, infinite buffers
    - No retransmissions needed
    - Output link capacity, R
    - Two flows
    - Sender: original data or arrival rate λ_{in} bps
    - Receiver throughput or goodput λ_{out} bps
    - Lost packets and duplicates:
      - Packets can be lost, dropped at router due to full buffers — requiring retransmission
      - Sender timer can time out prematurely, sending two copies, both of which arrive to the receiver
    - "Costs of congestion"
      - More work (retransmission) for given receiver throughput

    - Unneeded retransmissions: link carries multiple copies of a packet $\rightarrow$ wasted capacity; decreasing more achievable throughput
    - When packet dropped or when a duplicate is transmitted, any upstream transmission capacity and buffering used for that packet was wasted
    - The network can enter congestion collapse: senders are sending packets at maximum rate; packets suffer long delays or are lost and many of them are duplicates

- Congestion Control
  - End-to-end Congestion Control:
    - No explicit feedback from network
    - Congestion inferred from observed loss and delay
    - Approach usually taken by TCP
  - Network-Assisted Congestion Control:
    - Routers provide direct feedback to sending/receiving hosts with flows passing through congested router
    - May indicate congestion level or explicitly set sending rate
    - TCP ECN, ATM, DECbit protocols

- Rate Control: Congestion Window
  - TCP congestion control keeps track of another variable: congestion window $\rightarrow$ CWND
  - TCP sender limits sender window size: $N = LastByteSent - LastByteAcked \leq min(cwnd, rwnd)$
  - CWND is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

- TCP Congestion Control: AIMD
  - Approach: sender can increase sending rate until congestion is detected, then decrease sending rate and, when recovered from congestion, increase the rate again
  - Additive Increase: increase cognestion window by 1 maximum segment size (MSS) evert RTT until loss event
  - Multiplicative Decrease: cut congestion window in half at each loss event

- Classic TCP Congestion Control
  - Three Phases
    1. Slow start

        * Mandatory
    2. Congestion Avoidance (CA)
        * Mandatory
    3. Fast recovery (FR)
        * Optional
  - Initially: slow start
    * Congestion window bigger than a threshold → CA
  - Congestion: Loss event
    * TCP Reno
      · Loss detected by triple duplicate ACK → FR
      · After correct ACK received → CA
      · Loss detected by timeout event → slow start
    * TCP Tahoe
      · Loss detected by triple duplicate ACK or timeout event → slow start

- TCP Slow Start
  - When connection begins, increase rate exponentially until first loss event:
    * Initially, cwnd = 1 MSS
    * Double CWND every RTT → increment CWND for every ACK received
  - Initial rate is slow, but ramps us exponentially fast
  - When CWND grows larger than a threshold given by variable SSTHRESH:
    * Congestion may be around the corner
    * Enter CA → linear growth

- TCP Congestion Avoidance
  - CWND is conservatively increased by 1 MSS per RTT