Transport Layer

Srinidhi Varadarajan

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

Multiplexing/demultiplexing: examples
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 recovery!

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.

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: From current "state" next state uniquely determined by next event

event: From current "event" next state uniquely determined by next event
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 read data from underlying channel

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 packet was received correctly
  - negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK
- human scenarios using ACKs, NAKs?
  - Telephone conversation. OK, Could you repeat that please?
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) \(\text{rcvr} \to \text{sender}\)

Rdt2.0: FSM specification

sender FSM
receiver FSM

Rdt2.0: in action (no errors)

sender FSM
receiver FSM

Rdt2.0: in action (error scenario)

sender FSM
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

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

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 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

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

rdt3.0 sender
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:
  \[\text{rate} = \frac{8 \text{Kbps}}{15 \text{ms}} = 8 \text{KB/sec}\]
  \[
  \text{Utilization} = U = \frac{\text{fraction of time sender busy sending}}{8 \text{ microsec}} = 0.00015
  \]
  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

Homework Question?

Homework Question:
- Start with the 1 bit sequence number stop and wait protocol shown in rdt 3.0. For simplicity, also assume that you have a perfect checksum mechanism that can detects all types of packet corruption. Does the protocol handle all possible error conditions, such as lost packets, delayed packets/ACKs etc. If not, show an error condition that can occur which will not be detected by rdt 3.0.
  - Hint: Think of what will happen if you use rdt 3.0 in the Internet, where the source and destination go through multiple routers, and there are multiple routes between source and destination.
  - Suggest modifications to the rdt 3.0 sender FSM to improve response time during errors, without introducing additional states.

Reading

Required Reading