Transport Layer II

Dmitri Loguinov
Texas A&M University

February 26, 2019
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**

- **rdt_send()**: called by layer above to pass data to **rdt**

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

**reliable data transfer protocol (sending side)**

- **packet**

**unreliable channel**

**receive side**

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

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

**rdt_send()**: called by layer above to pass packets to lower layer

**rdt_rcv()**: called by lower layer when it has a packet to deliver to **rdt**
Reliable Data Transfer: Getting Started

We will:

• Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)

• Consider only unidirectional data transfer
  - With receiver feedback, packets travel in 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

sender

```
rdt_send(data)
packet = make_pkt(data)
udt_send(packet)
```

```
rdt_rcv(packet)
extract (packet, data)
deliver_data(data)
```

receiver
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**

\[ \text{rdt\_send}(\text{data}) \]
\[ \text{sndpkt} = \text{make\_pkt}(\text{data}, \text{checksum}) \]
\[ \text{udt\_send}(\text{sndpkt}) \]

\[ \text{rdt\_recv}(\text{rcvpkt}) \quad \text{AND} \quad \text{is\_NAK}(\text{rcvpkt}) \]
\[ \text{udt\_send}(\text{sndpkt}) \]

\[ \text{rdt\_recv}(\text{rcvpkt}) \quad \text{AND} \quad \text{is\_ACK}(\text{rcvpkt}) \]
\[ \text{λ} \]

\[ \text{λ} = \text{empty action, i.e., do nothing} \]

**receiver**

\[ \text{rdt\_recv}(\text{rcvpkt}) \quad \text{AND} \quad \text{corrupt}(\text{rcvpkt}) \]
\[ \text{udt\_send}(\text{NAK}) \]

\[ \text{Wait for call from below} \]

\[ \text{rdt\_recv}(\text{rcvpkt}) \quad \text{AND} \quad \text{NOT corrupt}(\text{rcvpkt}) \]
\[ \text{extract}(\text{rcvpkt}, \text{data}) \]
\[ \text{deliver\_data}(\text{data}) \]
\[ \text{udt\_send}(\text{ACK}) \]
Rdt2.0: Operation With No Errors

\[
\begin{align*}
\text{rdt\_send(data)} \\
\text{snkpkt = make\_pkt(data, checksum)} \\
\text{udt\_send(sndpkt)} \\
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && isNAK(rcvpkt)} \quad \text{udt\_send(sndpkt)} \\
\text{extract(rcvpkt, data)} \\
\text{deliver\_data(data)} \\
\text{udt\_send(ACK)} \\
\end{align*}
\]
Rdt2.0: Error Scenario

- `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(sndpkt)`
- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
- `udt_send(NAK)`
- `Wait for call from above`
- `Wait for ACK or NAK`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`

Any problems with this protocol?
Rdt2.0a: Handles Corrupted Feedback

sender

Any problems?

receiver

\[ \text{rdt\_send(data)} \]
\[ \text{snkpkt = make\_pkt(data, checksum)} \]
\[ \text{udt\_send(sndpkt)} \]

\[ \text{rdt\_rcv(rcvpkt) AND \{isNAK(rcvpkt) OR corrupt(rcvpkt)\}} \]
\[ \text{udt\_send(NAK, checksum)} \]

\[ \text{Wait for call from above} \]

\[ \text{Wait for ACK or NAK} \]

\[ \text{rdt\_rcv(rcvpkt) AND isACK(rcvpkt) AND NOT corrupt(rcvpkt)} \]

\[ \text{extract(rcvpkt, data)} \]
\[ \text{deliver\_data(data)} \]
\[ \text{udt\_send(ACK, checksum)} \]

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

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

\[ \text{udt\_send(sndpkt)} \]
Rdt2.0 and Rdt2.0a 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

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

\[
\text{rdt\_rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt})
\]

\[
\text{rdt\_rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isNAK}(\text{rcvpkt})
\]

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

\[
\text{rdt\_rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt})
\]

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

\[
\text{rdt\_rcv}(\text{rcvpkt}) \land \neg \text{corrupt}(\text{rcvpkt}) \land \text{isNAK}(\text{rcvpkt})
\]

\[
\text{udt\_send}(\text{sndpkt})
\]
Rdt2.1: Receiver, Handles Garbled ACK/NAKs

- **Wait for 0 from below**
  - `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)`
  - `sndpkt = make_pkt(NAK, checksum)
    udt_send(sndpkt)`

- **Wait for 1 from below**
  - `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(ACK, 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

sender FSM fragment

```
rdt_send(data)
```

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

```
wait for 0 from above
```

```
rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt,0)
```

```
Lambda
```

```
rdeadline(rcvpkt) AND [corrupt(rcvpkt) OR has_seq1(rcvpkt)]
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```

```
rdeadline(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt,1)
```

```
udt_send(sndpkt)
```

```
wait for ACK 0
```

```
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
```

receiver FSM fragment
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 delayed beyond the timer:
  - Retransmission will be duplicate, but the use of seq. #’s already handles this
  - Receiver must specify seq # of packet being ACKed
Rdt3.0 in Action (No Corruption)

Sender

0

ACK0

1

ACK1

0

ACK0

1

ACK1

Receiver

No loss

timeout

Sender

0

ACK0

1

X

0

ACK0

1

ACK1

0

ACK0

Forward loss
Rdt3.0 in Action (No Corruption)

sender

0

1

timeout

reverse loss

receiver

ACK0

ACK1

ACK0

1

X

0

1

ACK1

ACK1

sender

0

1

timeout

premature timeout

receiver

ACK0

ACK1

ACK1

ACK0

retx 0?
Rdt3.0 Sender

Must not retx: ACK1 may be from a premature timeout on pkt1

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

\[
\text{sndpkt} = \text{make\_pkt}(0, \text{data}, \text{checksum})
\]

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

\[
\text{start\_timer}
\]

Wait for call 0 from above

Wait for call 1 from above

Wait for ACK0

Wait for ACK1

timeout

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

\[
\text{start\_timer}
\]

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

\[
\Lambda
\]

\[
\text{rdt\_rcv(rcvpkt)} \text{ AND NOT corrupt(rcvpkt)} \text{ AND isACK(rcvpkt,1)}
\]

\[
\text{stop\_timer}
\]

\[
\text{rdt\_rcv(rcvpkt)} \text{ AND NOT corrupt(rcvpkt)} \text{ AND isACK(rcvpkt,0)}
\]

\[
\text{stop\_timer}
\]

\[
\text{rdt\_rcv(rcvpkt)} \text{ AND (corrupt(rcvpkt)} \text{ OR isACK(rcvpkt,0)}
\]
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:

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

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

- 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 → 264 Kbps throughput
- *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,** $t = L / R$
- **first packet bit arrives**
- **last packet bit arrives, ACK departs**
- **ACK arrives, send next packet,** $t = RTT + L / R$

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027$$
**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 \(7.7 \text{ pkts/s} \text{ or} \ 61.5 \text{ Kbps}\)

- 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: chapter 2 problems and systems notes
  - P1, P3-P11, P13-P14, P20-P21