Transport Layer II

Dmitri Loguinov
Texas A&M University

March 1, 2016

Original slides copyright © 1996-2004 J.F Kurose and K.W. Ross
Chapter 3: Roadmap

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

- Characteristics of unreliable channel will determine complexity of **reliable data transfer (rdt)** protocol
Reliable Data Transfer: Getting Started

**send side**

- `udt_send()`: called by `rdt` to pass packets to lower layer

**receive side**

- `rdt_rcv()`: called by lower layer when it has a packet to deliver to `rdt`

- `deliver_data()`: called by `rdt` to deliver data to upper layer

**Examples**

- `rdt_send()`: called by layer above to pass data to `rdt`
- `udt_send()`: called by `rdt` to pass packets to lower layer
We will:

- 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 both sender and receiver

- From any state, the next state is uniquely determined by next event
Rdt1.0: Transfer Over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
  - No reordering

- Separate FSMs for sender and receiver:
  - Sender transmits app data into underlying channel
  - Receiver passes data from underlying channel to app
Rdt2.0: Channel With Bit Errors

• Underlying channel may flip bits in packet (no loss)
  – Checksum to detect bit errors (assume perfect detection)
• Question: how to recover from errors?
• One possible approach is to use two feedback msgs:
  – Positive acknowledgments (ACKs): receiver explicitly tells sender that packet was received OK
  – Negative acknowledgments (NAKs): receiver explicitly tells sender that packet had errors
  – Sender retransmits packet on receipt of NAK
• New mechanisms in rdt 2.0 (beyond rdt 1.0):
  – Error detection
  – Receiver feedback (control msgs ACK/NAK)
  – Retransmission
Rdt2.0: FSM Specification

**sender**

- rdt_send(data)
- sndpkt = make_pkt(data, checksum)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt)
  - udt_send(NAK)
- rdt_rcv(rcvpkt) AND isNAK(rcvpkt)
  - Wait for call from above
- rdt_rcv(rcvpkt) AND isACK(rcvpkt)
  - Wait for call from above

**receiver**

- rdt_rcv(rcvpkt) AND corrupt(rcvpkt)
  - udt_send(NAK)
- rdt_rcv(rcvpkt) AND isACK(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)

Λ = empty action, i.e., do nothing
Rdt2.0: Operation With No Errors

Wait for call from above

\[
\text{snkpkt} = \text{make_pkt(data, checksum)}
\]

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

\[
\text{rdt\_send(data)}
\]

Wait for ACK or NAK

\[
\text{rdt\_rcv(rcvpkt) && isNAK(rcvpkt)}
\]

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

Wait for ACK or NAK

\[
\text{rdt\_rcv(rcvpkt) && isACK(rcvpkt)}
\]

\[
\Lambda
\]

Wait for call from below

\[
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)}
\]

extract(rcvpkt, data)

deliver\_data(data)

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

\[
\text{udt\_send(NAK)}
\]
Any problems with this protocol?
Rdt2.0a: Handles Corrupted Feedback

```
rdt_send(data)

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

rdt_rcv(rcvpkt) AND
[isNAK(rcvpkt) OR
corrupt(rcvpkt)]
udt_send(sndpkt)

Wait for call from above
Wait for ACK or NAK
```

```
rdt_rcv(rcvpkt) AND isACK(rcvpkt)
AND NOT corrupt(rcvpkt)
Λ
```

```
receiver

rdt_rcv(rcvpkt) AND
corrupt(rcvpkt)
udt_send(NAK, checksum)

Wait for call from below
```

```
sender

rdt_send(data)

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

Any problems?
Rdt 2.0 and Rdt 2.0a Have Fatal Flaws

• Rdt 2.0 does not work when ACK/NAK is corrupted
  – Sender doesn’t know what happened at receiver!
• Rdt 2.0a delivers duplicate packets to application

Proper algorithm:
• Sender adds sequence number to each pkt
• Sender retransmits current pkt if ACK/NAK is garbled
• Receiver discards (doesn’t deliver up) duplicate pkt

Stop-and-Wait protocol: sender sends one packet, then waits for receiver’s response
Rdt2.1: Sender, Handles Garbled ACK/NAKs

\[
\begin{align*}
\text{rdt}_\text{send}(\text{data}) \\
\text{sndpkt} = \text{make}\_\text{pkt}(0, \text{data}, \text{checksum}) \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt}) \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isNAK}(\text{rcvpkt}) \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\text{rdt}_\text{rcv}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt}) \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\end{align*}
\]
Rdt2.1: Receiver, Handles Garbled ACK/NAKs

- rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt)
  - AND has_seq0(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) AND corrupt(rcvpkt)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND has_seq1(rcvpkt)
  - sndpkt = make_pkt(NAK, checksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) AND corrupt(rcvpkt)
  - sndpkt = make_pkt(NAK, 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
  - Protocol must remember whether current pkt has 0 or 1 sequence number

**Receiver:**
- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is the expected packet seq #
- **Note:** receiver cannot know if its last ACK/NAK was received correctly at sender
Rdt2.2: NAK-free Protocol

- Same functionality as rdt2.1, using ACKs only
  - Most protocols are easier to generalize without NAKs
- Instead of NAKs, receiver sends an ACK for last packet received correctly
  - 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

```
rdt_send(data)

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

Wait for call 0 from above
```

```
rdt_rcv(rcvpkt) AND
[corrupt(rcvpkt) OR
has_seq1(rcvpkt)]

sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)

Wait for ACK 0
```

```
rdt_rcv(rcvpkt) AND
[corrupt(rcvpkt) OR
isACK(rcvpkt,1)]
udt_send(sndpkt)

```

```
Send pkt 0
```

```
rddt recv(rcvpkt) AND
NOT corrupt(rcvpkt) AND
isACK(rcvpkt,0)

Wait for ACK 0
```

```
Lambda
```

```
rddt recv(rcvpkt) AND NOT corrupt(rcvpkt)
AND has_seq1(rcvpkt)

extract(rcvpkt, data)
```

```
deliver_data(data)
```

```
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```
Rdt3.0: Channels With Errors and Loss

- **New assumption:** underlying channel can also lose packets (data or ACKs)
  - Still no reordering
- Checksum, sequence numbers, ACKs, retransmissions will be of help, but **not enough**
- Why not?

- **Approach:** sender waits a "reasonable" amount of time for ACK
  - Retransmits if no ACK received in this time
  - Sender requires a timer

- If packet (or ACK) is just delayed (not lost):
  - Retransmission will be duplicate, but the use of seq. #'s already handles this
  - Receiver must specify seq # of packet being ACKed
Rdt3.0 Sender

Must not retransmit: ACK1 could be in response to a duplicate pkt1
Rdt3.0 in Action

(a) operation with no loss

(b) lost packet
**Rdt3.0 in Action**

- **sender**
  - send pkt0
  - rcv ACK0
  - send pkt1
  - timeout
    - resend pkt1

- **receiver**
  - pkt0
  - ACK0
  - rcv pkt0
    - send ACK0
  - pkt1
  - ACK1
  - rcv pkt1
    - send ACK1
  - pkt1
    - ACK1
    - rcv pkt1
      - send ACK1

(c) lost ACK

(d) premature timeout
Performance of Rdt3.0

• Rdt 3.0 works, but performance is low
• Example: 1 Gbps link, 15 ms end-to-end propagation delay, 1 KB packets, no loss or corruption:

\[ T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8 \text{ Kbits/pkt}}{10^9 \text{ bits/sec}} = 8 \text{ microsec} \]

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

• Server spends 0.008 ms being busy and 30 ms being idle, thus its link utilization is only 0.027%
• 1-KB pkt every 30 ms \(\rightarrow\) 264 Kbps thruput
• Network protocol limits use of physical resources!

Notation: KB = Kilobyte; Kbps = Kilobits/sec; Gbps = Gigabits/sec
Rdt3.0: Stop-and-wait Operation

First bit transmitted, \( t = 0 \)

Last bit transmitted, \( t = \frac{L}{R} \)

First packet bit arrives

Last packet bit arrives, ACK departs

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

\[
U_{\text{sender}} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027
\]
Performance of Rdt3.0

• Next assume that 10% of data packets are corrupted/lost (no loss in retransmissions or ACKs) and the timeout is 1 second
  - 90% of packets take \((\text{RTT} + \frac{L}{R}) \approx 30 \text{ ms}\) to complete, while 10% require \([\text{timeout} + \text{RTT} + 2\frac{L}{R}] \approx 1.03 \text{ sec}\)
  - Average per-packet delay \(0.9 \times 0.03 + 0.1 \times 1.03 \text{ sec} = 130 \text{ ms}\)
  - Average rate 7.7 pkts/s or 61.5 Kbps
• Rdt3.0 similar to HTTP 1.0 or non-pipelined HTTP 1.1
• Next time we’ll improve this using pipelining, which allows multiple unack’ed packets at any time
• Quiz #2 this Thursday: variation on Chapter 2 problems
  - P1, P3-P11, P13-P14, P20-P21