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

**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**

**Diagram**
- **rdt_send()**
  - Reliable data transfer protocol (sending side)
  - Udt_send() (packet)
- **.udt_send()**
  - Unreliable channel
- **deliver_data()**
  - Reliable data transfer protocol (receiving side)
  - Rdt_rcv() (packet)
**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

![Diagram](image-url)
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

\[
\text{sender} \quad \text{receiver}
\]

\[
\text{rdt\_send(data)} \\
\text{packet = make\_pkt(data)} \\
\text{udt\_send(packet)} \\
\text{Wait for call from above}
\]

\[
\text{rdt\_rcv(packet)} \\
\text{extract (packet, data)} \\
\text{deliver\_data(data)} \\
\text{Wait for call from below}
\]
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**

- \( rdt_{send}(data) \)
- \( sndpkt = make\_pkt(data, checksum) \)
- \( udt\_send(sndpkt) \)

\( \Lambda \)

**Receiver**

- \( rdt\_rcv(rcvpkt) \)
- \( \Lambda \)
- \( \Lambda = empty\ action, \ i.e.,\ do\ nothing \)

- \( rdt\_rcv(rcvpkt) \)
- \( \text{extract}(rcvpkt, data) \)
- \( \text{deliver\_data}(data) \)
- \( udt\_send(ACK) \)

- \( udt\_send(NAK) \)

- \( rdt\_rcv(rcvpkt) \)
- \( rdt\_rcv(rcvpkt) \)
- \( rdt\_rcv(rcvpkt) \)
- \( \text{corrupt}(rcvpkt) \)
- \( \text{isNAK}(rcvpkt) \)
- \( \text{isACK}(rcvpkt) \)
- \( \Lambda \)

\( \Lambda \)
Rdt2.0: Operation With No Errors

- rdt_send(data)
- snkpkt = make_pkt(data, checksum)
- udt_send(sndpkt)
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
- udt_send(NAK)
- rdt_send(data)
- Wait for call from above

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

- rdt_rcv(rcvpkt) && isACK(rcvpkt)
- deliver_data(data)
- udt_send(ACK)
- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
- udt_send(NAK)
- Wait for call from below

- extract(rcvpkt, data)
Rdt2.0: Error Scenario

```
# Include rdt2.h

int rdt_send(int dest, int data, int checksum)
{
    struct snkpkt
    { int dest; int data; int checksum; };

    snkpkt = make_pkt(data, checksum);
    udt_send(sndpkt);
    return 0;
}
```

```
int rdt_rcv(struct rcpkt
    { int src; int dest; int data; int checksum; })
{
    if (src == dest)
    {
        if (notcorrupt(rcvpkt))
        {
            extract(rcvpkt, data);
            deliver_data(data);
            udt_send(ACK);
            return 0;
        }
    }
    return -1;
}
```

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

sender

rdt_send(data)

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

udt_send(sndpkt)

receiver

\[
\text{rdt_rcv(rcvpkt) \ AND \ notcorrupt(rcvpkt)}
\]

\[
\text{udt_send(ACK, checksum)}
\]

rdt_rcv(rcvpkt) AND isACK(rcvpkt)

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

\[
\Lambda
\]

Any problems?
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

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

Wait for call 0 from above

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

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

Wait for ACK or NAK 0

Wait for ACK or NAK 1

Wait for call 1 from above

\[
\text{rdt\_send(data)} \\
\text{sndpkt = make\_pkt(1, data, checksum)} \\
\text{udt\_send(sndpkt)} \\
\]

\[
\text{rdt\_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt)} \\
\Lambda \\
\]
Rdt2.1: Receiver, Handles Garbled ACK/NAKs

Wait for 0 from below

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \) AND NOT corrupt(\text{rcvpkt})
- AND has_seq0(\text{rcvpkt})
- \( \text{extract}(\text{rcvpkt}, \text{data}) \)
- deliver_data(\text{data})
- \( \text{sndpkt} = \text{make}_\text{pkt}(\text{ACK, chksum}) \)
- udt_send(\text{sndpkt})

Wait for 1 from below

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \) AND corrupt(\text{rcvpkt})
- \( \text{sndpkt} = \text{make}_\text{pkt}(\text{NAK, chksum}) \)
- udt_send(\text{sndpkt})

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \) AND NOT corrupt(\text{rcvpkt}) AND has_seq1(\text{rcvpkt})
- \( \text{sndpkt} = \text{make}_\text{pkt}(\text{ACK, chksum}) \)
- udt_send(\text{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)`

**Receiver FSM Fragment**

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

**States and Conditions**

- **Wait for call 0 from above**
- **Wait for ACK 0**
- **Wait for 0 from below**

**Transitions**

- `rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt,0)`
  `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) AND [corrupt(rcvpkt) OR has_seq1(rcvpkt)]`  
  `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) AND NOT corrupt(rcvpkt) AND isACK(rcvpkt,1)`  
  `udt_send(sndpkt)`

**Additional Actions**

- `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 data packets 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
- Redundant retransmission is now possible
  - ACK is lost or arrives after the timer expires
  - The use of seq. #'s in data pkts and ACKs already handles this
Rdt3.0 in Action (No Corruption)

sender  
0  ➔  ACK0
1  ➔  ACK1
0  ➔  ACK0
1  ➔  ACK1

no loss

receiver

sender  
0  ➔  ACK0
1  ➔  ACK1
1  ➔  ACK1

timeout

receiver

forward loss
Rdt3.0 in Action (No Corruption)

sender

receiver

sender

receiver

0

ACK0

1

ACK1

1

ACK1

0

ACK0

timeout

reverse loss

sender

receiver

sender

receiver

0

ACK0

1

ACK1

1

ACK1

0

ACK0

timeout

reverse loss

premature timeout
Rdt3.0 Sender

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

\[ \text{rtt}_\text{send}(\text{data}) \]
\[ \text{sndpkt} = \text{make}_\text{pkt}(0, \text{data}, \text{checksum}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \Lambda \]

Wait for call 0 from above

\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]

Wait for ACK0

\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]

Wait for ACK1

\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]

\[ \text{rtt}_\text{send}(\text{data}) \]
\[ \text{sndpkt} = \text{make}_\text{pkt}(1, \text{data}, \text{checksum}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]

Wait for call 1 from above

\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]

\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]

\[ \text{rtt}_\text{rcv}(\text{rcvpkt}) \]
\[ \text{udt}_\text{send}(\text{sndpkt}) \]
\[ \text{start}_\text{timer} \]
\[ \Lambda \]
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:

\[
T_{\text{transmit}} = \frac{L}{R} = \frac{8 \text{ Kbits/pkt}}{10^9 \text{ bits/sec}} = 8 \text{ microsec}
\]

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

• 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 → 264 Kbps throughput

• **Network protocol limits use of physical resources!**
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, $t = RTT + L / R$

Sender

Receiver

$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027$
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 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: chapter 2 problems and system notes
  – P1, P3-P11, P13-P14, P20-P21