Chapter 3: Transport Layer

Chapter goals:
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet

Chapter Overview:
- transport layer services
- multiplexing/demultiplexing
- connectionless transport: UDP
- principles of reliable data transfer
- connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management
- principles of congestion control
- TCP congestion control

Transport services and protocols
- provide logical communication between app' processes running on different hosts
- transport protocols run in end systems
- transport vs network layer services:
  - network layer: data transfer between end systems
  - transport layer: data transfer between processes
    - relies on, enhances, network layer services
Transport-layer protocols

Internet transport services:
- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast

Multiplexing/demultiplexing

Recall: segment - unit of data exchanged between transport layer entities
  - aka TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct app layer processes
Multiplexing/demultiplexing

Multiplexing:
gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications

Multiplexing/demultiplexing: examples

- host A server B
  - source port: x, dest port: 23
  - source port: 23, dest port: x
  - port use: simple telnet app

- Web client host C
  - Source IP: C, Dest IP: B
  - source port: y, dest. port: 80
  - Source IP: C, Dest IP: B
  - source port: x, dest. port: 80
  - port use: Web server

- Web client host A
  - Source IP: A, Dest IP: B
  - source port: x, dest. port: 80
  - Web server B

TCP/UDP segment format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (message)</td>
<td></td>
</tr>
</tbody>
</table>
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 handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses (why?):
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recover!

UDP segment format:
- 32 bits
- Length, in bytes of UDP segment, including header
- checksum
- length
- dest port #
- source port #
- Application data (message)

3: Transport Layer 3a-7
**UDP checksum**

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless? More later …

---

**Principles of Reliable data transfer**

- important in app., transport, link layers
- top-10 list of important networking topics!

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: getting started

We'll:
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

State: When in this "state" next state uniquely determined by next event
Event causing state transition
Actions taken on state transition

3: Transport Layer 3a-11
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

![Diagram of Rdt1.0](image)

(a) rdt1.0: sending side
(b) rdt1.0: receiving side

Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender
rdt2.0: FSM specification

sender FSM

wait for call from above

wait for ACK or NAK

receiver FSM

wait for call from below

rdt.0: in action (no errors)

sender FSM

wait for call from above

wait for ACK or NAK

receiver FSM
rdt2.0: in action (error scenario)

sender FSM

wait for call from above

wait for ACK or NAK

wait for call from below

receiver FSM

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver!
- can’t just retransmit: possible duplicate

What to do?
- sender ACKs/NAKs receiver’s ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

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

stop and wait
Sender sends one packet, then waits for receiver response
rdt2.1: sender, handles garbled ACK/NAKs

- rdt sender(data)
  - compute checksum
  - make_pkt(sndpkt, data, checksum)
  - udt_send(sndpkt)

- rdt rcv(rcvpkt)
  - wait for call 0 from above
  - rdt rcv(rcvpkt) && not corrupt(rcvpkt)
    - && is ACK(rcvpkt)
    - rdt_send(sndpkt)

- rdt rcv(rcvpkt) && (corrupt(rcvpkt) || is NAK(rcvpkt))
  - udt_send(sndpkt)

- rdt rcv(rcvpkt)
  - wait for ACK or NAK
  - rdt rcv(rcvpkt) && not corrupt(rcvpkt)
    - && is ACK(rcvpkt)
    - extract(rcvpkt, data)
    - deliver data(data)
    - make_pkt(sndpkt, ACK, checksum)
    - udt_send(sndpkt)

- rdt rcv(rcvpkt)
  - wait for call 1 from above
  - rdt rcv(rcvpkt) && not corrupt(rcvpkt)
    - && has seq(rcvpkt)
    - extract(rcvpkt, data)
    - deliver data(data)
    - make_pkt(sndpkt, ACK, checksum)
    - udt_send(sndpkt)

rdt2.1: receiver, handles garbled ACK/NAKs

- rdt rcv(rcvpkt)
  - wait for 0 from below
  - rdt rcv(rcvpkt) && not corrupt(rcvpkt)
    - && has seq(rcvpkt)
    - extract(rcvpkt, data)
    - deliver data(data)
    - make_pkt(sndpkt, ACK, checksum)
    - udt_send(sndpkt)

- rdt rcv(rcvpkt)
  - wait for 1 from below
  - rdt rcv(rcvpkt) && not corrupt(rcvpkt)
    - && has seq1(rcvpkt)
    - extract(rcvpkt, data)
    - deliver data(data)
    - make_pkt(sndpkt, ACK, checksum)
    - udt_send(sndpkt)
rdt2.1: discussion

Sender:
- seq # added to pkt
- two seq. #’s (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 NAKs 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
rtdt3.0: 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

**Q:** how to deal with loss?

- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

**Approach:** sender waits “reasonable” amount of time for ACK

- retransmits 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 handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

---

rdt3.0 sender

```plaintext
rdt_send(data)
compute_checksum(make_pkt(data, checksum), start_timer)

wait for call from above

rdt_rcv(pkt)
& & rdt_corrupt(pkt)
& & isACK(pkt)

timeout
udt_send(n+d+pkt)
start_timer

wait for call from above

rdt_rcv(pkt)
& & rdt_corrupt(pkt)
& & isACK(pkt)

rdt_rcv(pkt)
& & rdt_corrupt(pkt)
& & isACK(pkt)

timeout
udt_send(n+d+pkt)
start_timer

wait for call from above
```
rdt3.0 in action

(a) operation with no loss

(b) lost packet

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{transmit} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec} \]

\[ \text{Utilization} = U = \frac{\text{fraction of time}}{\text{sender busy sending}} = \frac{8 \text{ microsec}}{30.016 \text{ msec}} = 0.00015 \]

- 1KB pkt every 30 msec \(\rightarrow\) 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver
- Two generic forms of pipelined protocols: go-Back-N, selective repeat
Go-Back-N

Sender:
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM

```c
rat_send(data)
if (nextseqnum < base+N)
  compute chksum;
  make_pkt(sendpkt[nextseqnum].nextseqnum.data,chksum);
  udt_send(sendpkt[nextseqnum]);
else
  if (base == nextseqnum)
    start_timer
    nextseqnum = nextseqnum + 1
  else
    refuse_data(data)
```

```c
cdf((cv.pkt) & notcorrupt(cv.pkt))
base = getacknum(cv.pkt)+1
if (base == nextseqnum)
  stop_timer
else
  start_timer

timeout
start_timer
udt_send(sendpkt(base))
udt_send(sendpkt(base+1))
......
udt_send(sendpkt(nextseqnum-1))
```
**GBN: receiver extended FSM**

```
| default | rdt_rcv(rcvpkt) &
| udt_send(indpkt) | notcorrupt(rcvpkt) &
| rdt_send(WAIT) | has_seqnum(rcvpkt,expectedseqnum)
| extract(rcvpkt.data) | make_pkt(indpkt,ACK,expectedseqnum)
| deliver_data(data) | udt_send(indpkt)
```

**receiver simple:**

- **ACK-only:** always send ACK for correctly-received pkt with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember expectedseqnum
- **out-of-order pkt:**
  - discard (don't buffer) -> no receiver buffering!
  - ACK pkt with highest in-order seq #

---

**GBN in action**

- **sender:**
  - send pkt0
  - send pkt1
  - send pkt2
  - send pkt3 (wait)

- **receiver:**
  - rcv pkt0, send ACK0
  - rcv pkt1, send ACK1
  - rcv pkt3, discard
  - send ACK1
  - rcv pkt4, discard
  - send ACK1
  - rcv pkt5, discard
  - send ACK1
  - rcv pkt2, deliver
  - send ACK2
  - rcv pkt3, deliver
  - send ACK3

(pkt2 timeout)
Selective Repeat

- receiver individually acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - $N$ consecutive seq #'s
  - again limits seq #'s of sent, unACKed pkts

Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

**Data from above:**
- If next available seq # in window, send pkt

**Timeout(n):**
- Resend pkt n, restart timer

**ACK(n) in [sendbase, sendbase+N]:**
- Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

**Sender**
- Packet n in [rcvbase, rcvbase+N-1]
- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

**Receiver**
- Packet n in [rcvbase-N, rcvbase-1]
- ACK(n)
- Otherwise:
  - Ignore

Selective repeat in action

```
pkt0 sent  0 1 2 3 4 5 6 7 8 9
pkt1 sent  0 1 2 3 4 5 6 7 8 9
pkt2 sent  0 1 2 3 4 5 6 7 8 9
pkt3 sent, window full 0 1 2 3 4 5 6 7 8 9
ACK0 rcvd, pkt4 sent 0 1 2 3 4 5 6 7 8 9
pkt2 timeout, pkt2 resent 0 1 2 3 4 5 6 7 8 9
ACK1 rcvd, pkt5 sent 0 1 2 3 4 5 6 7 8 9

pkts rcvd, delivered, ACK0 sent 0 1 2 3 4 5 6 7 8 9
pkts rcvd, delivered, ACK1 sent 0 1 2 3 4 5 6 7 8 9
pkts rcvd, delivered, ACK2 sent 0 1 2 3 4 5 6 7 8 9
```

```
pkt3 rcvd, buffered, ACK3 sent 0 1 2 3 4 5 6 7 8 9
pkt4 rcvd, buffered, ACK4 sent 0 1 2 3 4 5 6 7 8 9
pkt2 rcvd, deliver pkts 2,3,4
ACK2 sent

pkt5 rcvd, delivered, ACK5 sent 0 1 2 3 4 5 6 7 8 9
```
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 # size and window size?