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)

  (a) provided service
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)

(a) provided service
(b) service implementation

Transport Layer 3-4
Reliable data transfer: getting started

**send side**

- **rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

**receive side**

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

- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

- **rdt_rcv()**: called when packet arrives on rcv-side of channel
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

state: when in this “state” next state uniquely determined by next event
**rdt1.0: 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

```plaintext
wait for call from above

Wait for call from above

Wait for call from below

sender

receiver
```

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

- `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

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

\[\text{Wait for call from above}\]
\[\text{rdt\_rcv}(\text{rcvpkt}) \land \text{isACK}(\text{rcvpkt})\]

\[\text{receiver}\]

\[\text{rdt\_rcv}(\text{rcvpkt}) \land \text{isNAK}(\text{rcvpkt})\]
\[\text{udt\_send}(\text{sndpkt})\]

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

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

Transport Layer 3-10
rdt2.0: operation with no errors

\[
\begin{align*}
\text{rdt}\_send(\text{data}) \\
\text{sndpkt} &= \text{make}\_ pkt(\text{data, checksum}) \\
\text{udt}\_send(\text{sndpkt}) \\
\text{rdt}\_rcv(\text{rcvpkt}) && \text{isACK}(\text{rcvpkt}) \\
\text{udt}\_send(\text{sndpkt}) \\
\text{rdt}\_rcv(\text{rcvpkt}) && \text{isNAK}(\text{rcvpkt}) \\
\text{udt}\_send(\text{sndpkt}) \\
\text{rdt}\_rcv(\text{rcvpkt}) && \text{corrupt}(\text{rcvpkt}) \\
\text{udt}\_send(\text{NAK}) \\
\text{Wait for call from above} \\
\text{Wait for ACK or NAK} \\
\text{Wait for call from below} \\
\Lambda \\
\text{extract}(\text{rcvpkt, data}) \\
\text{deliver}\_\text{data}(\text{data}) \\
\text{udt}\_send(\text{ACK})
\end{align*}
\]
rdt2.0: error scenario

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

- Wait for call from above

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

- rdt_rcv(rcvpkt) &&
  isACK(rcvpkt)

- Wait for call from below

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

- Wait for ACK or NAK

- rdt_rcv(rcvpkt) &&
  corrupt(rcvpkt)
  udt_send(NAK)
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

\[
\begin{align*}
\text{rdt\_send(data)} \\
\text{sndpkt} = \text{make\_pkt}(0, \text{data}, \text{checksum}) \\
\text{udt\_send(sndpkt)} \\
\text{Wait for call 0 from above} \\
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)} \\
\Lambda \\
\text{rdt\_rcv(rcvpkt) && ( corrupt(rcvpkt) || isNAK(rcvpkt) )} \\
\text{udt\_send(sndpkt)} \\
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)} \\
\Lambda \\
\text{Wait for ACK or NAK 0} \\
\text{rdt\_send(data)} \\
\text{sndpkt} = \text{make\_pkt}(1, \text{data}, \text{checksum}) \\
\text{udt\_send(sndpkt)} \\
\text{Wait for call 1 from above} \\
\text{rdt\_rcv(rcvpkt) && ( corrupt(rcvpkt) || isNAK(rcvpkt) )} \\
\text{udt\_send(sndpkt)}
\end{align*}
\]
rdt2.1: receiver, handles garbled ACK/NAKs

- \( rdt_{rcv}(rcvpkt) \) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- \( rdt_{rcv}(rcvpkt) \) && (corrupt(rcvpkt)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- \( rdt_{rcv}(rcvpkt) \) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- \( rdt_{rcv}(rcvpkt) \) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- \( rdt_{rcv}(rcvpkt) \) && (corrupt(rcvpkt)
  - sndpkt = make_pkt(ACK, checksum)
  - udt_send(sndpkt)

- \( rdt_{rcv}(rcvpkt) \) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
  - sndpkt = make_pkt(ACK, 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
  - 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
**rdt3.0: channels with errors and loss**

**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**

```
## State Machine Diagram

**Transition Diagram:**
- **Wait for call 0 from above**
  - `rdt_rcv(rcvpkt)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `timeout` (leads to `Wait for ACK0`)
  - `.udt_send(sndpkt)`
  - `start_timer`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`

**Wait for ACK0**
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))`
- `Lambda`
- `udt_send(sndpkt)`
- `start_timer`
- `timeout` (leads to `Wait for call 1 from above`)
- `rdt_rcv(rcvpkt)`
- `&& notcorrupt(rcvpkt)`
- `&& isACK(rcvpkt, 1)`
- `stop_timer`

**Wait for call 1 from above**
- `rdt_rcv(rcvpkt)`
- `Lambda`
- `rdt_send(data)`
- `sndpkt = make_pkt(1, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `timeout` (leads to `Wait for ACK1`)
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `&& isACK(rcvpkt, 0)`
- `stop_timer`

**Wait for ACK1**
- `rdt_rcv(rcvpkt)`
- `Lambda`
- `rdt_send(data)`
- `sndpkt = make_pkt(1, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `timeout` (leads to `Wait for call 0 from above`)
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_rcv(rcvpkt)`
- `&& notcorrupt(rcvpkt)`
- `&& isACK(rcvpkt, 0)`
- `stop_timer`
```
(a) no loss

The diagram shows the process of transferring data packets using the RDT3.0 protocol. The protocol involves the exchange of packets and acknowledgments between the sender and receiver. In the case of no packet loss, the packets are successfully acknowledged, and the process continues.

(b) packet loss

However, in the case of packet loss, a timeout occurs, and the packet is resent. This ensures that all packets are delivered successfully.
rdt3.0 in action

sender

send pkt0

rcv pkt0

send ack0

rcv pkt0

send ack0

(c) ACK loss

resend pkt1

receiver

pkt0

ack0

send ack1

pkt1

recipient

ack1

send ack1

ack0

send ack0

ack1

send ack0

(d) premature timeout/ delayed ACK

Transport Layer 3-20
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

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link

- network protocol limits use of physical resources!
rdr3.0: stop-and-wait operation

sender

first packet bit transmitted, $t = 0$

last packet bit transmitted, $t = L / R$

receiver

RTT

first packet bit arrives, send ACK

ACK arrives, send next packet, $t = RTT + L / R$

last packet bit arrives, send ACK
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**

![Diagram of pipelined protocols](image)
Pipelining: increased utilization

\[ U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081 \]
Pipelined protocols: overview

**Go-back-N:**
- sender can have up to \( N \) unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

**Selective Repeat:**
- sender can have up to \( N \) unack’ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet
Go-Back-N: sender

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed
- \text{send\_base} \rightarrow \text{nextseqnum}

- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window
GBN: sender extended FSM

\[
\begin{align*}
\text{send} & \text{packet(s) to receiver} \\
\text{wait for acknowledgement} \\
\text{if acknowledgement received, do} & \\
\begin{cases}
\text{update base,} & \text{if base} \neq \text{nextseqnum} \\
\text{reset timeout,} & \text{if base} = \text{nextseqnum}
\end{cases}
\end{align*}
\]

- \text{udt\_send(sndpkt[base])}
- \text{udt\_send(sndpkt[base+1])}
- \text{...}
- \text{udt\_send(sndpkt[nextseqnum-1])}

\text{timeout} \rightarrow \text{start\_timer}
\text{udt\_send(sndpkt[base])}
\text{udt\_send(sndpkt[base+1])}
\text{...}
\text{udt\_send(sndpkt[nextseqnum-1])}

\text{refuse\_data(data)}

\text{base} = \text{getacknum(rcvpkt)} + 1
\text{if (base} \neq \text{nextseqnum)}
\begin{cases}
\text{stop\_timer,} & \text{if base} = \text{nextseqnum} \\
\text{start\_timer,} & \text{else}
\end{cases}
\text{else}

\text{start\_timer}

\text{udt\_send(sndpkt[base])}
\text{udt\_send(sndpkt[base+1])}
\text{...}
\text{udt\_send(sndpkt[nextseqnum-1])}

\text{notcorrupt(rcvpkt)}

\text{base} = \text{getacknum(rcvpkt)} + 1
\text{if (base} \neq \text{nextseqnum)}
\begin{cases}
\text{stop\_timer,} & \text{if base} = \text{nextseqnum} \\
\text{start\_timer,} & \text{else}
\end{cases}
\text{else}

\text{start\_timer}

\text{udt\_send(sndpkt[base])}
\text{udt\_send(sndpkt[base+1])}
\text{...}
\text{udt\_send(sndpkt[nextseqnum-1])}

\text{notcorrupt(rcvpkt)}

\text{base} = \text{getacknum(rcvpkt)} + 1
\text{if (base} \neq \text{nextseqnum)}
\begin{cases}
\text{stop\_timer,} & \text{if base} = \text{nextseqnum} \\
\text{start\_timer,} & \text{else}
\end{cases}
\text{else}

\text{start\_timer}

\text{udt\_send(sndpkt[base])}
\text{udt\_send(sndpkt[base+1])}
\text{...}
\text{udt\_send(sndpkt[nextseqnum-1])}

\text{notcorrupt(rcvpkt)}
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember `expectedseqnum`

- out-of-order pkt:
  - discard (don’t buffer): *no receiver buffering*
  - re-ACK pkt with highest in-order seq #
GBN in action

sender window (N=4)

sender

receiver

send pkt0
send pkt1
send pkt2
send pkt3

(re)send ack1

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1

rcv ack0, send pkt4
rcv ack1, send pkt5

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

ignore duplicate ACK

(pkt 2 timeout)

Transport Layer 3-29
Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - $N$ consecutive seq #'s
  - limits seq #'s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

**sender**

- data from above:
  - if next available seq # in window, send pkt
- **timeout(n):**
  - resend pkt n, restart timer
- **ACK(n) in [sendbase, sendbase+N-1]:**
  - mark pkt n as received
  - if n smallest unACKed pkt, advance window base to next unACKed seq #

**receiver**

- **pkt n in [rcvbase, rcvbase+N-1]:**
  - send ACK(n)
  - out-of-order: buffer
  - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- **pkt n in [rcvbase-N,rcvbase-1]:**
  - ACK(n)
- otherwise:
  - ignore
Selective repeat in action

**sender window (N=4)**

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>send pkt0</td>
<td>receive pkt0, send ack0</td>
</tr>
<tr>
<td>send pkt1</td>
<td>receive pkt1, send ack1</td>
</tr>
<tr>
<td>send pkt2</td>
<td>receive pkt3, buffer, send ack3</td>
</tr>
<tr>
<td>(wait)</td>
<td>receive pkt4, buffer, send ack4</td>
</tr>
<tr>
<td>send pkt3</td>
<td>receive pkt5, buffer, send ack5</td>
</tr>
<tr>
<td>rcv ack0, send pkt4</td>
<td>rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2</td>
</tr>
<tr>
<td>rcv ack1, send pkt5</td>
<td></td>
</tr>
</tbody>
</table>

**Q: what happens when ack2 arrives?**