Announcements

Lab & Tutorial 2 – online

Mid-session test
• Email lecturer scheduling constraints this week.
• Will cover lectures thus far & Tanenbaum Ch 1-4.
• You can submit potential questions.
• Sample questions online ASAP.
Announcements

Summer work:

- Sep. 30: UNSW Taste Of Research
  http://www.eng.unsw.edu.au/current/scholar/tasteof.htm
- Aug. 31: ANU
  http://www.anu.edu.au/graduates/srs
- CSIRO
Data Link Layer
Lecture outline

Link layer services

Principles of reliable data transfer (RDT) Tanenbaum’s protocols

- Naïve retransmission (RDT 1.0)
- Stop-and-wait
  - Errors in payload (RDT 2.0)
  - Loss of acknowledgements (RDT 2.1)
  - No negative acknowledgements (RDT 2.2)
  - Loss of payload (RDT 3.0)
- Go-back-N
- Selective repeat

Duplex data transfer

Exemplary link layer protocols

- Protocol 2: Flow control
Basic link layer service

Link layer:
• uses the bit stream service provided by the Physical Layer
• to provide a frame transfer service to the network layer.

Links have no memory of what was transmitted
⇒ never mis-order or duplicate frames
⇒ If enhancing basic link layer service, want to preserve these characteristics.
Virtual communication, as perceived by network layer

Actual communication

Encapsulation of packet to achieve link layer service

Sending machine

Packet

Frame

Header Payload field Trailer

Receiving machine

Packet

Header Payload field Trailer

*Fig. 3-1 and 3-2*
Enhanced Link Layer services

May also enhance the “quality” of these frames on a link-by-link basis to improve system performance:

1. Flow control: Match tx rate to rx
   - Often omitted from the link layer
   - Notable exceptions: Old X.25, new Gigabit Ethernet

2. Error control
   - Retransmission-based recovery
   - Detect errors

3. Framing

Order of listing is important:

- Can implement in layered fashion:
  - Higher functions depend on lower ones (e.g. error control upon framing)
  - Can also intertwine the functions
    - (e.g. use redundancy for error detection to help framing; withhold error control acks to provide flow control)

- The higher ones are the most often omitted.

Desirability of these enhancements depends on the physical transmission media; e.g. tend to omit error control for reliable links (e.g. fibre) but emphasise it on unreliable links (e.g. wireless)

We’ve considered framing & error detection; now consider error recovery protocols. (Leave flow control for transport layer)
Outline
Reliable Data Transfer (RDT)

Goal: Ensure that frames or packets are delivered reliably: Everything gets through (no loss) with the correct value (integrity).

May be implemented in the:
- link layer (dealing with frames) or
- transport layer (dealing with packets/segments)

This discussion will be based on “packets” & using checksums as the integrity check.

Something to think about: The protocols that we’ll cover assume that the source is concerned about the reliability. Often it is the destination that cares, and will re-request the information if it doesn’t arrive reliably. e.g. web browsing: Client is the source of the URL, but the destination of the content. If the content doesn’t arrive, the client might re-request the information.
RDT: Context

send side

\texttt{rdt\_send()}: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

\texttt{udt\_send()}: called by \texttt{RDT}, to transfer packet over unreliable channel to receiver

receive side

\texttt{deliver\_data()}: called by \texttt{RDT} to deliver data to upper layer

\texttt{rdt\_rcv()}: called when packet arrives on rcv-side of channel

\texttt{packet}\hspace{1cm}\texttt{data}\hspace{1cm}\texttt{packet}\hspace{1cm}\texttt{data}

We’ll focus on \texttt{rdt\_send()} and \texttt{rdt\_rcv()} – functions that implement the RDT protocol.
RDT: Our approach

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

![State diagram]

**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 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

sender

receiver

Trivial; only slightly more complicated if sender doesn’t know receiver’s speed ⇒ need flow control.
Outline
RDT2.0: channel with bit errors

Underlying channel may flip bits in packet
  • recall: CRCs & checksums to detect bit errors

The question: how to recover from errors:
  • acknowledgements (ACKs): receiver explicitly tells sender that packet received OK
  • negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors†
    • sender retransmits packet on receipt of NAK
  • human scenarios using ACKs, NAKs?

New mechanisms in RDT2.0 (beyond RDT1.0):
  • error detection
  • Feedback (receiver->sender): control msgs (ACK,NAK)

† Assumes that receiver can tell that a packet was sent in the first place. e.g. if packets have sequence numbers, may detect loss by virtue of next received packet not having expected sequence number.
RDT2.0: FSM specification

sender

Wait for call from above

rdt_send(data)

sndpkt = make_pkt(data, checksum)

udt_send(sndpkt)

receiver

wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

Wait for call from below

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt, data)

deliver_data(data)

udt_send(ACK)
RDT2.0: operation with no errors

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

Wait for call from above

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

Wait for ACK or NAK

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

Wait for ACK or NAK

```
λ
```

Wait for call from below

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

udt_send(NAK)
RDT2.0: error scenario

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

Wait for call from above

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

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Lambda

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```
Stop-and-wait

RDT 2.0 is an example of the “Stop-and-wait” family of protocols:
1. Source sends a frame.
2. Source stops sending, and waits to receive feedback, before sending again.
5 (of many) possible scenarios

O = Original, A = Acknowledgement, R = Retransmission
O’, R’ = Original & Retransmission of next frame (may have identical content)

**Error-free**

O
A
R

**Original lost**

O
A
R

**Ack lost**

O
A
R

**O, ACK^R & R lost**

O
A
R

**2 originals lost**

O
A
R

**Round Trip Time**

- Source can’t distinguish these
  - ⇒ It will retransmit if ack is lost
  - ⇒ No need for destination to retransmit acks

- Receiver can’t distinguish these
  - Should discard R in 1\textsuperscript{st} case,
  - But keep R’ in 2\textsuperscript{nd} case.
  - ⇒ Sequence numbers

- Don’t want duplication @ rx ⇒ discard one
  - (Usually minimise delay by delivering original to higher layer as soon as received ⇒ discard R)
RDT2.0 has a fatal flaw!

What happens if feedback (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 packet!

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

stop and wait
Sender sends one packet, then waits for receiver response
RDT2.1: Sender

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

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

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

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

```
rdt_send(data)
```

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

```
rdt_send(data)
```

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

```
rdt_send(data)
```

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

```
rdt_send(data)
```

```
rdt_send(data)
```

```
rdt_send(data)
```
RDT2.1: Receiver

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

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

- `rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK, chksum)`
  - `udt_send(sndpkt)`

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

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

Packet 0

Packet 1
RDT2.1: discussion

**Sender:**
- seq # added to packet
- two seq. #’s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “current” packet has 0 or 1 seq. #

**Receiver:**
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected packet 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 packet received OK
  - receiver must *explicitly* include seq # of packet being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current packet*
RDT2.2: sender, receiver fragments

sender FSM fragment

\[ \text{sndpkt} = \text{makepkt}(0, \text{data}, \text{checksum}) \]
\[ \text{udt\_send}(\text{sndpkt}) \]

Wait for call 0 from above

Wait for ACK 0

\[ \text{udt\_send}(\text{sndpkt}) \]

\[ \text{rdt\_rcv}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt}, 1) \]

\[ \text{rdt\_rcv}(\text{rcvpkt}) \land \text{notcorrupt}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt}, 0) \]

receiver FSM fragment

\[ \text{udt\_send}(\text{sndpkt}) \]

Wait for 0 from below

\[ \text{rdt\_rcv}(\text{rcvpkt}) \land \text{notcorrupt}(\text{rcvpkt}) \land \text{has\_seq1}(\text{rcvpkt}) \]

\[ \text{extract}(\text{rcvpkt}, \text{data}) \]
\[ \text{deliver\_data}(\text{data}) \]
\[ \text{sndpkt} = \text{makepkt}(\text{ACK1, checksum}) \]
\[ \text{udt\_send}(\text{sndpkt}) \]
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 packet (or ACK) just delayed† (not lost):
  - retransmission will be duplicate, but use of seq. #’s already handles this
  - receiver must specify seq # of packet being ACKed
- requires countdown timer

† More common for transport protocols than link protocols
RDT3.0 sender

No need to resend; that will happen after timeout

Wait for call 0 from above

Wait for call 1 from above

packet 0

packet 1
RDT3.0 in action

(a) operation with no loss

(b) lost packet
RDT3.0 in action

(c) lost ACK

(d) premature timeout