Chapter 3
Transport Layer

A note on the use of these Powerpoint slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we’d like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

© J.F Kurose and K.W. Ross, All Rights Reserved

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016
Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
  • segment structure
  • reliable data transfer
  • flow control
  • connection management

3.6 principles of congestion control

3.7 TCP congestion control
Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

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

- important in application, transport, link layers
  - top-10 list of important networking topics!

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

- important in application, 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.

**Flow**
- **`rdt_send()`** sends data to `udt_send()`, which in turn transfers the packet over the unreliable channel to the receiver.
- The receiver then calls `rdt_rcv()` to handle the packet, which finally calls `deliver_data()` to deliver the data to the upper layer.
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

![Diagram of finite state machines](image)

**state:** when in this “state” next state uniquely determined by next event

**event causing state transition**

**actions taken on state transition**

**state 1**

**state 2**
**rtd 1.0: 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

Diagram:

sender

- Wait for call from above
- rdt_send(data)
- packet = make_pkt(data)
- udt_send(packet)

receiver

- Wait for call from below
- rdt_rcv(packet)
- extract (packet, data)
- deliver_data(data)
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:

How do humans recover from “errors” during conversation?
**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
**rdt2.0: FSM specification**

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

**Receiver**
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
  - `udt_send(sndpkt)`

**Flowchart**
- Wait for call from above
- Wait for ACK or NAK
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `Λ`

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

- `udt_send(NAK)`

---

*Transport Layer 3-11*
rdt2.0: operation with no errors

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && isNAK(rcvpkt)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && isACK(rcvpkt)
- udt_send(ACK)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
- extract(rcvpkt, data)
- deliver_data(data)
- udt_send(ACK)

Wait for call from above

Wait for ACK or NAK

Wait for call from below
rdt2.0: error scenario

```
rdt_send(data)

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

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

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

Wait for call from below

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

```c
switch (rcvpkt_type) {
    case ACK:
        // Process ACK
        rdt_send(data)
        break;
    case NAK:
        // Process NAK
        rdt_send(data)
        break;
    case DATA:
        // Process DATA
        rdt_send(data)
        break;
    default:
        // Handle unknown packet type
        break;
}
```
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)
  sndpkt = make_pkt(0, data, checksum)
  udt_send(sndpkt)
- rdt_recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
- Wait for call 0 from above
- rdt_recv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
  udt_send(sndpkt)
- Wait for ACK or NAK 0
- rdt_recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
- Wait for call 1 from above
- rdt_recv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
  udt_send(sndpkt)
- rdt_send(data)
  sndpkt = make_pkt(1, data, checksum)
  udt_send(sndpkt)
**rdt2.1: receiver, handles garbled ACK/NAKs**

- **Wait for 0 from below**
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK, chksum)`
  - `udt_send(sndpkt)`

- **Wait for 1 from below**
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK, chksum)`
  - `udt_send(sndpkt)`

- **Wait for 0 from below**
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt)`
    - `sndpkt = make_pkt(NAK, chksum)`
    - `udt_send(sndpkt)`

- **Wait for 1 from below**
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt)`
    - `sndpkt = make_pkt(NAK, chksum)`
    - `udt_send(sndpkt)`
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*
rdt2.2: sender, receiver fragments

sender FSM fragment

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

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

```
Wait for ACK 0
```

```
rdt_rcv(rcvpkt) &&
   ( corrupt(rcvpkt) ||
     isACK(rcvpkt,1) )
udt_send(sndpkt)
```

receiver FSM fragment

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

```
wait for 0 from below
```

```
rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
   && has_seq1(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```
new assumption: underlying channel can also lose packets (data, 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 seq. #’s already handles this
    • receiver must specify seq # of pkt being ACKed
  ▪ requires countdown timer
rdt3.0 sender

- \( \text{rdt\_rcv(rcvpkt)} \)
- \( \Lambda \)
- \( \text{rdt\_rcv(rcvpkt)} \) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)
- \( \text{stop\_timer} \)

Wait for call 0 from above
- \( \text{rdt\_send(data)} \)
- \( \text{sndpkt = make\_pkt(0, data, checksum)} \)
- \( \text{udt\_send(sndpkt)} \)
- \( \text{start\_timer} \)

Wait for ACK0
- \( \text{rdt\_rcv(rcvpkt)} \) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
- \( \text{stop\_timer} \)

Wait for call 1 from above
- \( \text{rdt\_rcv(rcvpkt)} \) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)

Wait for ACK1
- \( \text{rdt\_rcv(rcvpkt)} \) && ( corruprt(rcvpkt) || isACK(rcvpkt, 1) )
- \( \Lambda \)

timeout
- \( \text{udt\_send(sndpkt)} \)
- \( \text{start\_timer} \)

rdt\_rcv(rcvpkt)
- \( \Lambda \)
rdt3.0 in action

(a) no loss

(b) packet loss

Transport Layer 3-22
rdt3.0 in action

**sender**
- send pkt0
- rcv ack0
- send pkt1
- timeout
- resend pkt1
- rcv ack1
- send pkt0
- rcv pkt0
- send ack0
- rcv pkt1
- (detect duplicate)
- send ack1
- rcv pkt1
- send ack1

**receiver**
- send pkt0
- rcv pkt0
- ack0
- send ack0
- rcv pkt1
- send pkt1
- send ack1
- rcv ack1
- send pkt0
- rcv pkt0
- ack0
- send ack0
- rcv pkt0
- send ack0
- rcv pkt0
- (detect duplicate)

(c) ACK loss
(d) premature timeout/ delayed ACK
### Performance of rdt3.0

- **rdt3.0** is correct, but performance stinks
- **e.g.:** 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[
D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}
\]

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

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

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput 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, \( t = RTT + \frac{L}{R} \)

\[
U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.00027
\]
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
Pipelining: increased utilization

first packet bit transmitted, \( t = 0 \)
last bit transmitted, \( t = \frac{L}{R} \)

ACK arrives, send next packet, \( t = RTT + \frac{L}{R} \)

first packet bit arrives
last packet bit arrives, send ACK
last bit of 2\(^{nd}\) packet arrives, send ACK
last bit of 3\(^{rd}\) packet arrives, send ACK

\[ U_{\text{sender}} = \frac{3L / R}{RTT + \frac{L}{R}} = \frac{0.0024}{30.008} = 0.00081 \]

3-packet pipelining increases utilization by a factor of 3!