DATA COMMUNICATOIN NETWORKING

Instructor: Ouldooz Baghban Karimi

Course Book: Computer Networking, A Top-Down Approach
By: Kurose, Ross

Course Overview

- **Basics of Computer Networks**
  - Internet & Protocol Stack
  - Application Layer
  - Transport Layer
  - Network Layer
  - Data Link Layer
- **Advanced Topics**
  - Case Studies of Computer Networks
  - Internet Applications
  - Network Management
  - Network Security

RDT 2.0 : Channel with bit errors

- Unreliable underlying channel
  - Flipped bits in the packet
  - Checksum to detect bit errors

- How to recover from errors?
  - Acknowledgements (ACKs)
    - Receiver explicitly tells sender packet received OK

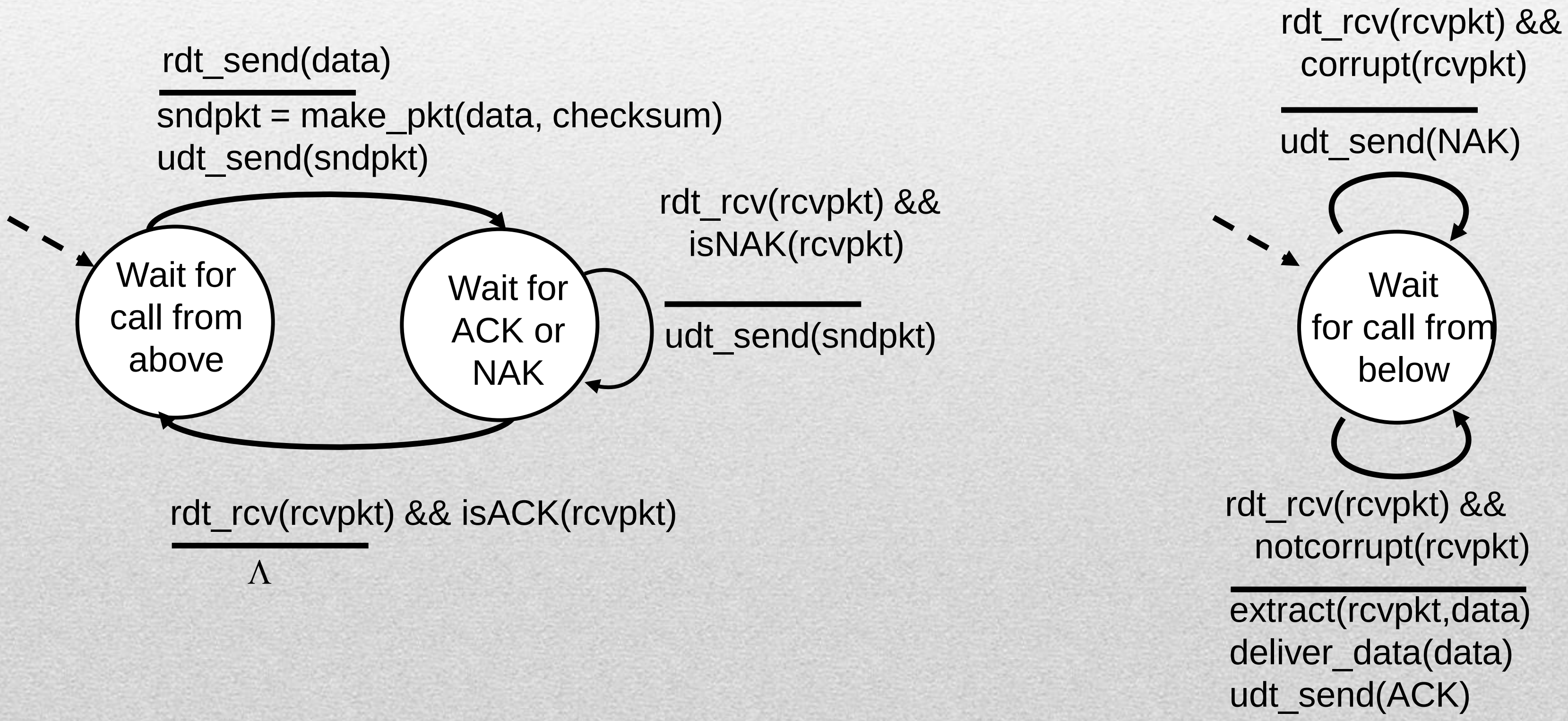
  - Negative acknowledgements (NAKs):
    - Receiver explicitly tells sender packet had errors
    - Sender retransmits packet on receipt of NAK

- New mechanisms in **rdt2.0** (beyond **rdt1.0**)
  - Error detection
  - Receiver feedback: control messages (ACK,NAK) receiver → sender

RDT 2.0 : FSM Specification

sender

receiver



rdt_send(data)

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

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

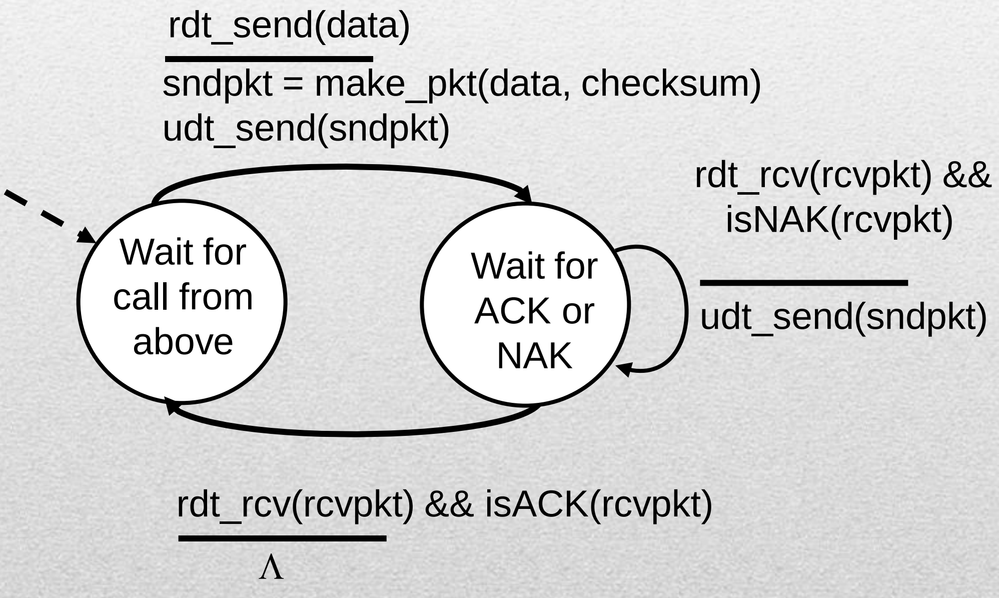
Wait for call from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

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

RDT 2.0 : FSM Specification

sender



rdt_send(data)

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

Wait for call from above

Wait for ACK or NAK

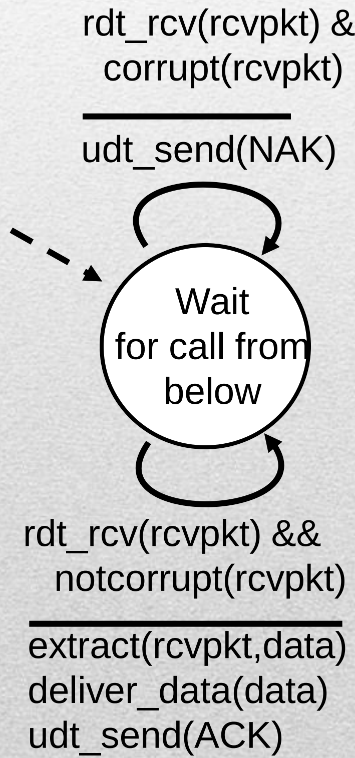
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

receiver



rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

Wait for call from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

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

RDT 2.0 : A Fatal Flaw!

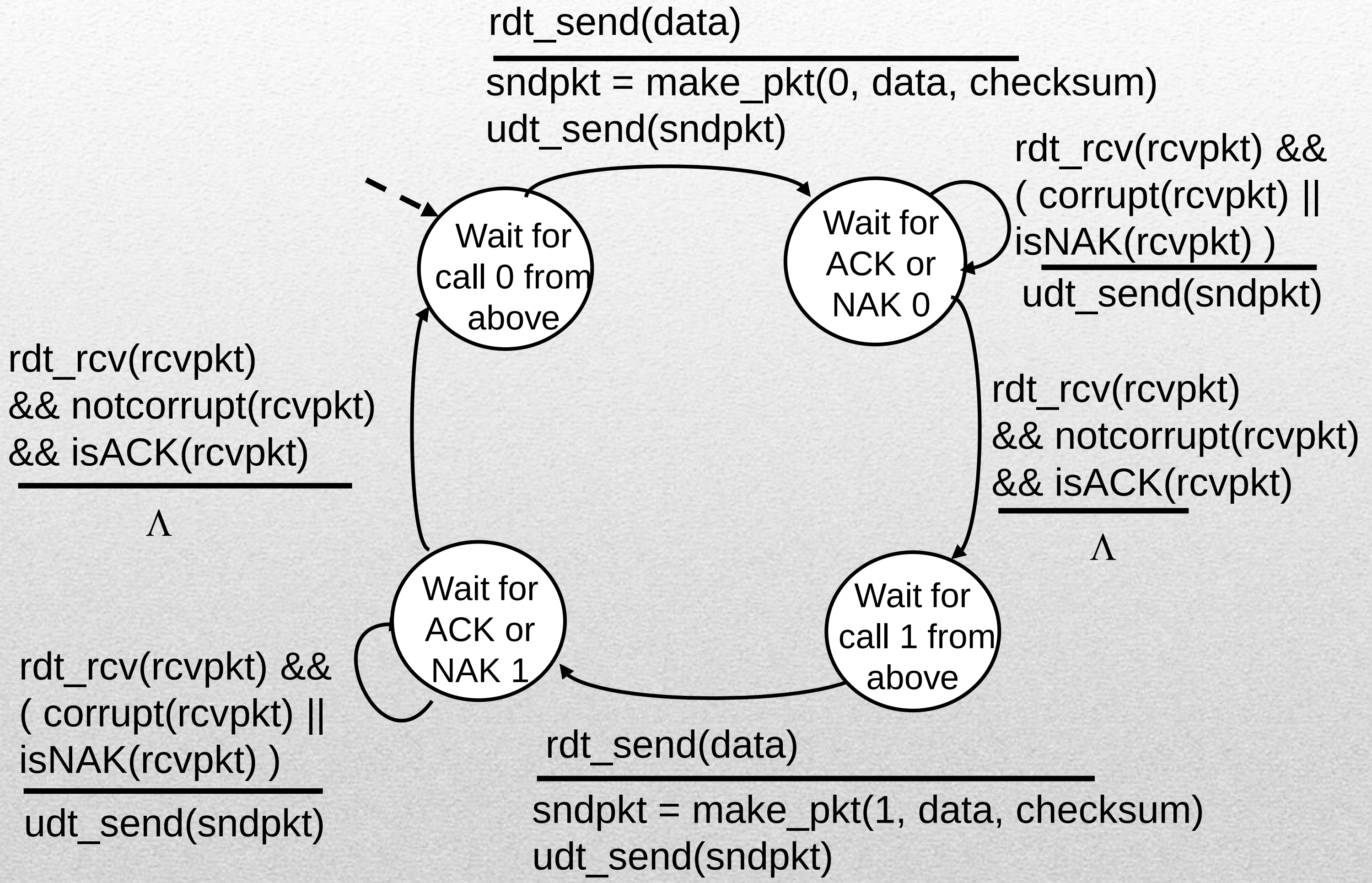
- Corrupted ACK/NAK
  - Sender does not know what happened at receiver!
  - Can't just retransmit: possible duplicate

- Duplicates
  - Sender retransmits current packet if ACK/NAK corrupted
  - Sender adds *sequence number* to each packet
  - Receiver discards (does not deliver up) duplicate packet

Stop & Wait

sender sends one packet, then waits for receiver response

RDT 2.1 : Garbled ACK/NACK-Sender



Wait for call 0 from above

Wait for ACK or NAK 0

Wait for call 1 from above

Wait for ACK or NAK 1

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

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

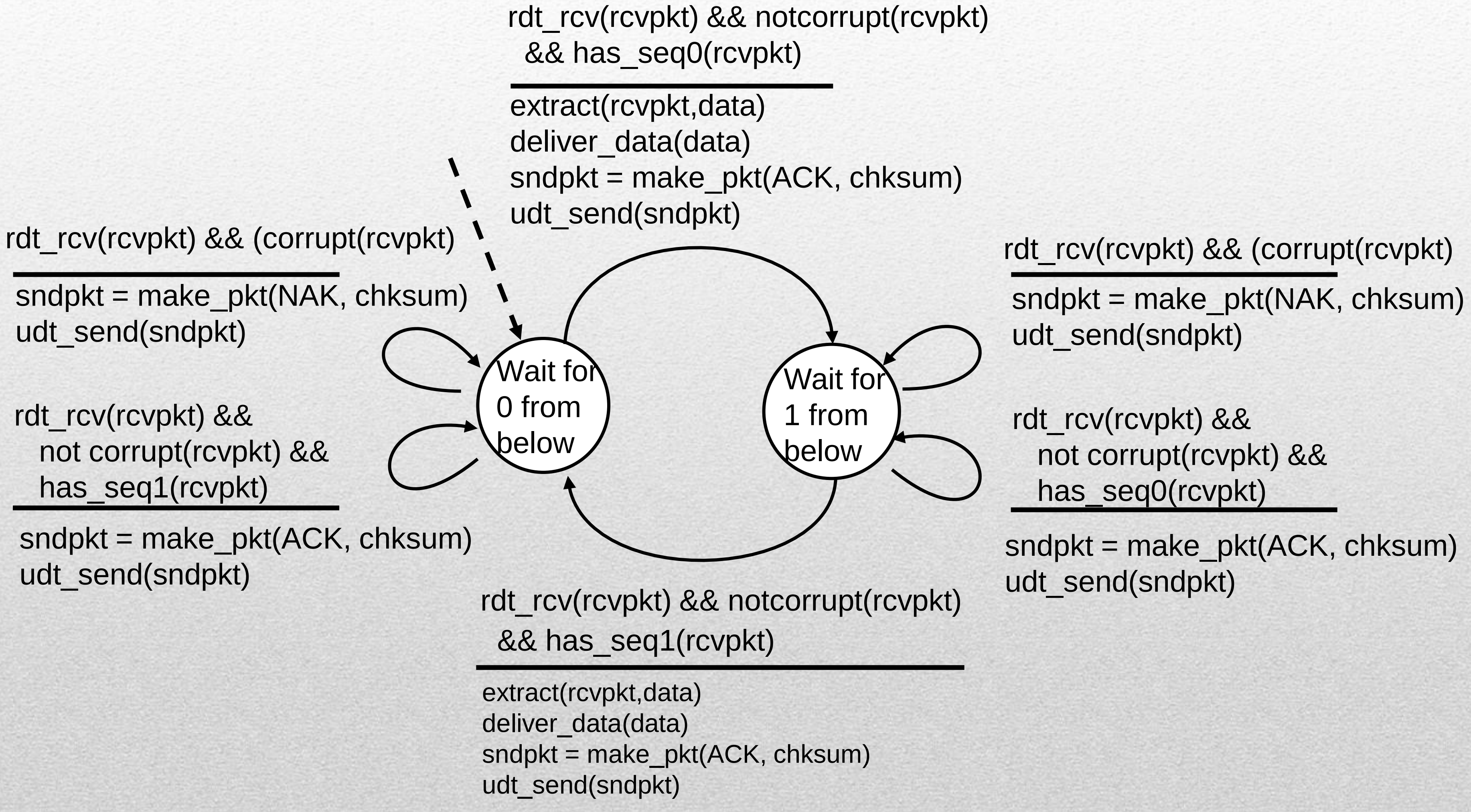
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ

rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

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

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ

RDT 2.1 : Garbled ACK/NACK-Receiver



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Wait for
0 from
below

Wait for
1 from
below

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

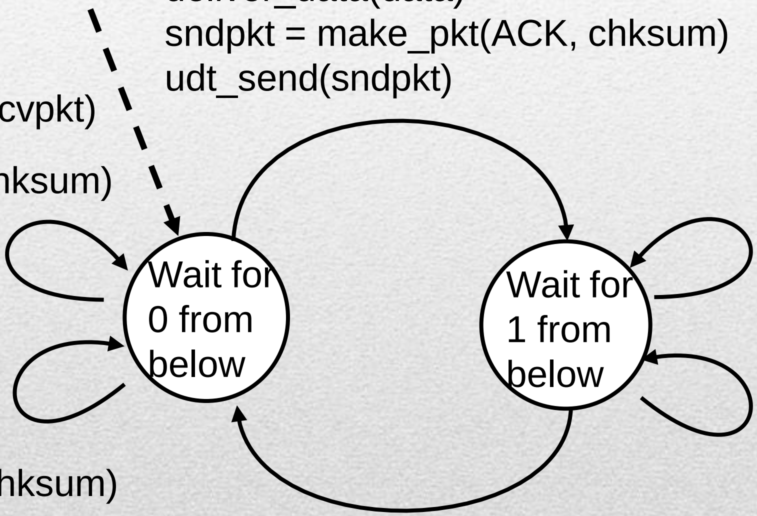
rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

RDT 2.1 : Garbled ACK/NACK-Receiver



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

RDT 2.1 : Discussion

Sender

- Sequence number added to packet
- Two sequence numbers (0,1) will suffice. Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - state must "remember" whether "expected" packet should have sequence number of 0 or 1

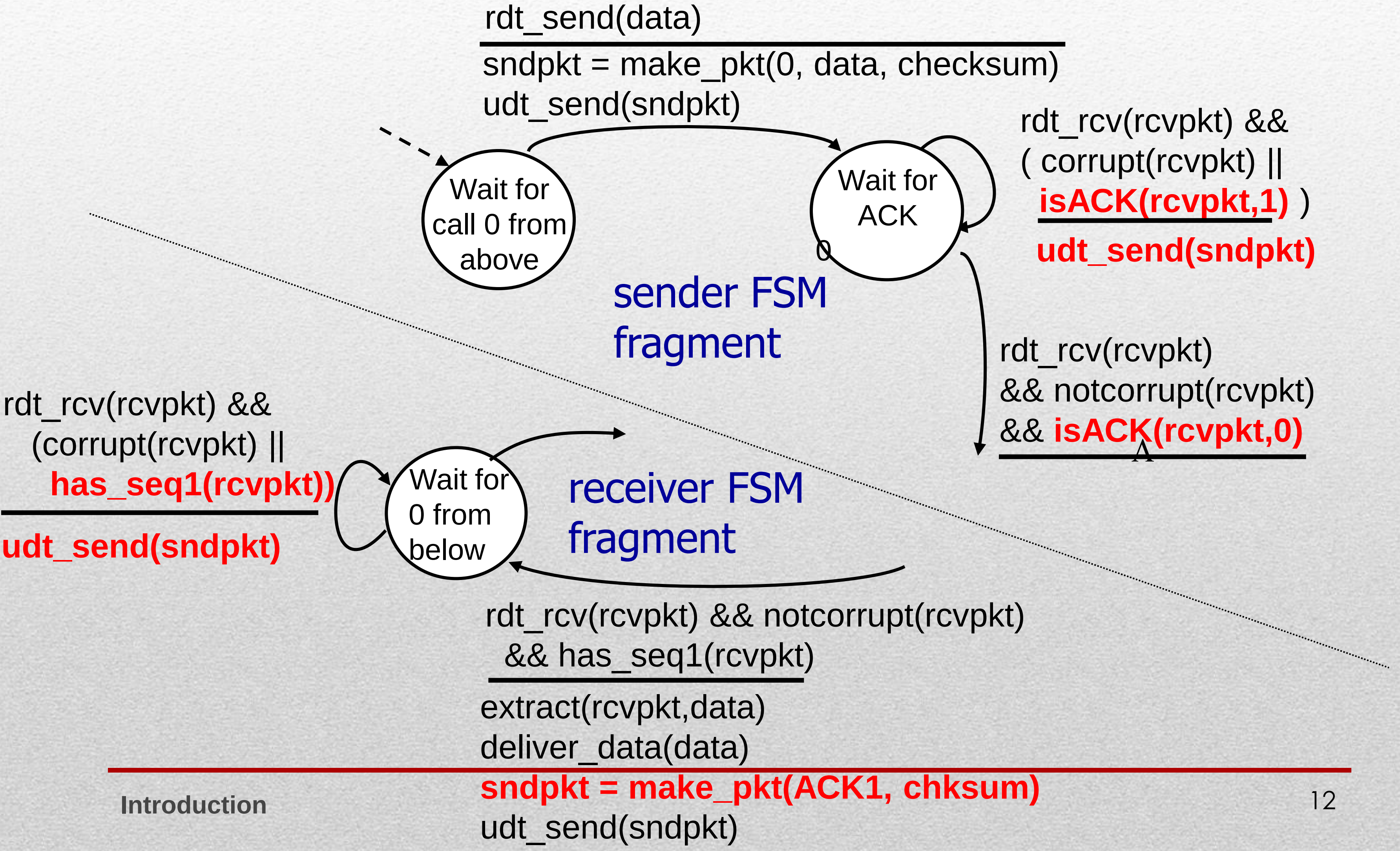
Receiver

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected packet sequence number
- Note: receiver can *not* know if its last ACK/NAK received OK at sender

RDT 2.2 : NACK-Free!

- Same functionality as RDT2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last packet received OK
  - Receiver must *explicitly* include sequence number of packet being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current packet*

RDT 2.2 : Sender, Receiver, Fragments



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)

sender FSM fragment

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

Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))

udt_send(sndpkt)

Wait for 0 from below

receiver FSM fragment

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

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

RDT 3.0 : Channels with Errors & Loss

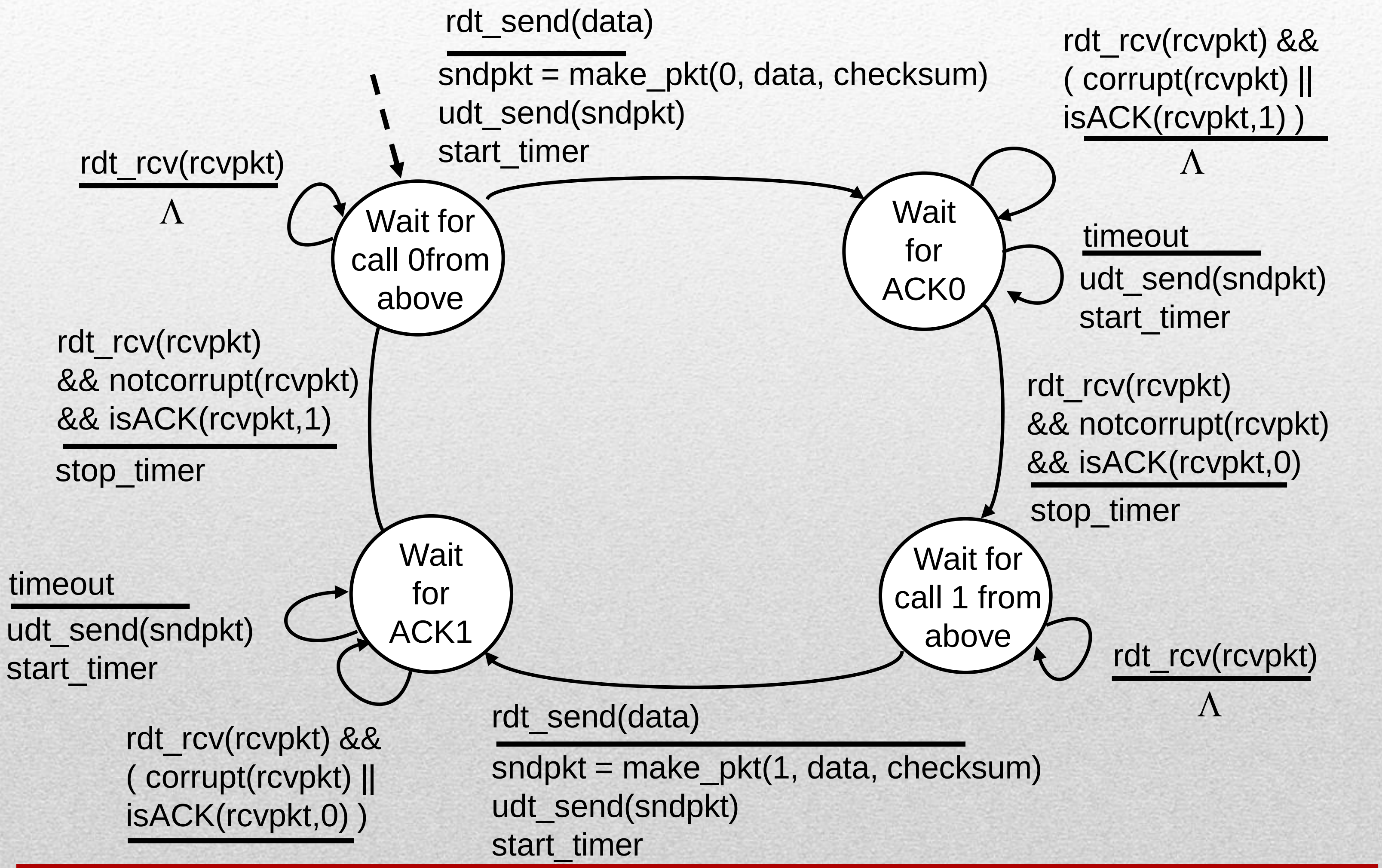
New Assumption

- Underlying channel can also lose packets (data, ACKs)
- checksum, sequence numbers, 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 packet (or ACK) just delayed (not lost)
  - Retransmission will be duplicate, but sequence numbers already handles this
  - Receiver must specify sequence number of packet being ACKed
- Requires countdown timer

RDT 3.0 : Sender



rdt_send(data)

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

rdt_rcv(rcvpkt)

Λ

Wait for call 0from above

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))

Λ

Wait for ACK0

timeout

udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)

stop_timer

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

stop_timer

Wait for ACK1

timeout

udt_send(sndpkt)
start_timer

Wait for call 1 from above

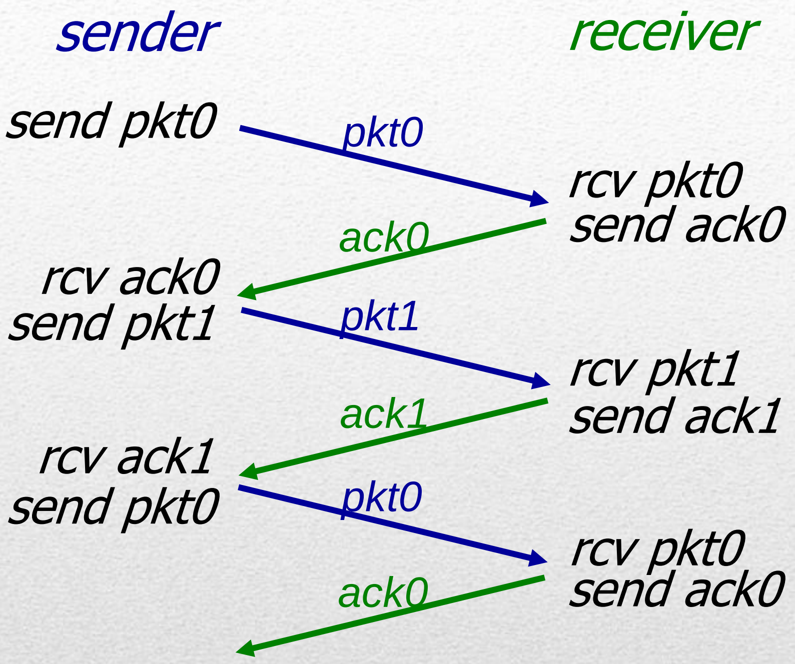
rdt_rcv(rcvpkt)

Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

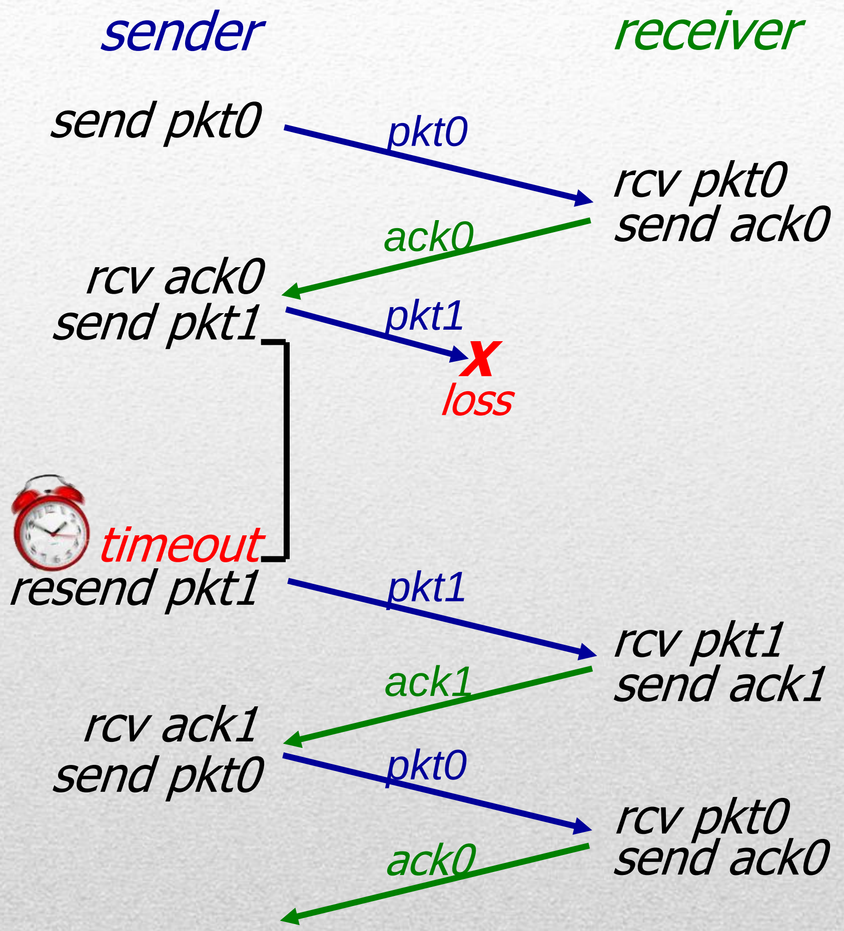
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(a) no loss



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

X
loss

timeout
resend pkt1

pkt1

rcv pkt1
send ack1

ack1

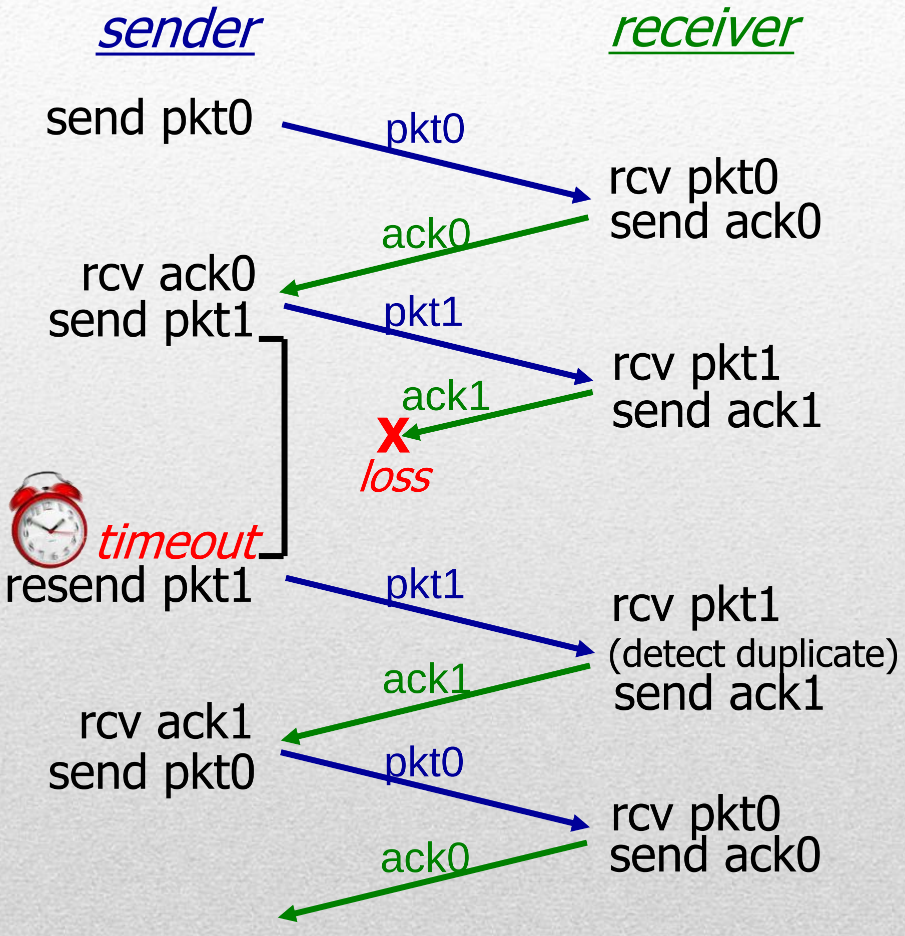
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(b) packet loss



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

X

loss

timeout

resend pkt1

pkt1

rcv pkt1
(detect duplicate)
send ack1

ack1

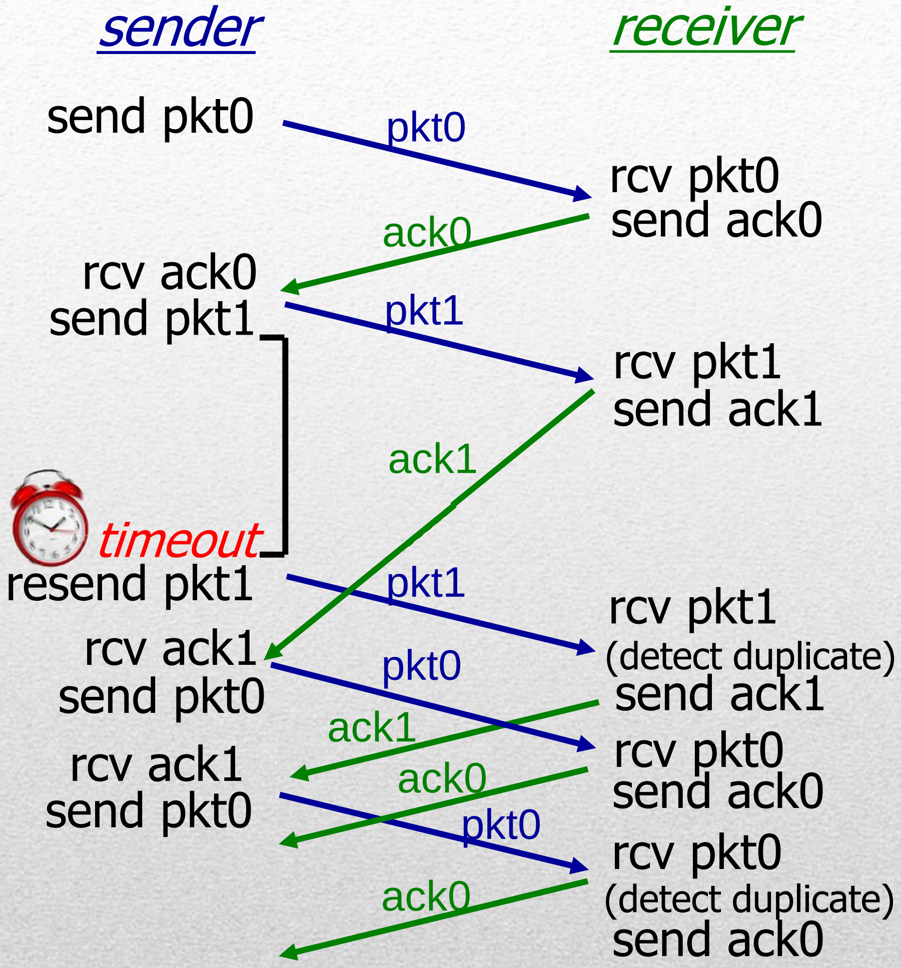
rcv ack1
send pkt0

pkt0

rcv pkt0
send ack0

ack0

(c) ACK loss



sender

receiver

send pkt0

pkt0

rcv pkt0
send ack0

ack0

rcv ack0
send pkt1

pkt1

rcv pkt1
send ack1

ack1

timeout

resend pkt1

pkt1

rcv ack1
send pkt0

pkt0

rcv pkt1
(detect duplicate)
send ack1

ack1

rcv pkt0
send ack0

rcv ack1
send pkt0

ack0

pkt0

rcv pkt0
(detect duplicate)
send ack0

ack0

(d) premature timeout/ delayed ACK

RDT 3.0 : Sender

- RDT3.0 is correct, but performance stinks

- Example
  - 1 Gbps link, 15ms propagation delay, 8000 bit packet:

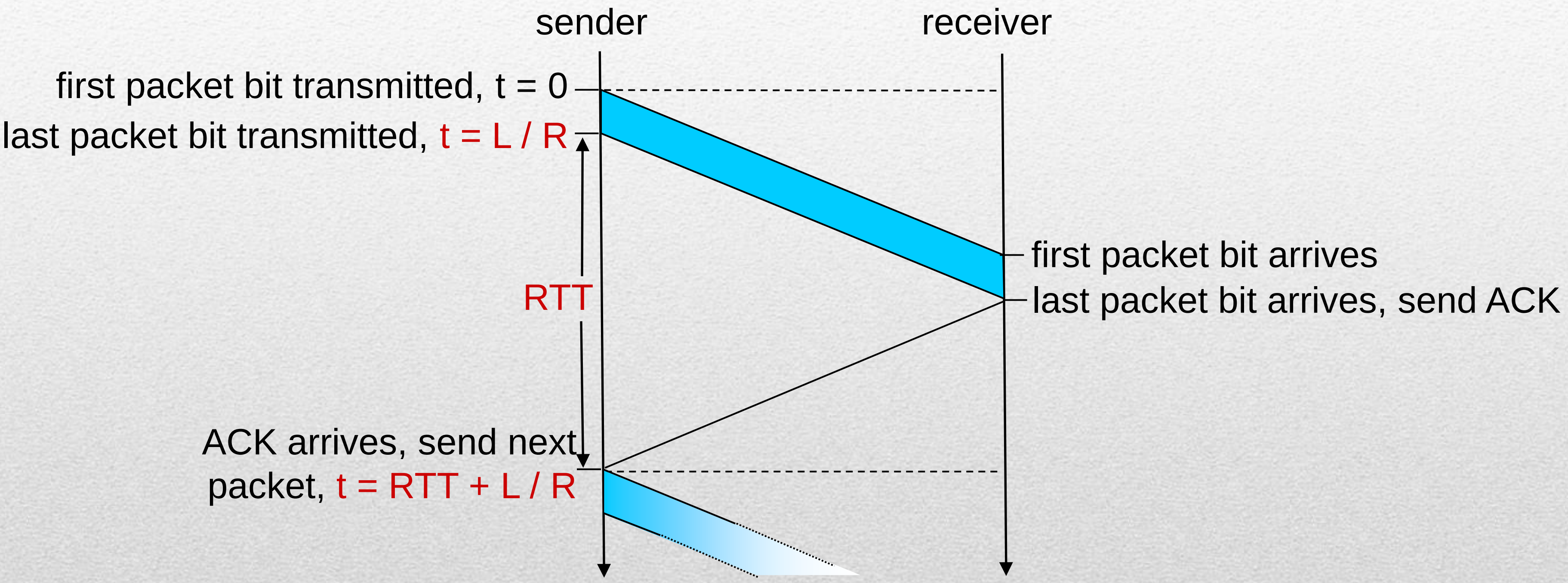
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U $_{sender}$: *utilization* – fraction of time sender busy sending

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

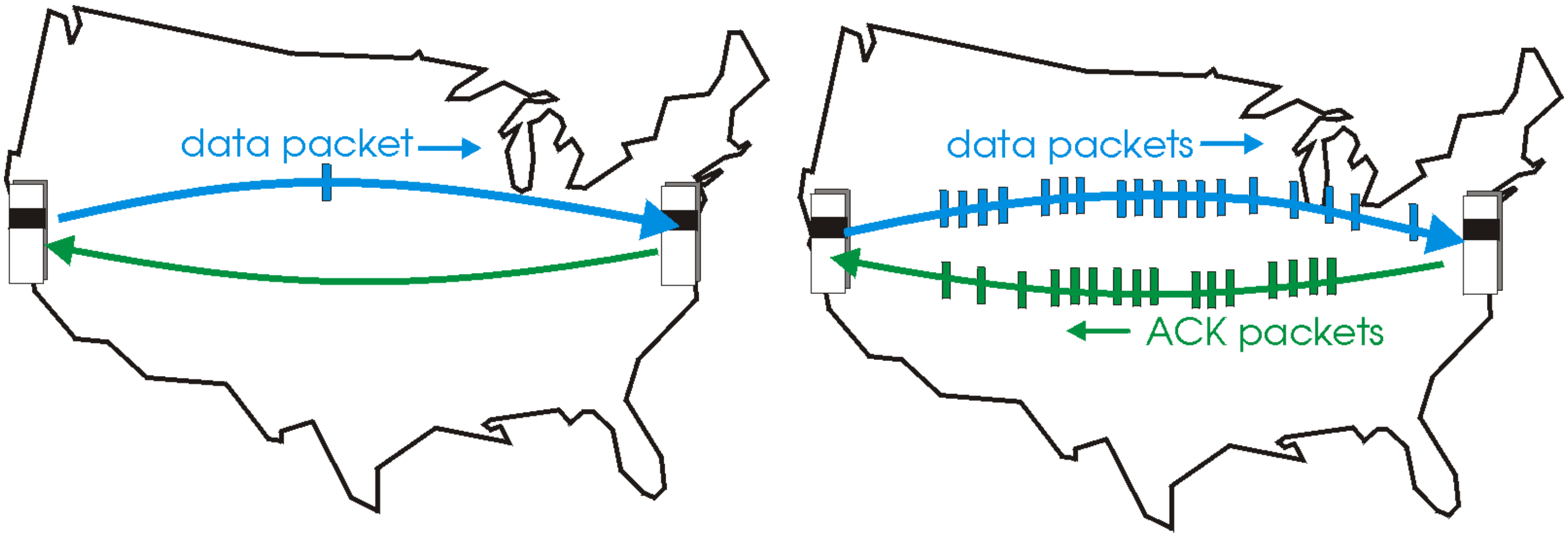
  - If RTT=30 msec, 1KB packet every 30 msec: 33kB/sec throughput over 1 Gbps link
  - Network protocol limits use of physical resources!

RDT 3.0 : Stop & Wait Operation


sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

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

Pipelined Protocols



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

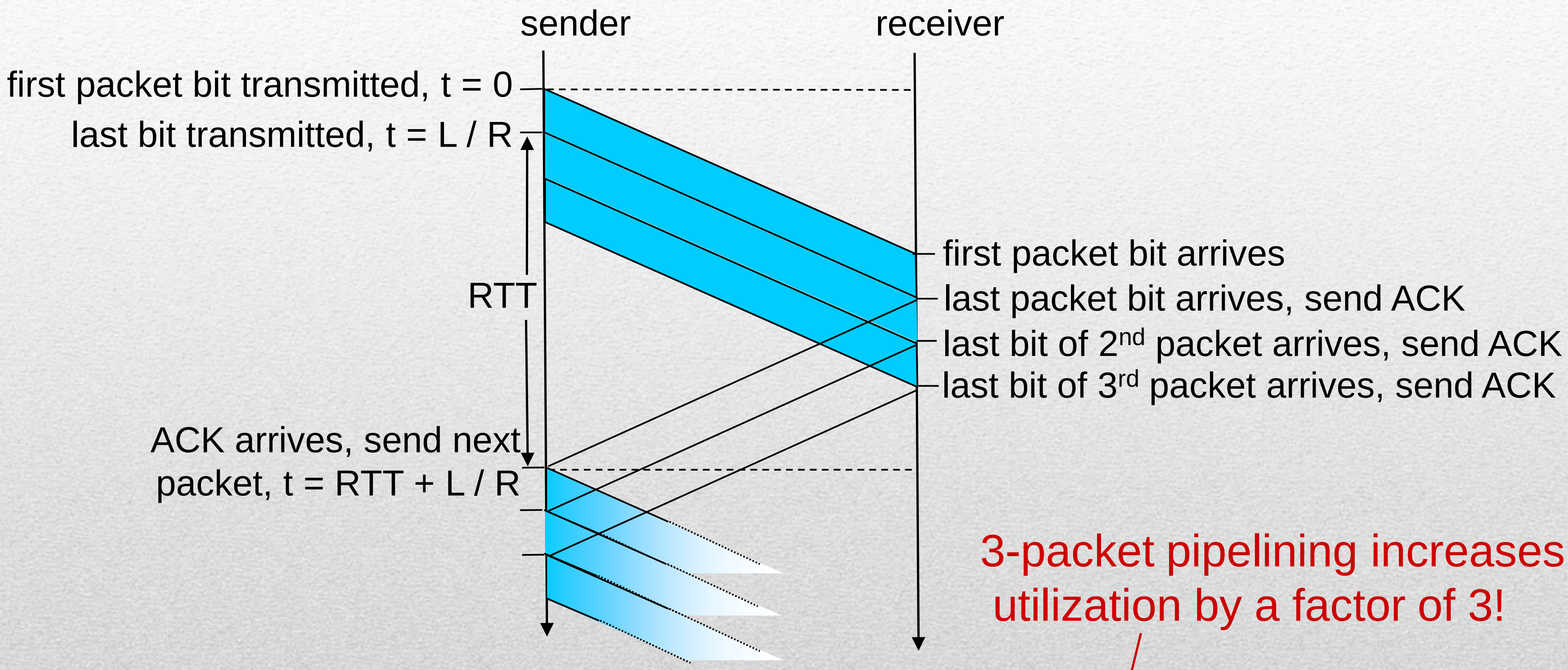
Pipelined Protocols

Sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

- Two generic forms of pipelined protocols:
  - **go-Back-N**
  - **Selective repeat**

Pipelined Protocols


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

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

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined Protocols

Go-back-N:

- Sender can have up to N un-ACKed packets in pipeline
- Receiver only sends *cumulative ACK*
  - Does not ACK packet if there is a gap
- Sender has timer for oldest un-ACKed packet
  - When timer expires, retransmit *all* un-ACKed packets

Selective Repeat:

- Sender can have up to N un-ACKed packets in pipeline
- Receiver sends *individual ACKs* for each packet
- Sender maintains timer for each un-ACKed packet
  - When timer expires, retransmit only that un-ACKed packet