CMPE 150: Introduction to Computer Networks

Katia Obraczka
Computer Engineering
UCSC Baskin Engineering
Lecture 9
Announcements

- Homework 2 posted.
  - Covers Application Layer (chapter 2).

- Midterm exam:
  - February 19th.
  - It may get pushed back so stayed tuned...
  - Closed book, closed notes.

- i-NRG field trip make-up.
  - Tomorrow, Wed, 02.06, 10-12.
Reliable data transfer
What is reliable data transfer?

- Reliable data delivery over unreliable channel.

- Unreliable channel:
  - Transmission errors.
  - Packet loss.
Principles of reliable data transfer

(a) provided service
Principles of reliable data transfer

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol
Principles of reliable data transfer

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

- **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 layer.
- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver.
- **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

![Finite State Machine Diagram]

- **state**: when in this “state” next state uniquely determined by next event
- **event causing state transition**
- **actions taken on state transition**
- **event**
- **actions**

*CMPE 150 Winter 13*
**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
```

```
rdt_send(data)
```

```
packet = make_pkt(data)
```

```
udt_send(packet)
```

```
Wait for call from below
```

```
rdt_rcv(packet)
```

```
extract (packet, data)
```

```
deliver_data(data)
```

sender

receiver
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) rcvr->sender

ARQ protocols: Automatic Repeat reQuest protocols
**rdt2.0: FSM specification**

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

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
```

**sender**

```
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
```

**receiver**

```
Wait for call from below
```

**NOTE: “Stop-and-Wait” protocol**
rdt2.0: operation with no errors

- **rdt_send(data)**
  - snkpkt = make_pkt(data, checksum)
  - udt_send(sndpkt)

- **Wait for call from above**

- **Wait for ACK or NAK**
  - rdt_rcv(rcvpkt) && isNAK(rcvpkt)
  - udt_send(sndpkt)

- **rdt_rcv(rcvpkt) && isACK(rcvpkt)**

- **rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)**
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)

- **Wait for call from below**

- **corrupt(rcvpkt)**
  - udt_send(NAK)

- **Λ**
rdt2.0: error scenario

- rdt_send(data)
  - snkpkt = make_pkt(data, checksum)
  - udt_send(sndpkt)

- Wait for call from above

- Wait for ACK or NAK

- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && isACK(rcvpkt)

- Λ

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)

- Wait for call from below

- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
  - udt_send(NAK)
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 garbled
- sender adds sequence number to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt
- Receiver sends Acks and NAKs.

stop and wait

Sender sends one packet, then waits for receiver response
rdt 2.1

- Bit errors but no packet loss (congestion).
- Sender and receiver have twice as many states as rdt 2.0.
  - Why?
rdt2.1: sender handles garbled ACK/NAKs

```
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
\Lambda
```

```
Wait for call 0 from above
```

```
rdt_recv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)

Wait for ACK or NAK 0
```

```
rdt_send(data)
```

```
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```

```
rdt_recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
\Lambda
```

```
Wait for call 1 from above
```

```
rdt_recv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
```

```
Wait for ACK or NAK 1
```

CMPE 150 Winter13
rdt2.1: receiver, handles garbled ACK/NAKs

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(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 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

**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
rtt3.0 sender

```
rtt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

Wait for call 0 from above

rtt_rcv(rcvpkt)
ζ

Wait for ACK0

rtt_send(data)

Wait for call 1 from above

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_send(data)

Wait for ACK1

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,1) )
ζ

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_send(data)

Wait for call 0 from above

rtt_rcv(rcvpkt)
ζ
udt_send(sndpkt)
start_timer

rtt_send(data)

Wait for call 1 from above

rtt_rcv(rcvpkt)
ζ
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

CMPE 150 Winter13
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[ d_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bps}} = 8 \text{ microseconds} \]

- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending

\[ U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027 \]

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
**rdt3.0: stop-and-wait operation**

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = \frac{L}{R}$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = \text{RTT} + \frac{L}{R}$

$$U_{\text{sender}} = \frac{\frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027$$