Reliable data transfer: rdt2.2, rdt3.0, GBN, and SR

Many slides are (inspired/adapted) from the above source © all material copyright; all rights reserved for the authors

Office Hours: T 17:00–18:00 R 9:00–10:00 BA4222
TA Office Hours: W 16:00-17:00 BA3201 R 10:00-11:00 BA7172
csc358ta@cs.toronto.edu

Transport Layer 3-2

Send side

rdt_send(): called from above, e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by rdt to deliver data to upper

receive side

udt_send(): called by rdt to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Transport Layer 3-3

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

Wait for call from above

sender

Wait for call from below

receiver

Transport Layer 3-4

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
  - feedback: control msgs (ACK, NAK) from receiver to sender

Transport Layer 3-5

rdt2.0: FSM specification

Wait for call from above

sender

Transport Layer 3-6

rdt2.0: operation with no errors

Wait for call from above

receiver

Transport Layer 3-7

Transport Layer 3-8
**rdt2.0: error scenario**

```
- rdt_send(data)
  snpkt = make_pkt(data, checksum)
  udt_send(snpkt)
- rdt_rcv(rcvpkt) & isACK(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  udt_send(ACK)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  udt_send(NAK)
- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
  udt_send(snpkt)
- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
  Wait for call from below
```

**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

**handling duplicates:**
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- 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_send(data)
  snpkt = make_pkt(0, data, checksum)
  udt_send(snpkt)
- rdt_rcv(rcvpkt) & notcorrupt(rcvpkt) & has_seq0(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  rdt_send(data)
  snpkt = make_pkt(1, data, checksum)
  udt_send(snpkt)
- rdt_rcv(rcvpkt) & notcorrupt(rcvpkt) & has_seq1(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  rdt_send(data)
  snpkt = make_pkt(NAK, checksum)
  udt_send(snpkt)
- rdt_rcv(rcvpkt) & notcorrupt(rcvpkt) & has_seq0(rcvpkt)
  extract(rcvpkt, data)
  deliver_data(data)
  rdt_send(data)
  snpkt = make_pkt(ACK, checksum)
  udt_send(snpkt)
```

**rdt2.1: receiver**

```
- rdt_rcv(rcvpkt) & notcorrupt(rcvpkt)
  & has_seq0(rcvpkt)
  & has_seq1(rcvpkt)
  or
  & has_seq0(rcvpkt) & has_seq1(rcvpkt)
  or
  & has_seq0(rcvpkt) & has_seq1(rcvpkt)
  & has_seq1(rcvpkt)
  send_pkt = make_pkt(ACK, checksum)
  udt_send(send_pkt)
```

---

**rdt2.1: summary**

**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 expected pkt should have seq # of 0 or 1

**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
**rtd2.2: sender, receiver fragments**

**Transport Layer 3-13**

- **sender FSM**
  - `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `wait for ACK 0 from above`

- **receiver FSM**
  - `rdt_rcv(rcvpkt)`
  - `& & corrupt(rcvpkt)`
  - `& & isACK(rcvpkt, 0)`
  - `& & notcorrupt(rcvpkt)`
  - `& & isACK(rcvpkt, 1)`
  - `& & notcorrupt(rcvpkt)`

**rtd3.0: channels with errors and loss**

**Transport Layer 3-14**

- **new assumption:** underlying channel can also lose packets (data, ACKs)
- **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 seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
  - requires countdown timer

**rtd3.0 sender**

**Transport Layer 3-15**

- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `wait for ACK 0 from above`
- `timeout udt_send(sndpkt)`
- `timeout udt_send(sndpkt)`
- `timeout udt_send(sndpkt)`
- `timeout udt_send(sndpkt)`

**rtd3.0 in action**

**Transport Layer 3-16**

- **a) no loss**
  - `send pkt0`
  - `rcv ack0`
  - `send pkt1`
  - `rcv ack1`

- **b) packet loss**
  - `send pkt0`
  - `timeout resend pkt1`
  - `rcv pkt1`
  - `rcv pkt0`
  - `send ack0`
  - `send ack1`
  - `send ack0`

**Performance of rtd3.0**

- rtd3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
  \[ D_{trans} = \? \]
  - \[ U_{sender} = \frac{\text{fraction of time sender busy sending}}{\text{fraction of time sender busy sending}} \]
  - \[ U_{sender} = \? \]

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!
Transport Layer 3

**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:**
    - sender can have up to N unacked packets in pipeline
    - receiver only sends cumulative ack
      - does’t ack packet if there’s a gap
    - sender has timer for oldest unacked packet
      - when timer expires, retransmit all unacked packets
  - **Selective Repeat:**
    - sender can have up to N unack’ed packets in pipeline
    - rcvr sends individual ack for each packet
    - sender maintains timer for each unacked packet
      - when timer expires, retransmit only that unacked packet

**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 receive duplicate ACKs (see receiver)
    - timer for oldest in-flight pkt
  - timeout(n): retransmit packet n and all higher seq # pkts in window

**GBN: sender extended FSM**

- rdt_send(data)
- udt_send(sndpkt[base+1])
- rdt_send(sndpkt[nextseqnum])
- rdt_rcv(rcvpkt)
- rdt_rcv(rcvpkt[base+1])
- rdt_rcv(rcvpkt[nextseqnum])
- udt_send(sndpkt[base+1])
  - udt_send(sndpkt[nextseqnum])
  - udt_send(sndpkt[nextseqnum+1])
  - base = getacknum(rcvpkt)+1
  - start_timer
  - stop_timer
  - rdt_send(sndpkt)
  - & corrupt(rcvpkt)
  - & corrupt(rcvpkt)
  - & corrupt(rcvpkt)

**Pipelining: increased utilization**

- first packet bit transmitted, t = 0
- last packet bit transmitted, t = L / R
- ACK arrives, send next packet, t = RTT + L / R

U sender = \( \frac{L}{R} \times \frac{3L}{RTT + L / R} \)

\[ \begin{align*}
U_{sender} & = \frac{L}{R} \\
& \approx \frac{0.008}{30.008} \\
& = 0.00027
\end{align*} \]
GBN: receiver extended FSM

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!
  - re-ACK pkt with highest in-order seq #

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
  - limits seq #’s of sent, unACKed pkts

Selective repeat in action

sender
_pkt n in [sendbase, sendbase+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
_pkt n in [recvbase-N, recvbase-1]:
- ACK(n)
- otherwise:
  - ignore

Q: what happens when ack2 arrives?
Selective repeat: dilemma

example:
- seq #s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?

Next
- Midterm on Chapters 1 and 2
- TCP
- Congestion control