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
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**

- application
- transport
- network

*Segment header*

- Application-layer data

P1

- P3

- P4

- P2

3: Transport Layer
Multiplexing/demultiplexing:

- 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

TCP/UDP segment format:

- source port #
- dest port #
- other header fields
- application data (message)
Multiplexing/demultiplexing: examples

- Host A to Server B: Source port: x, dest. port: 23
- Source port: 23, dest. port: x

Port use: simple telnet app

- Web client from Host A to Web server B: Source IP: A, dest. port: 80
- Source IP: B, dest. port: 80

- Web client from Host C to Web server B: Source IP: C, dest. port: 80
- Source IP: B, dest. port: 80

Port use: Web server
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](image)

- source port #
- dest port #
- length
- checksum

Application data (message)

UDP segment format

Length, in bytes of UDP segment, including header
**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

**send side**

- **rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**receive side**

- **deliver_data()**: called by rdt to deliver data to upper
- **rdt_rcv()**: called when packet arrives on rcv-side of channel
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

event
actions
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

(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

receiver FSM

```
rdt_send(data)
compute checksum
make_pkt(sndpkt, data, checksum)
udt_send(sndpkt)

wait for call from above

wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isNACK(rcvpkt)
udt_send(NACK)

wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```
rdt2.0: in action (no errors)

sender FSM

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

sender FSM

receiver FSM
rtd2.0 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

**Flowchart:***
- `rdt_send(data)`
  - `compute checksum`
  - `make_pkt(sndpkt, 0, data, checksum)`
  - `udt_send(sndpkt)`

**Transition States:**
1. **Wait for call0 from above**
   - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`
   - `udt_send(sndpkt)`
2. **Wait ACK or NAK 0**
   - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`
   - `wait for call1 from above`

**Actions:**
- `rdt_send(data)`
  - `compute checksum`
  - `make_pkt(sndpkt, 1, data, checksum)`
  - `udt_send(sndpkt)`
rdt2.1: receiver, handles garbled ACK/NAKs
**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
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

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

rdt_send(data)
compute chksum
make_pkt(sndpkt, 0, data, chksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
rdt_send(data)
compute chksum
make_pkt(sndpkt, 1, data, chksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& isACK(rcvpkt, 1)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& isACK(rcvpkt, 0)

wait for call from above

wait ACK0
timeout
udt_send(sndpkt)
start_timer

wait for call from above

wait ACK1
timeout
udt_send(sndpkt)
start_timer
rdt3.0 in action

(a) operation with no loss

(b) lost packet
rdt3.0 in action

(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:

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

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

- 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*

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
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

```plaintext
rdt_send(data)

if (nextseqnum < base+N) {
    compute checksum
    make_pkt(sndpkt(nextseqnum)), nextseqnum, data, checksum
    udt_send(sndpkt(nextseqnum))
    if (base == nextseqnum)
        start_timer
    nextseqnum = nextseqnum + 1
} else
    refuse_data(data)

rdt_rcv(rcv_pkt) & notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
if (base == nextseqnum)
    stop_timer
else
    start_timer

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

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)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

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
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

**sender**

- **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 #

**receiver**

- **pkt 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

- **pkt 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

pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 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
0 1 2 3 4 5 6 7 8 9

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?