Chapter 3: Transport Layer

Our goal:
- understand principles behind transport layer services
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connection-less transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
  - 3.6 Principles of congestion control
  - 3.7 TCP congestion control

Transport services and protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
- send side: breaks app messages into segments, passes to network layer
- recv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
- Internet: TCP and UDP

Transport vs. network layer

- network layer: logical communication between hosts
- transport layer: logical communication between processes
- reliable, enhanced network layer services

Household analogy:
- 12 kids sending letters to 12 kids
- processes = kids
- app messages = letters
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Multiplexing/demultiplexing

- DatagramSocket(99222);

How demultiplexing works

- host receives IP datagrams
- each datagram has source IP address, destination IP address
- each datagram carries transport-layer segment
- each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

Connectionless demultiplexing (cont)

DatagramSocket serverSocket = new DatagramSocket(9428);

Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - destination IP address
  - destination port number
- server host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
Connection-oriented demux (cont)

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order
to app
- connectionless:
  - no connection between UDP sender, receiver
each UDP segment handled independently of others

UDP: more

- often used for streaming multimedia apps
- low latency
- "best effort" service
- used for TCP, ICMP, and other protocols
- reliable transfer over UDP
- adds reliability at application layer
- application-specific error recovery

UDP checksum

Good detect "errors," e.g., flipped bits, in transmitted segment

<table>
<thead>
<tr>
<th>Sender:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. treat segment contents as sequence of 16-bit integers</td>
</tr>
<tr>
<td>2. checksum addition (i.e., complement sum) of segment contents</td>
</tr>
<tr>
<td>3. sender puts checksum value into UDP checksum field</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Receiver:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. compute checksum of received segment</td>
</tr>
<tr>
<td>2. check if computed checksum equals checksum field value</td>
</tr>
<tr>
<td>3. NO - error detected</td>
</tr>
<tr>
<td>4. YES - no error detected</td>
</tr>
<tr>
<td>5. if error detected, maybe try again; otherwise, more later...</td>
</tr>
</tbody>
</table>
Internet Checksum Example

- Note
  - When adding numbers, a carry-out from the most significant bit needs to be added to the result.
- Example: add two 16-bit integers

\[
\begin{align*}
\text{wraparound} & : 1111 1001 1100 1110 1110 1101 1111 1111 0111 1111 1111 1111 1111 1111 1111 \\
\text{sum checksum} & : 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101
\end{align*}
\]

Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Principles of Reliable data transfer

- Important in app., transport, link layers
- Top-10 list of important networking tactical

(a) provided service
(b) service implementation

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (PDR)

Reliable data transfer: getting started

-uds_send() called by app., (e.g., by app.). Puts data to deliver to upper layer
-uds_send() called by uds to deliver data to upper
-uds_send() called by rtt to transfer packet over unreliable channel to receiver
-uds_send() called when packet arrives over one-side of channel
-uds_send() called when packet arrives over one-side of channel
-uds_send() called by rtt to transfer packet over unreliable channel to receiver
**Rdt1.0: reliable transfer over a reliable channel**

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

**Rdt2.0: channel with bit errors**

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:
  - acknowledgements (ACKs), receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs), receiver explicitly tells sender that pkt had errors
- sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback control msgs (ACK, NAK) recv-sender

**Rdt2.0: operation with no errors**

**Rdt2.0: error scenario**
rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?
- sender doesn't know what happened to receiver?
- can't just retransmit, possible duplicate

Handling duplicates:
- sender retransmits current pkt if ACK/NAK garbled
- sender adds sequence number to each pkt
- receiver discards (doesn't deliver) duplicate pkt

Step and wait
- sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs

Sender:
- seq # added to pkt
- two seq. dup (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
- state must "remember" whether "current" pkt has 0 or 1 seq #

Receiver:
- must check if received packet is duplicate
- state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
- receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: retransmit current pkt

rdt2.2: sender, receiver fragments
rdt3.0: channels with errors and loss

New assumption: underlying channel can also lose packets (data or ACKs).
- checksum, seq #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits reasonable amount of time for ACK.
- if no ACK received in this time
- if pkt (or ACK) just delayed (not lost), retransmission will be duplicate, but use of seq #s already handled this
- receiver must specify seq # of pkt being ACKed
- requires counter(s) 

Performance of rdt3.0
- rdt3.0 works, but performance shrinks
- example: 1 Gbps link, 15 ms + e-prop. delay, 1 KB packet:
  \[
  \text{Transmit} = \frac{\text{packet length (in bits)}}{\text{bit/s}} = \frac{1024}{10^9} = 8 \text{ microsec}
  \]
  \[
  U_{\text{utilization}} = \frac{\text{fraction of time sender busy sending}}{\text{RTT} + 2 \times L/R} = \frac{0.00007}{0.00007} = 0.00007
  \]
- 1 KB pkt every 30 microsec = 333 Kbps throughput over 1 Gbps link
- network/protocol limits use of physical resources
Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

- Range of sequence numbers must be increased
- May receive duplicate ACKs (see receiver)

Two generic forms of pipelined protocols: go-Back-N, selective repeat

Go-Back-N

Sender:
- k-bit seq # in pit header
- "window" of up to N consecutive unack'd packets allowed

- Window sizes

- ACK(s): ACKs of all packets up to, including seq # n - "cumulative ACK"
- May receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- Time-out: retransmit seq # and all higher seq # packets in window

GBN: sender extended FSM

ACK-only: always send ACKs for correctly-received pkt with highest-in-order seq #
- May generate duplicate ACKs
- Need only remember expected sequence
- Out-of-order pkt:
  - Discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest-in-order seq #

GBN: receiver extended FSM

ACK-only: always send ACKs for correctly-received pkt with highest-in-order seq #
- May generate duplicate ACKs
- Need only remember expected sequence
- Out-of-order pkt:
  - Discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest-in-order seq #

GBN in action

Pipelining: increased utilization

Expected seqnum

Increase utilization by a factor of 31

Transport Layer 3-45

Transport Layer 3-44

Transport Layer 3-43

Transport Layer 3-42
Selective Repeat

- Receiver individually acknowledges all correctly received packets
- Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only retransmits packets for which ACK not received
- Send timer for each unACKed packet

Sender window
- N consecutive seq #'s
- Acknowledges seq # of sent, unACKed packets

Selective repeat in action

Selective repeat: dilemma

Example:
- seq #s: 0, 1, 2, 3
- window size: 3

Receiver sees no difference in two scenarios
- Incorrectly passes duplicate data as new in (a)
- Q: what relationship between seq # 0, 3?

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
TCP: Overview

- point-to-point
- reliable, in-order byte stream
- full-duplex data
- bi-directional data flow in same connection
- MSS: maximum segment size
- connection-oriented
- handshaking (exchange of control messages) in TCP
- sender/receiver state before data exchange

TCP segment structure

Options (variable length)
Application data (variable length)
Internet checksum
TCP header
Segment number
Sequence number
Transport Layer 347

TCP seq. #s and ACKs

Seq #s
- bytestream: "number" of first byte in segment's data

ACKs
- seq # of next byte expected from other side
- cumulative ACK

Q: how many windows allowed out-of-order segments
- A: TCP spec doesn't say - up to cwnd

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
- two short premature timeouts
- unnecessary retransmissions
- too long slow start reaction to segment loss

TCP Round Trip Time and Timeout

Example RTT estimation:

EstimatedRTT = (1 - a) * EstimatedRTT + a * SampleRTT

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value a = 0.125
TCP Round Trip Time and Timeout

Setting the timeout
- EstimatedRTT plus "safety margin"
  - Large variation in EstimatedRTT -> larger safety margin
- First estimate of how much SimpleRTT deviates from
  EstimatedRTT:
  \[ \text{DevRTT} = (1 - \Phi) \times \text{SimpleRTT} \]
  (typically, \( \Phi = 0.25 \))

Then set timeout:
\[ \text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \]

TCP reliable data transfer

- TCP creates end-to-end service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeouts
  - Duplicate acks
- Initially consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
- 3.6 Principles of congestion control
  - 3.7 TCP congestion control

TCP sender events:

- Data-rcvd from app:
  - Create segment with seq \( i \)
  - seq \( i \) is byte stream number of first data byte in segment
- Start timer if not already running (think of timer as for oldest unacknowledged segment)
- Expire timeout if
  \[ \text{ElapsedTime} > 4 \times \text{DevRTT} \]

TCP: retransmission scenarios

- Last ACK received
  - Last ACK stays in the system
- Last ACK received
  - Last ACK stays in the system

TCP sender (simplified)

- Comment: Send beside last correctly acknowledged byte
- strncmp (Sendbase + y, y + Sendlen, 7) as the new wants 73):
  - y = Sendbase, so that new data is acknowledged

TCP: retransmission scenarios
TCP retransmission scenarios (more)

Fast Retransmit
- Time-out period often relatively long.
- May delay before resending last 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.

Chapter 3 outline
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - congestion management
  - TCP congestion control

TCP ACK generation [RFC 1122, RFC 2581]
- Event at Receiver
  - Arrivals of in-order segments with expected seq.
    - Data sent if already ACKed.
  - Arrivals of out-of-order segments
    - Delayed ACK sent if not expected.
- TCP Receiver action
  - Immediate send single cumulative ACK, ACKing both in-order segments
  - Immediate send duplicate ACK indicating seq.
  - Delay if new expected byte

Fast retransmit algorithm:
- event: ACK received, with ACK field value of y
  - y > Sackpack
  - Sack Warning
  - If there were previously unacknowledged segments
    - Sack Warning
    - Start timer
    - Increment count of dup ACKs received for y
      - If count of dup ACKs received for y > 3, send segment with sequence number
    - Select duplicate ACK for already ACKed segment
    - Fast retransmit

TCP Flow Control
- Receive side of TCP connection has a receive buffer
  - Flow control
    - Credit/Window
    - Speed-matching service
      - Matching the send rate to the receiving app's drain rate
TCP Flow control: how it works

- Receiver advertises spare room by including value of RecvWindow in segments.
- Sender limits unACKed data to RecvWindow.
- Guarantee received buffer doesn't overflow.

TCP Connection Management

Recall: TCP sender/receiver establish "connection" before exchanging data segments.
- Initiate TCP variables:
  - seq #
  - Buffers, flow control info (i.e., wait windows).
- Client connection initiated by socket listen(port number).
- Server contacted by client socket connection():
- Wait for connection(), accepts().

Three way handshake:

- Step 1: Client host sends TCP SYN segment to server:
  - Specifies initial seq #.
  - No data.
- Step 2: Server host receives SYN, replies with SYNACK segment:
  - Server allocates buffers.
  - Specifies server initial seq #.
- Step 3: Client receives SYNACK, replies with ACK segment, which may controllled over.

TCP Connection Management (cont.)

Step 3: Client receives FIN, replies with ACK:
- Enter "time-wait" will receive with ACK to received FIN.

Step 4: Server receives ACK. Connection closed.

Note: With small modification, can handle simultaneous FINs.

Chapter 3 outline

- 3.1 Transport-layer services:
  - Segment structure.
  - Reliable data transfer.
- 3.3 Connectionless transport: UDP:
  - Flow control.
- 3.6 Principles of congestion control:
  - 3.7 TCP congestion control.
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Principles of Congestion Control

Congestion:
- informally: "too many sources sending too much data to a fast for network to handle"
- different from flow control
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
  - a top-10 problem.

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput

Causes/costs of congestion: scenario 2

- always \( \lambda_{in} = \lambda_{out} \) (as input)
- "perfect" retransmission only when lost \( \lambda_{in} > \lambda_{out} \)
- retransmission of delayed (not just) packet makes \( \lambda_{in} \) larger when perfect case for some \( \lambda_{out} \)

- costs of congestion
  - more work (retrans) for given "goodput"
  - unneeded retransmissions: link carries multiple copies of pkt

Causes/costs of congestion: scenario 3

- four senders
- multipath paths
- time out/retransmit

- what happens as \( p \) increases?
Causes/costs of congestion: scenario 3

Another "cost" of congestion:
- when packet dropped, any "upstream transmission capacity used for that packet was wasted"

Approaches towards congestion control

Two broad approaches towards congestion control:
- End-end congestion control
  - no explicit feedback from network
  - congestion inferred from end-system observed loss, delay
  - approach taken by TCP
- Network-assisted congestion control
  - masters provide feedback to end systems
  - single bit indicating congestion (SNA, DECT, TCP/IP, P Patrick ATM)
  - explicit rate sender should send at

Case study: ATM ABR congestion control

Case study: ATM ABR congestion control

Chapter 3 outline

TCP congestion control: additive increase, multiplicative decrease

- Approach: increase transmission rate (window size), probing for usable bandwidth, until loss occurs
- additive increase: increase CongWin by 1 MSS every RTT until loss detected
- multiplicative decrease: cut CongWin in half after loss

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
TCP Congestion Control: details

- Sender limits transmission
  LastByteOut - LastByteAcked
  CongWin

- Roughly:
  CongWin = RTT * Bytes/sec

- CongWin is dynamic, function of perceived network congestion

- How does sender perceive congestion?
  Loss event = timeout or 3 duplicate acks
  TCP sender reduces CongWin after loss event

- Three mechanisms:
  - AIMD
  - Slow start
  - Conservative after timeout GBN

TCP Slow Start

- When connection begins, CongWin = 1 MSS
- Example: MSS = 500 bytes & RTT = 200 msec
- Initial rate = 20 kbps

- Available bandwidth may be >> MSS/RTT

- Desirable to quickly ramp up to respectable rate

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double CongWin every RTT
  - decays by incrementing CongWin for every ACK received

- Summary: initial rate is slow but ramps up exponentially fast

Refinement

- What should the exponential increase switch to linear?
  - When CongWin gets to 1/2 of its value before timeout

- Implementation:
  - Variable Threshold
  - At loss event, Threshold is set to 1/2 of CongWin just before loss event

Refinement: inferring loss

- After 3 dup ACKs:
  - CongWin is cut in half
  - window then grows linearly

- But after timeout event:
  - CongWin instead set to 1 MSS
  - window then grows exponentially
  - to a threshold, then grows linearly

Summary: TCP Congestion Control

- When CongWin is below Threshold, sender in slow start phase, window grows exponentially.

- When CongWin is above Threshold, sender is in congestion avoidance phase, window grows linearly.

- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.

- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS
TCP sender congestion control

<table>
<thead>
<tr>
<th>State</th>
<th>Brate</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronize (SR)</td>
<td>High</td>
<td>Congestion window expanded</td>
<td>Congestion window expanded every RTT</td>
</tr>
<tr>
<td>Synchronize (SR)</td>
<td>Low</td>
<td>Congestion window contracted</td>
<td>Congestion window contracted every RTT</td>
</tr>
<tr>
<td>No SR</td>
<td>High</td>
<td>Transmission at rate of congestion window</td>
<td>Congestion window expanded every RTT</td>
</tr>
<tr>
<td>No SR</td>
<td>Low</td>
<td>Transmission at rate of congestion window</td>
<td>Congestion window expanded every RTT</td>
</tr>
</tbody>
</table>

TCP throughput

- What's the average throughput of TCP as a function of window size and RTT?
  - Ignore slow start
  - Let W be the window size when loss occurs.
  - When window is W, throughput is W/RTT
  - Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughput: .75 W/RTT

TCP Futures

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

\[
\text{Throughput} = \frac{\text{Window size}}{\text{Loss rate}}
\]

- L = 2 \times 10^{-5} \text{ Wm}
- New versions of TCP for high-speed needed

TCP Fairness

Fairness goal: if K TCP sessions share a bottleneck link of bandwidth R, each should have average rate of R/K

![TCP Fairness Diagram](image)

Why is TCP fair?

Two competing sessions:
- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally

Chapter 3: Summary

- Principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
  - Instantiation and implementation in the Internet
- UDP
- TCP

Next:
- Leaving the network "edge" (application, transport layers)
- Into the network "core"