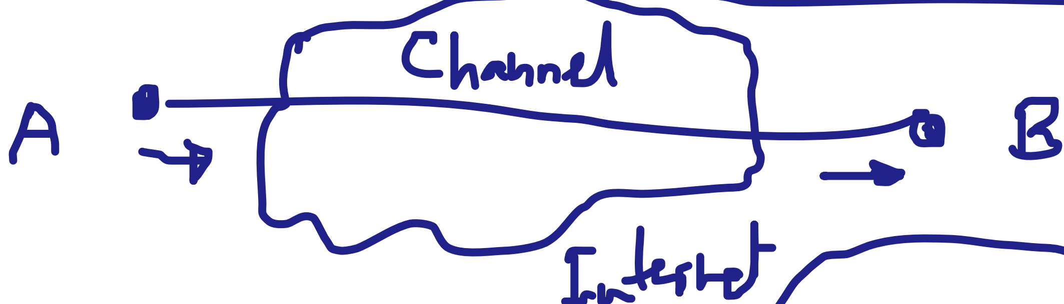
Error handling (reliability)
Flow control
Congestion Control
Connection Management.




A : msg $m \rightarrow$

B: recv $\underline{m'} \leftarrow$

$m = m'$

Transmitter
+ CPU

msg → signal

Sound wave | Electromagnetic wave e.g. light, radio wave ...

EM wave

How are you?

letter → As. code
7 bit

ASCII: (0110 ...... 0110 ...)

→ Coding / Signal → wire / air, wireless

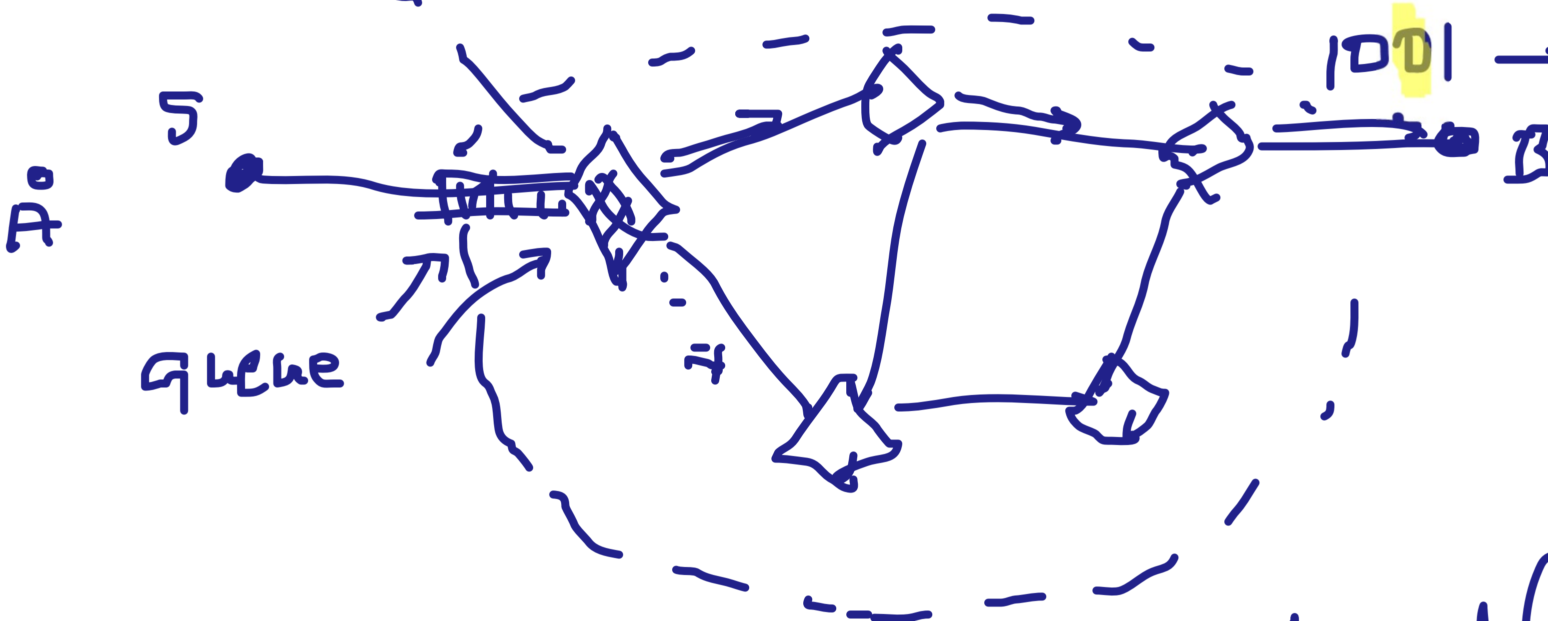

Modulate

Receiver → demodulate

1011

• **Channel** (wire/wireless)

noise → changes the signal



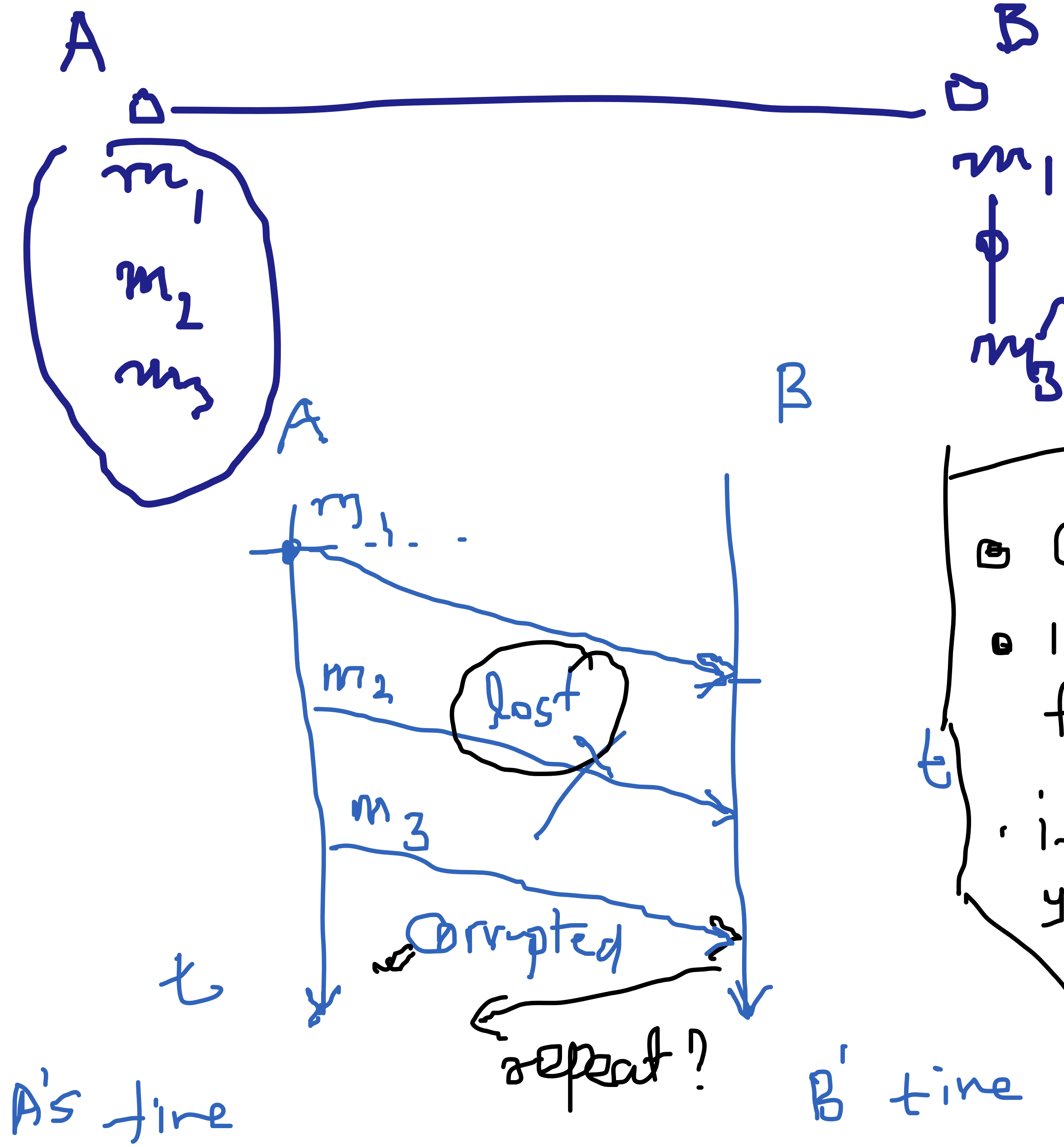

1001 → 1011

buffer overflow → packet dropped (msg lost)

mis-routed → lost

$m = m'$ · whether $m$ is lost.

or $\phi$




TCP

t

- Check for error
- If error, req for repeat
- if no error, you acknowledge

4 bit msg

A 0 1 1 0

1 1 1 0 B

how does B know that the msg is correct ?

Vivek: Checksum, parity bit.

ORANGE → ORANGE ✓ Renish

MANGO. wrong ORAMGE × Dictionary

Guessing

word w1, w2 close

distance (w1, w2)

error detection.

correction

Formula based correct word

parity bit

0 1 1 0

a b c d | parity bit

11101 → 11100 ?

noise

wrong

word → even parity : $\underbrace{0+1+1+0}+\underbrace{0}_{p\ 0/1}$ → even

$\underbrace{1110}\underbrace{1}_{p}$ →

Recv

11101 → even parity ? ✓

$\underbrace{1110}$ → correct.

→ ~~bit~~ 11001 → parity ? → odd → wrong

→ 11011 noise → parity ? → odd → correct ✗

parity bit → multiple bit Checksum ✓

↳ 16 bit / 32 bit

4 bit checksum.

16 bit

msg 1011 0011 1000 1110 | 4 bit | checksum

1100 ✓

1011
0011
1000
1110

+
over 1 0 1 0 0

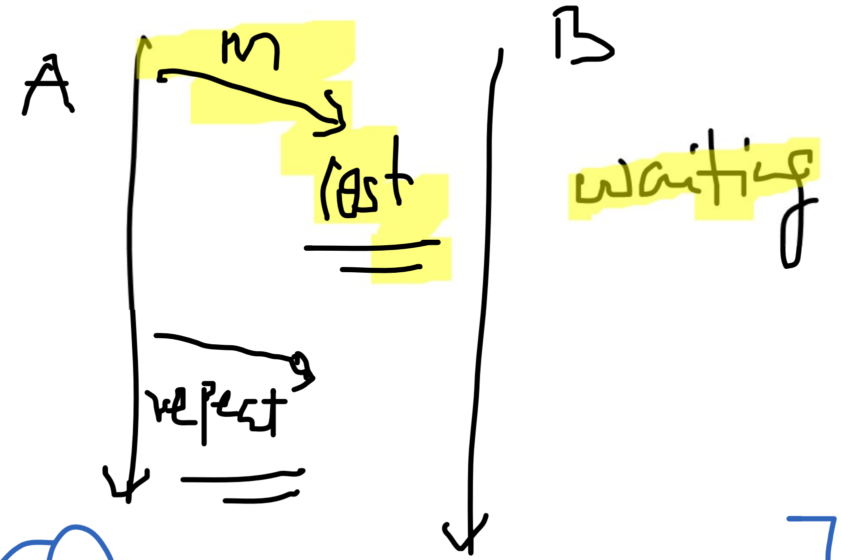
1110
0001

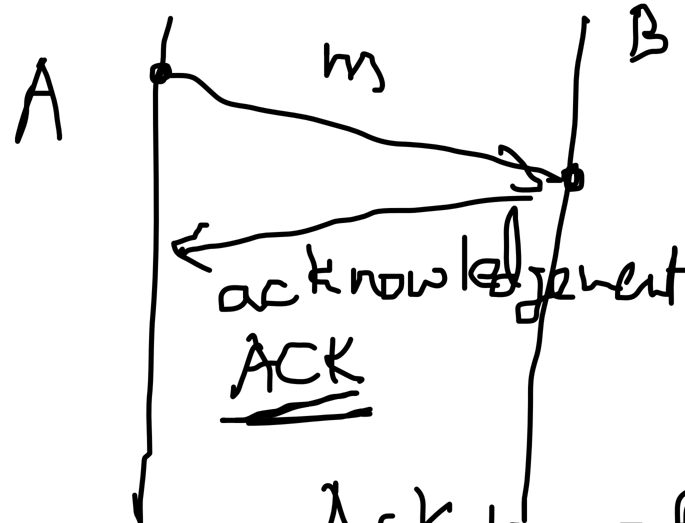
+ 1100 Checksum
0000

CRC

Cyclic Redundancy Check.

→ number theory




ACK time = $t_a$

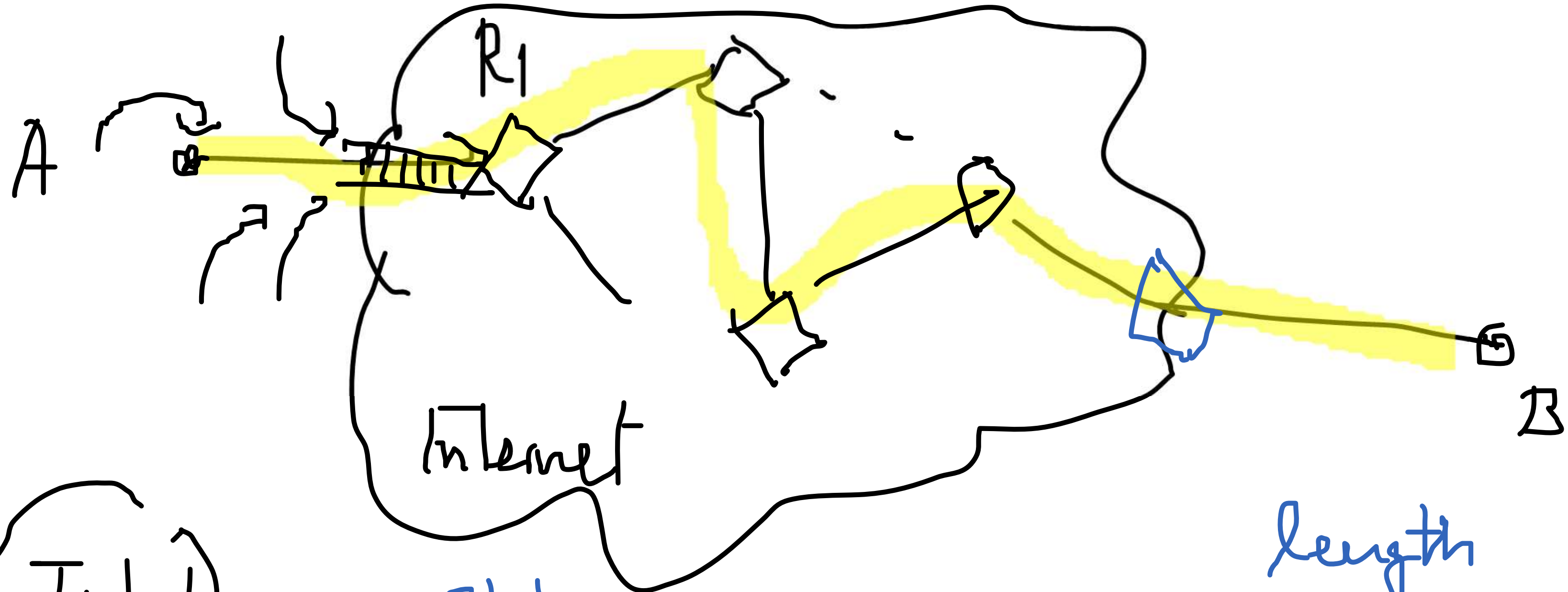
A send msg.
wait for sometime
if ACK doesn't come,
send msg again

- msg lost
- ack lost
- msg corrupted

Max. time to wait.

Maximum packet size.

history: $[t_a^1, t_a^2, \ldots, t_a^k] \rightarrow \text{MAX} = t_w$




Total delay = $\underline{\text{Transmission delay}}$ + $\underline{\text{propagation delay}}$ + $\underline{\text{queuing delay}}$ + $\underline{\text{processing delay}}$

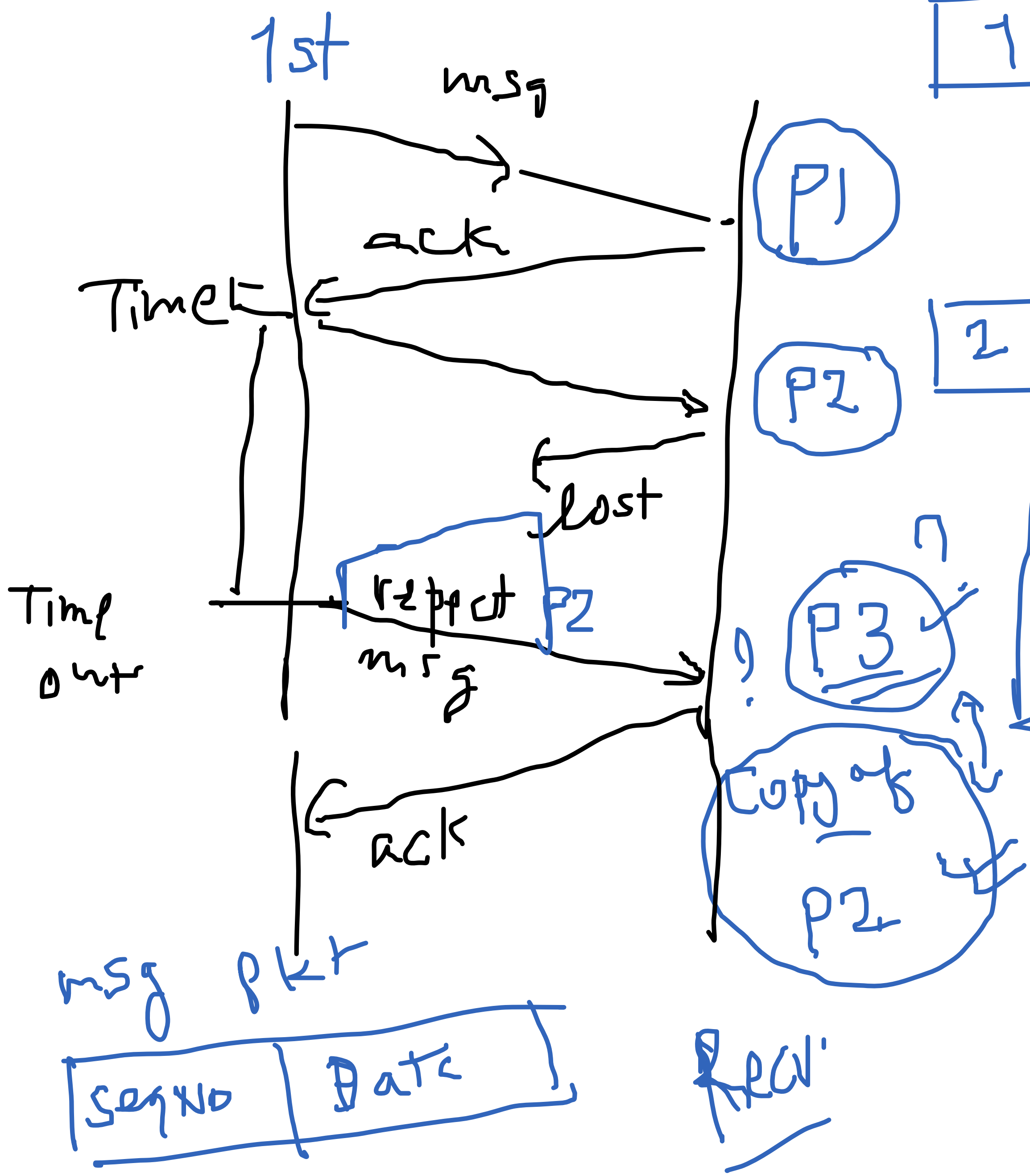
pkt size

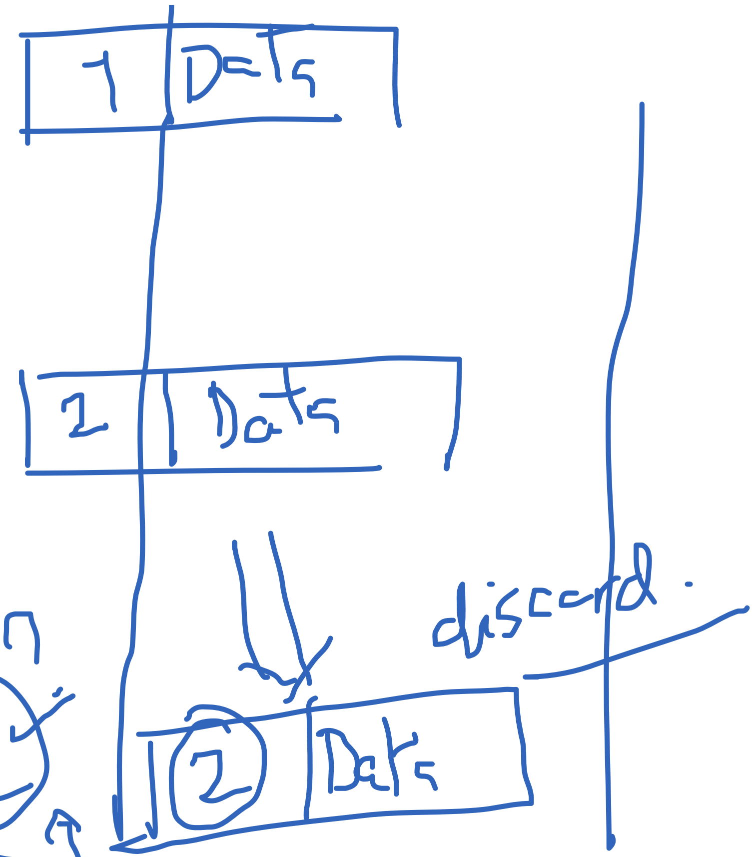
length

CPU

variable.

traffic

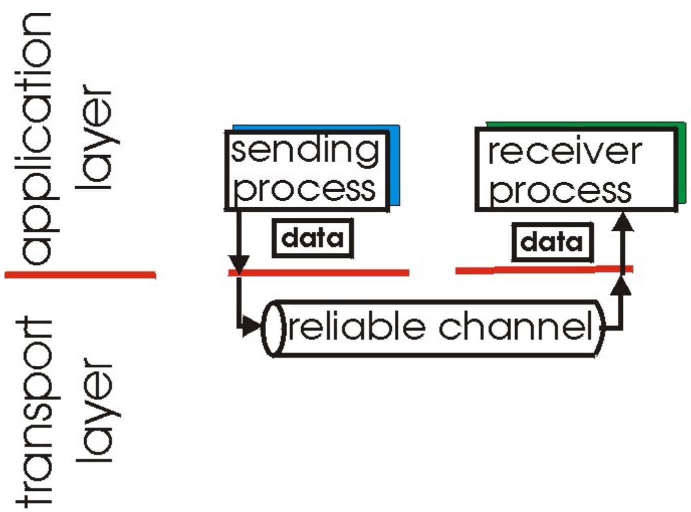



Introduce seq. no. in a pkt

# Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

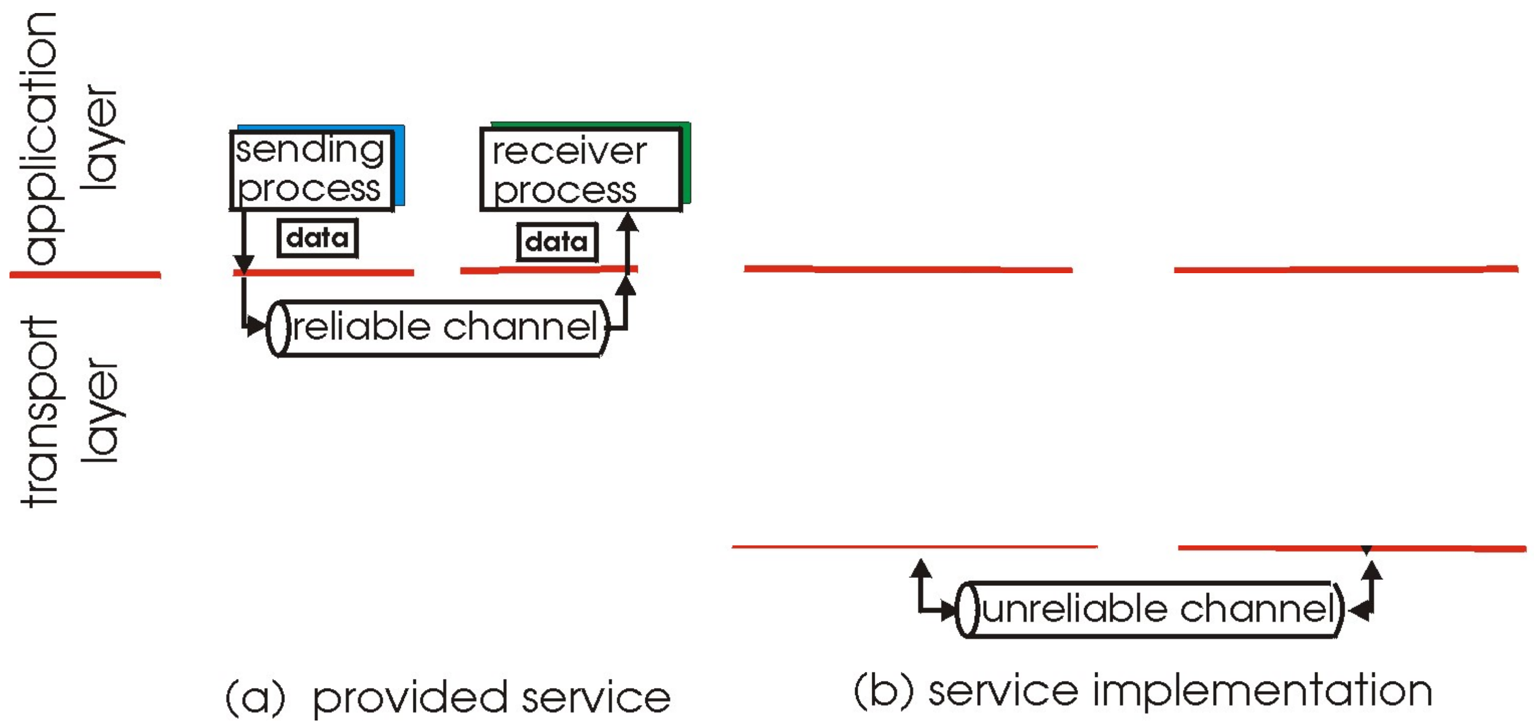



(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Principles of reliable data transfer

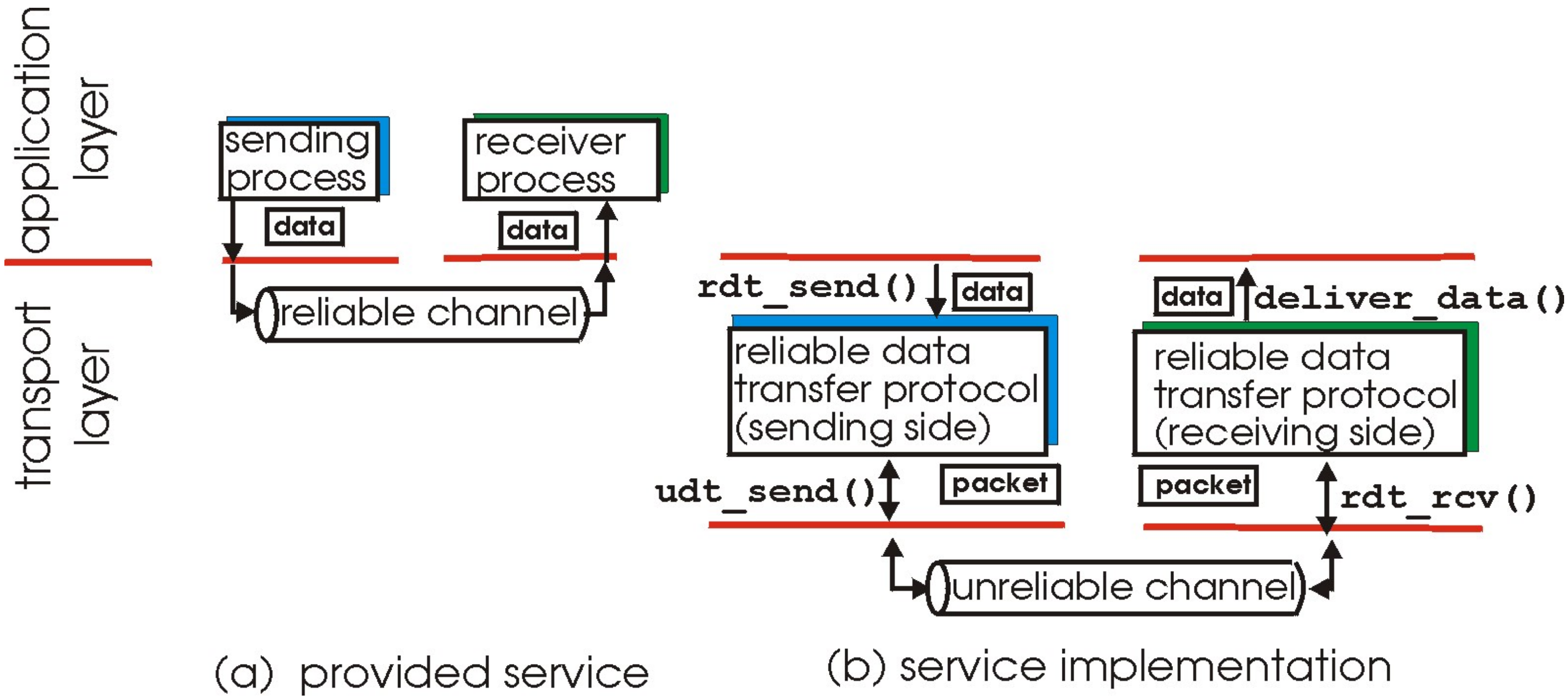
- important in application, transport, link layers
  - top-10 list of important networking topics!




- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

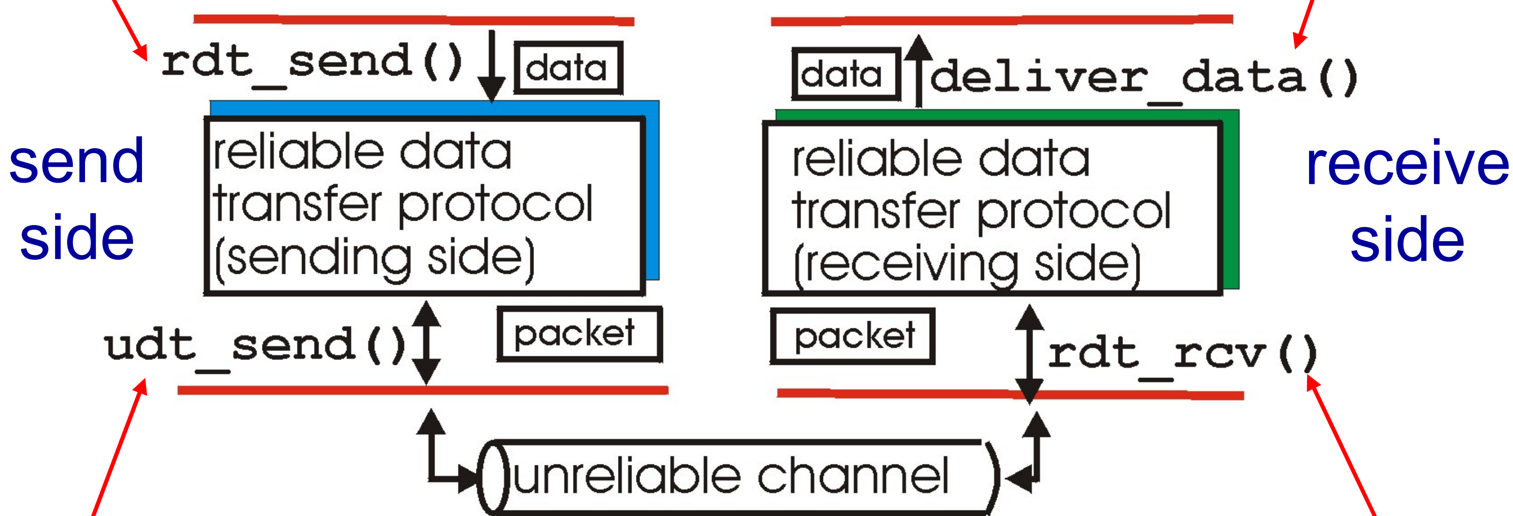



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Reliable data transfer: getting started

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

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



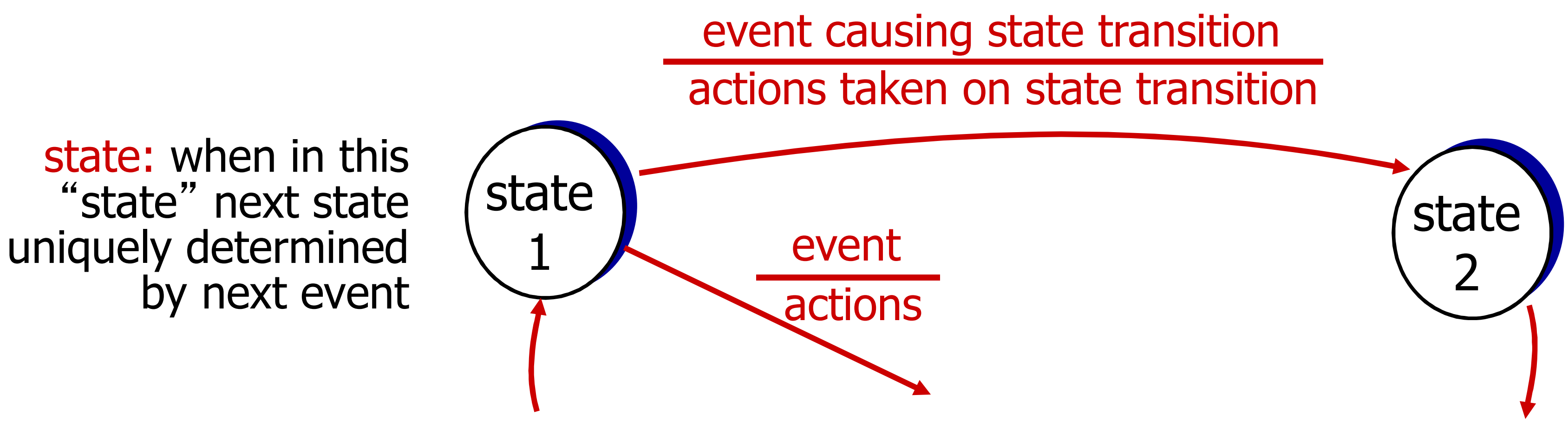

**udt_send():** called by rdt, to transfer packet over unreliable channel to receiver

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

# Reliable data transfer: getting started

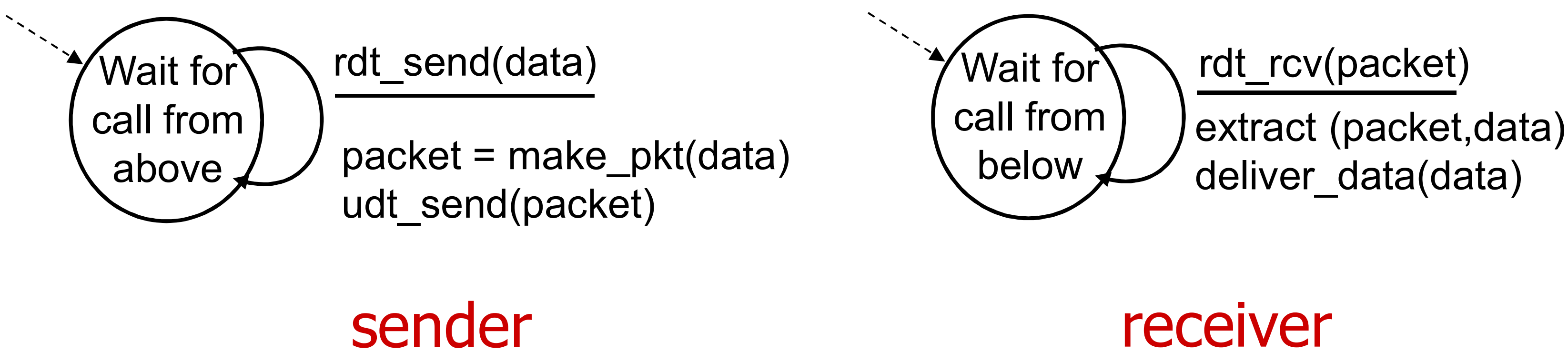
we' ll:

- 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



# 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



# rdt2.0: channel with bit errors

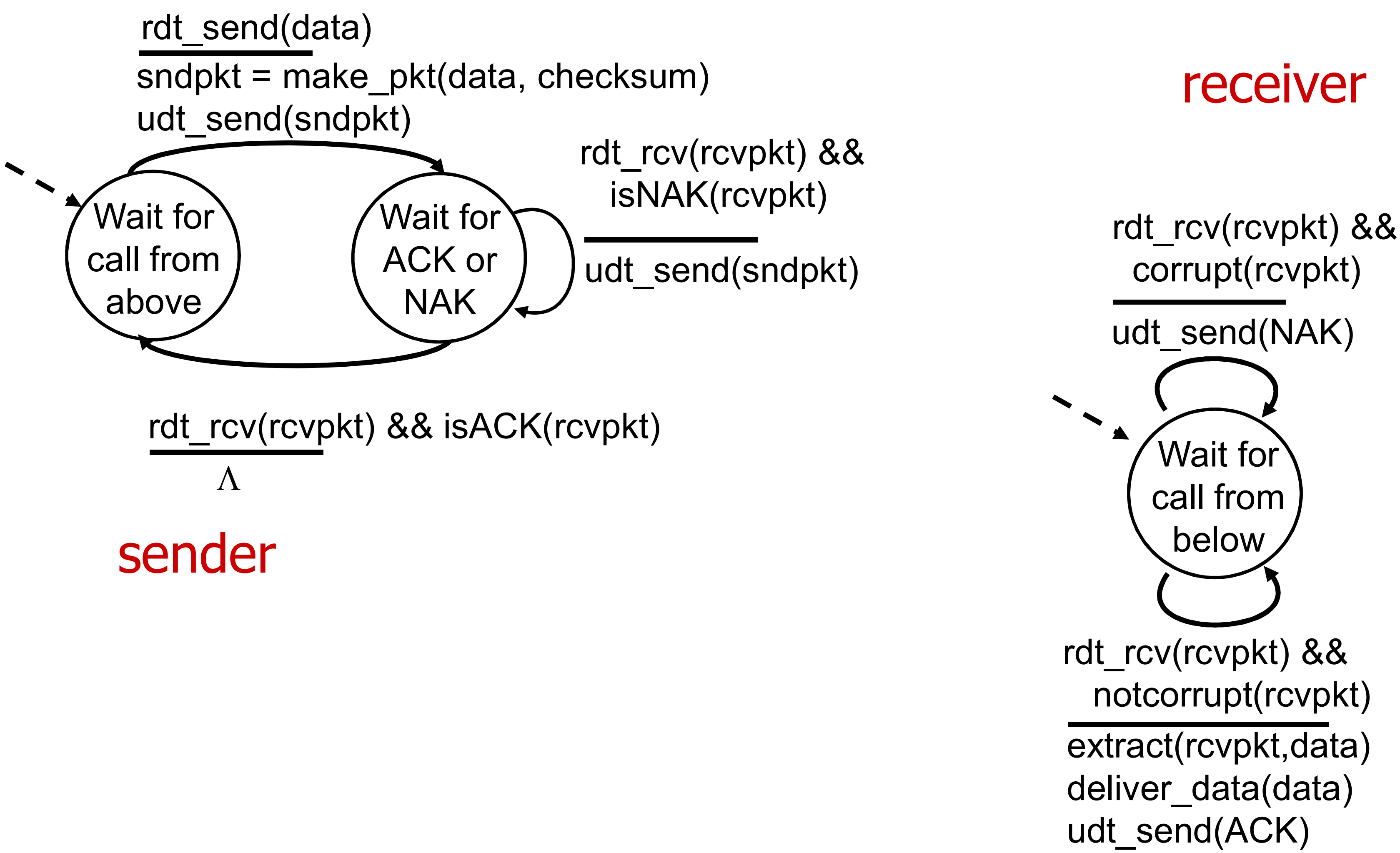
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

*How do humans recover from "errors" during conversation?*

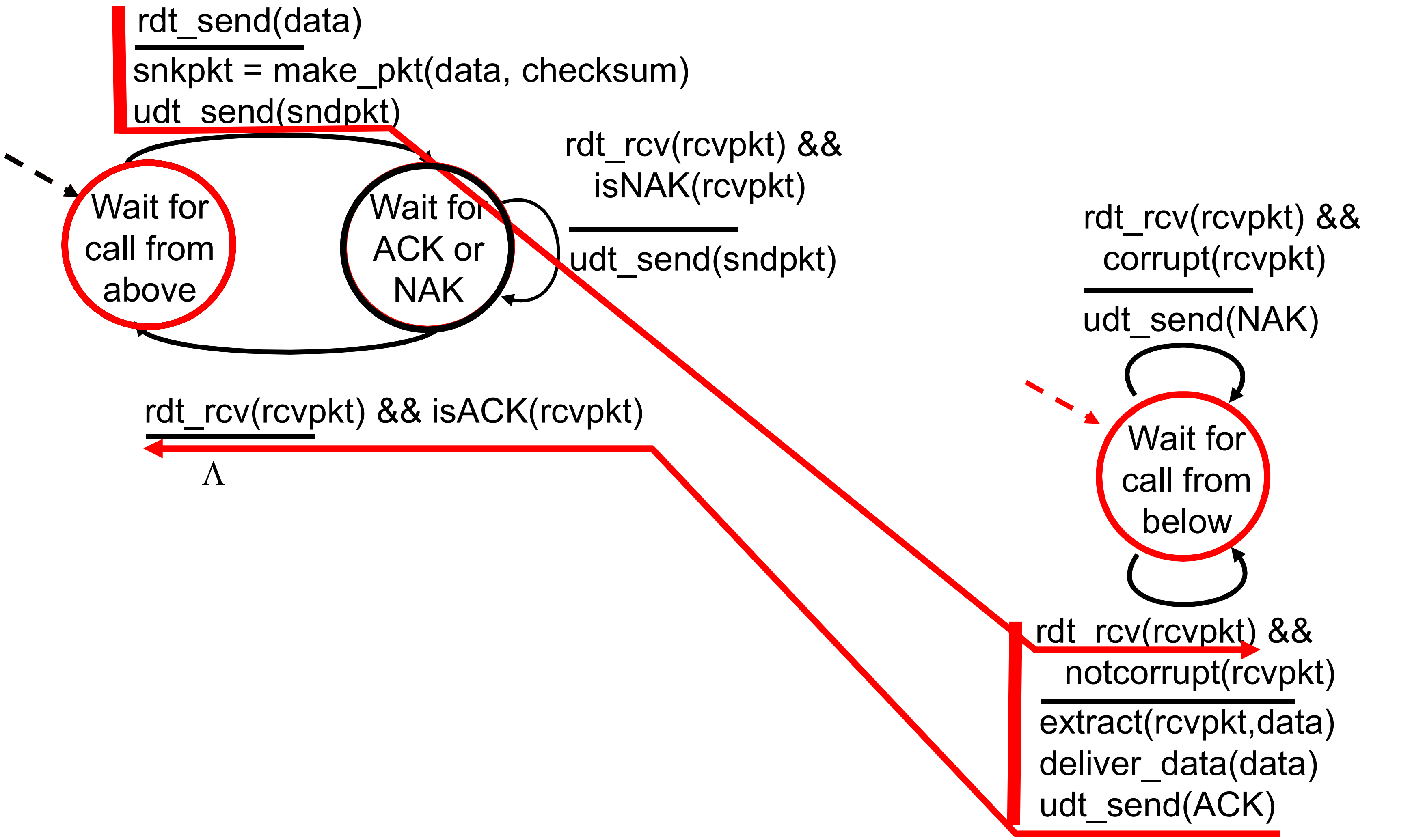
# rdt2.0: channel with bit errors

- 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

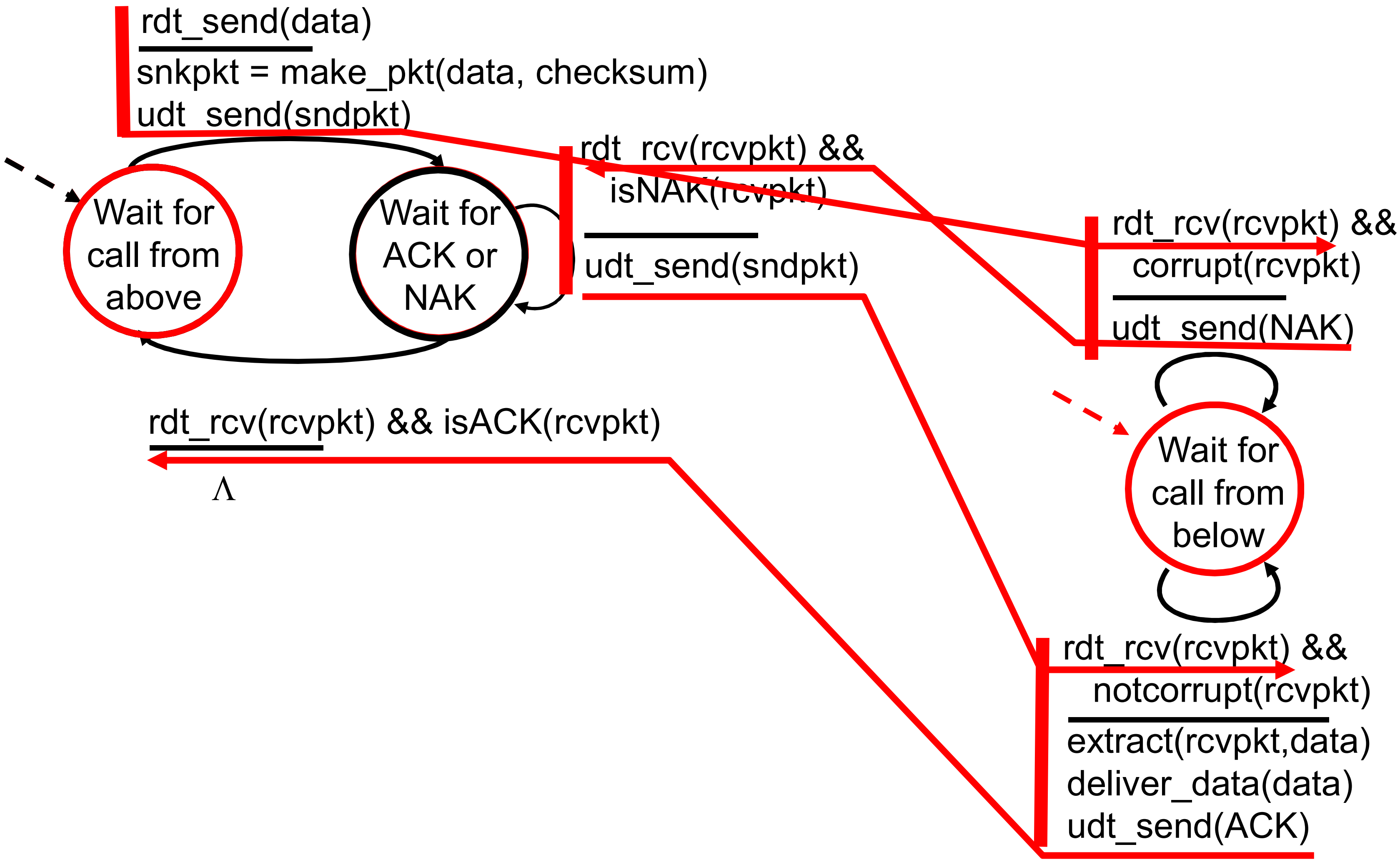
# rdt2.0: FSM specification



# rdt2.0: operation with no errors



# rdt2.0: error scenario



# 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

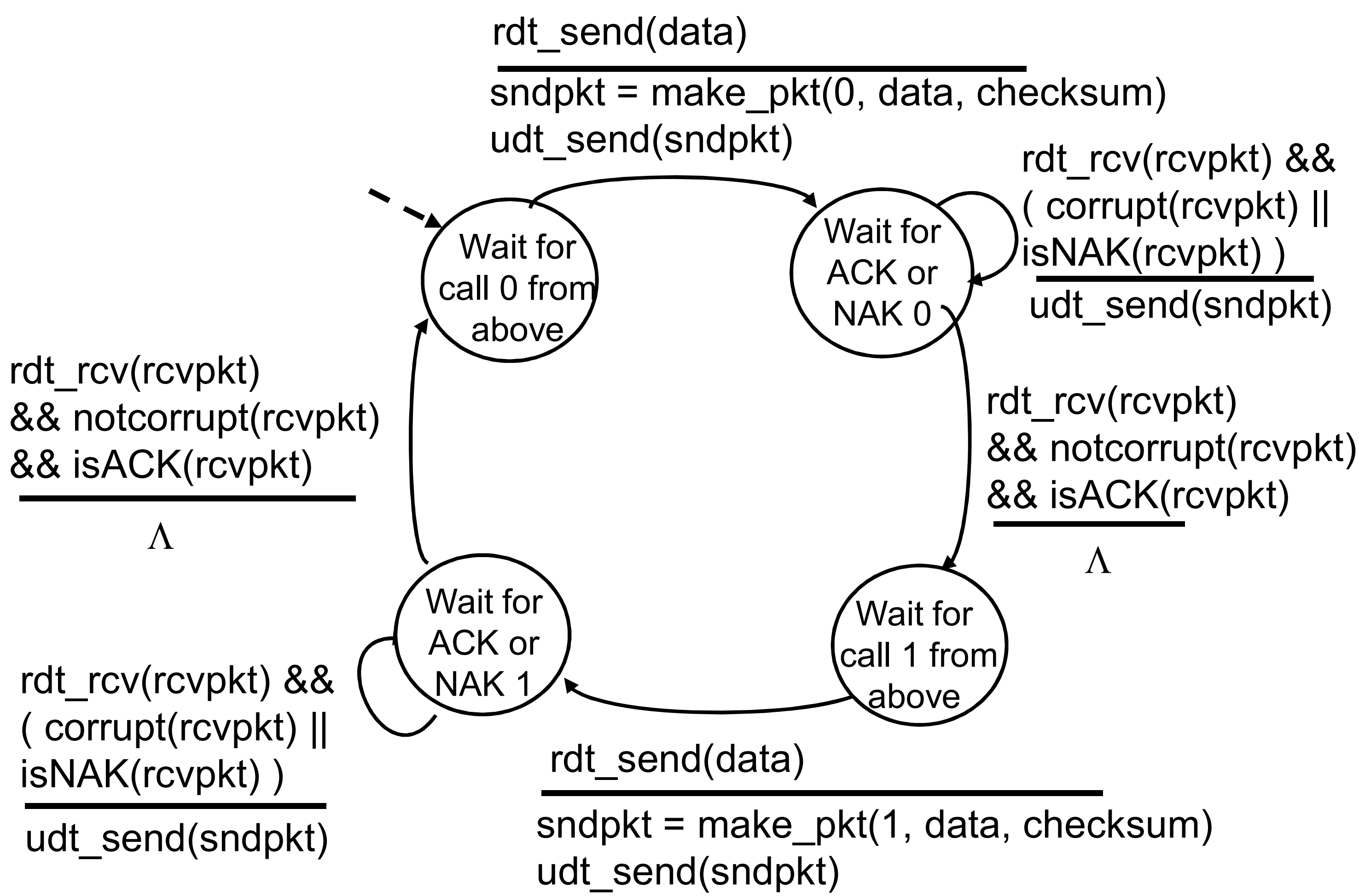
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

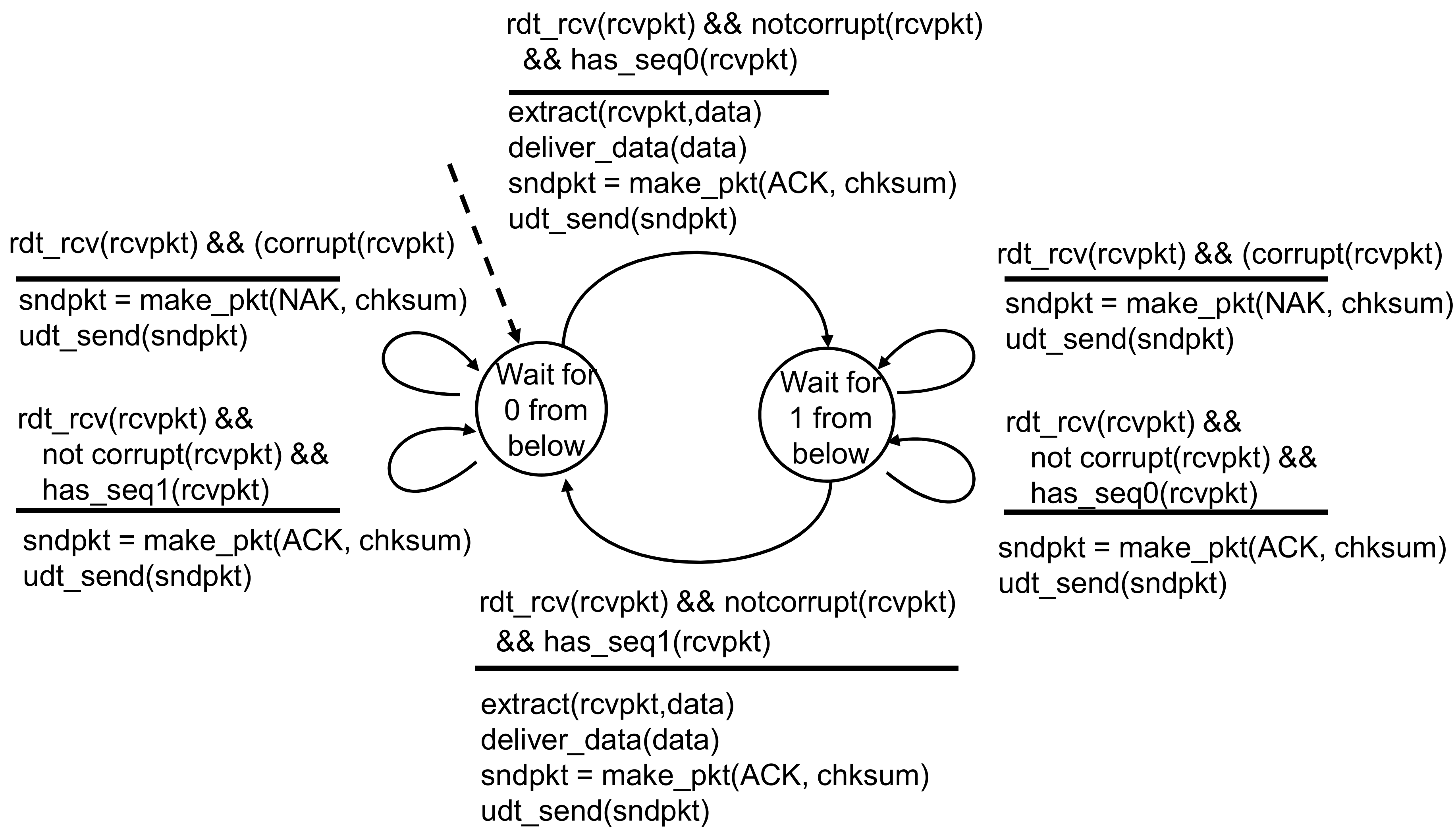
**stop and wait**

sender sends one packet, then waits for receiver response

# rdt2.1: sender, handles garbled ACK/NAKs



# rdt2.1: receiver, handles garbled ACK/NAKs



# 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

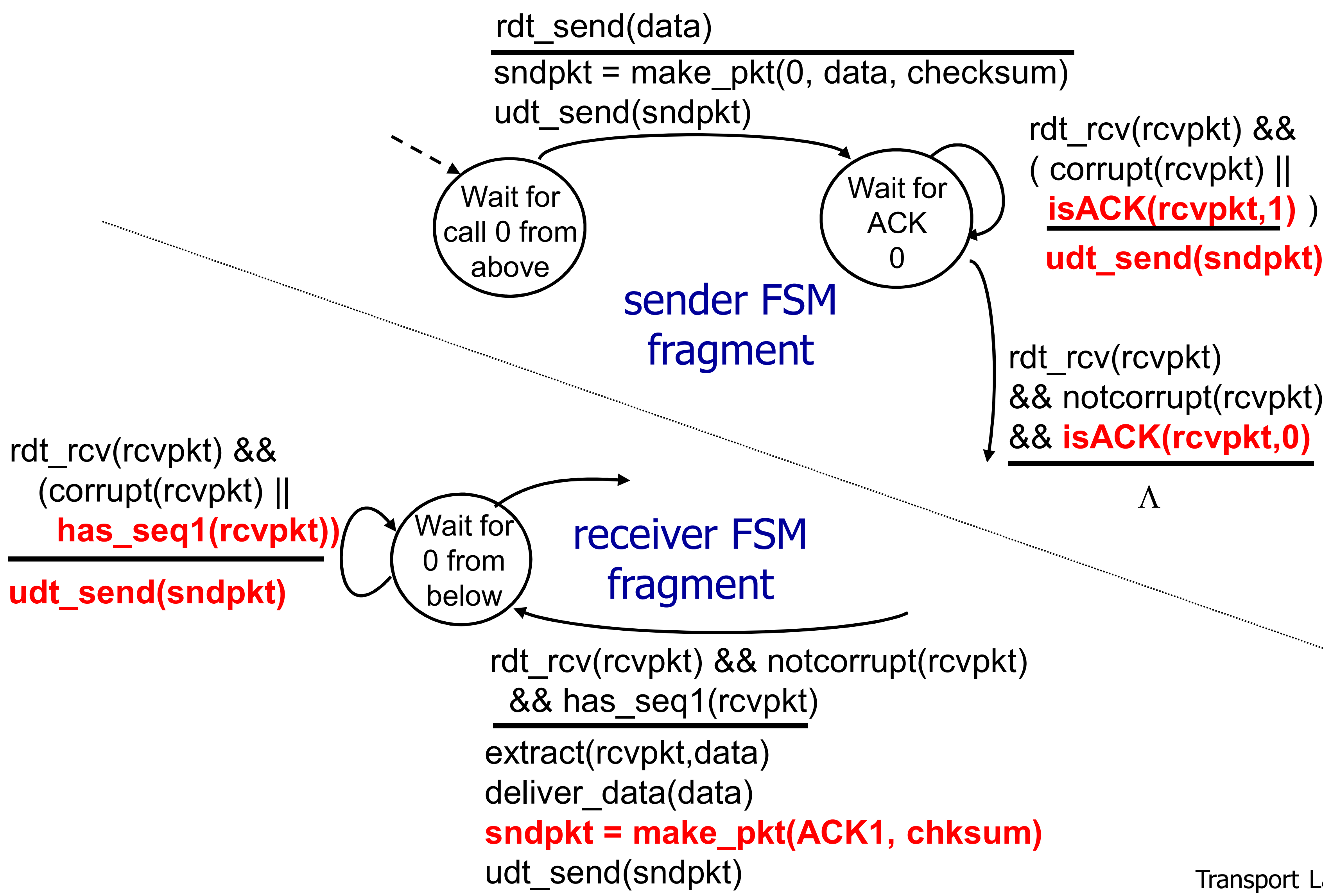
receiver:

- 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

# rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - 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



# rdt3.0: channels with errors *and* loss

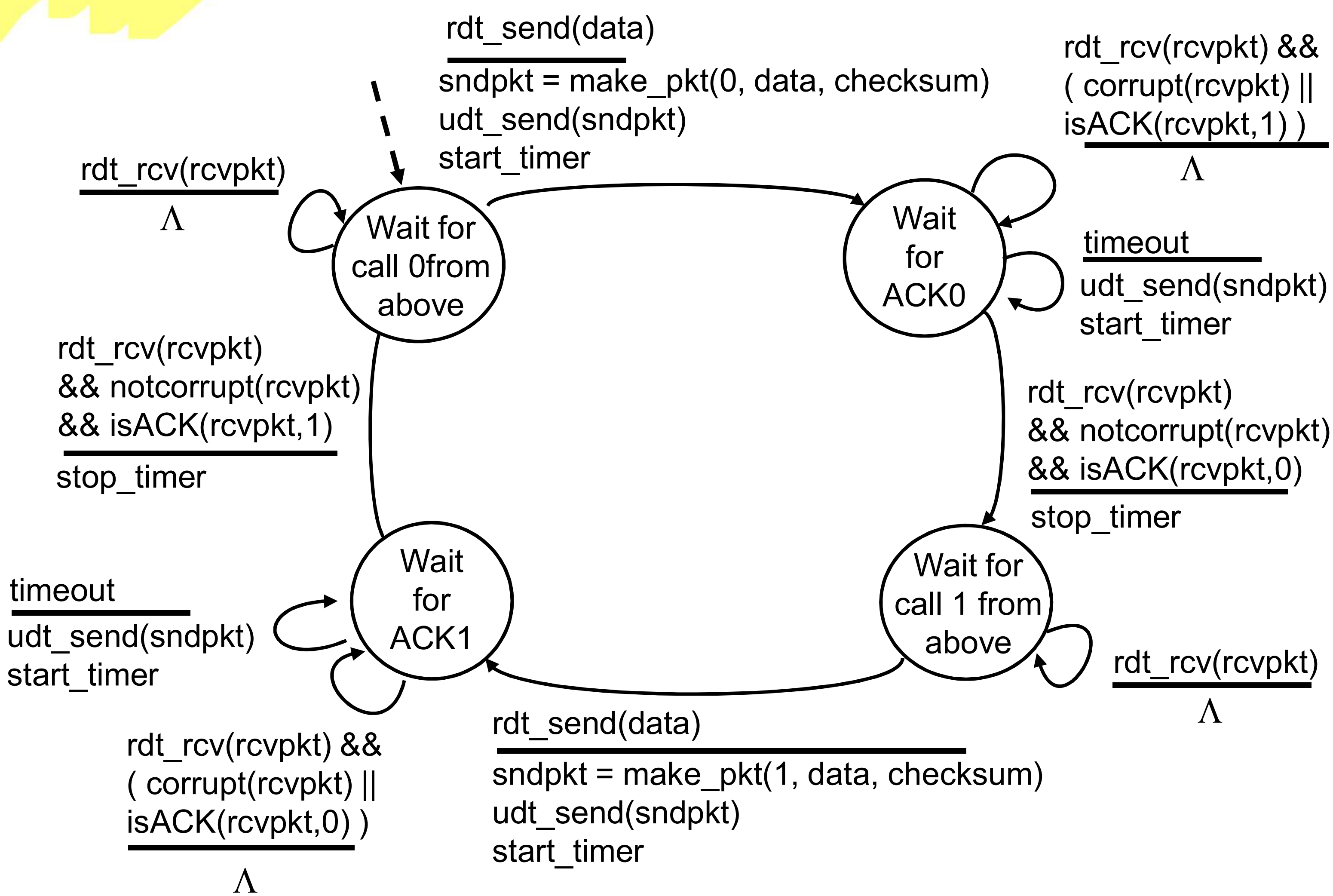
new assumption:
underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

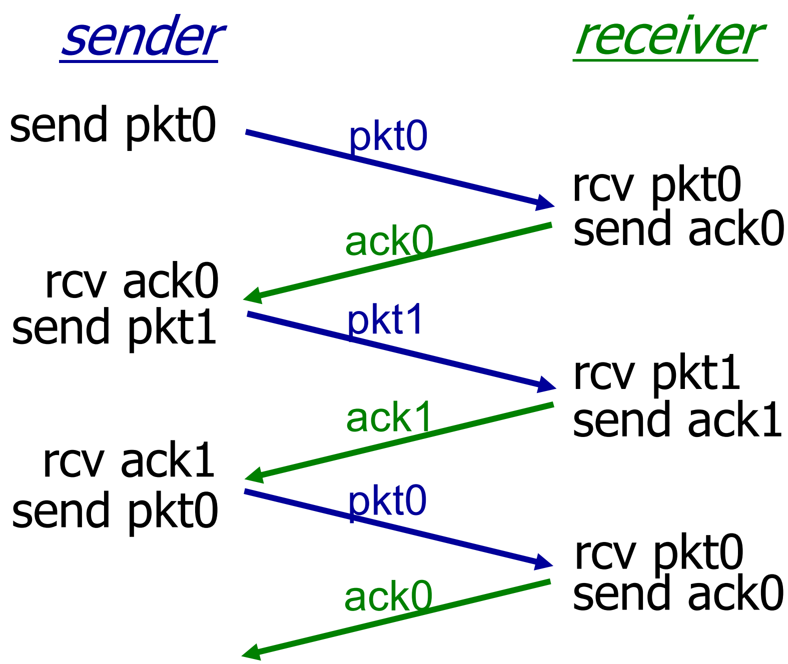
approach: sender waits "reasonable" amount of time for ACK

- 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

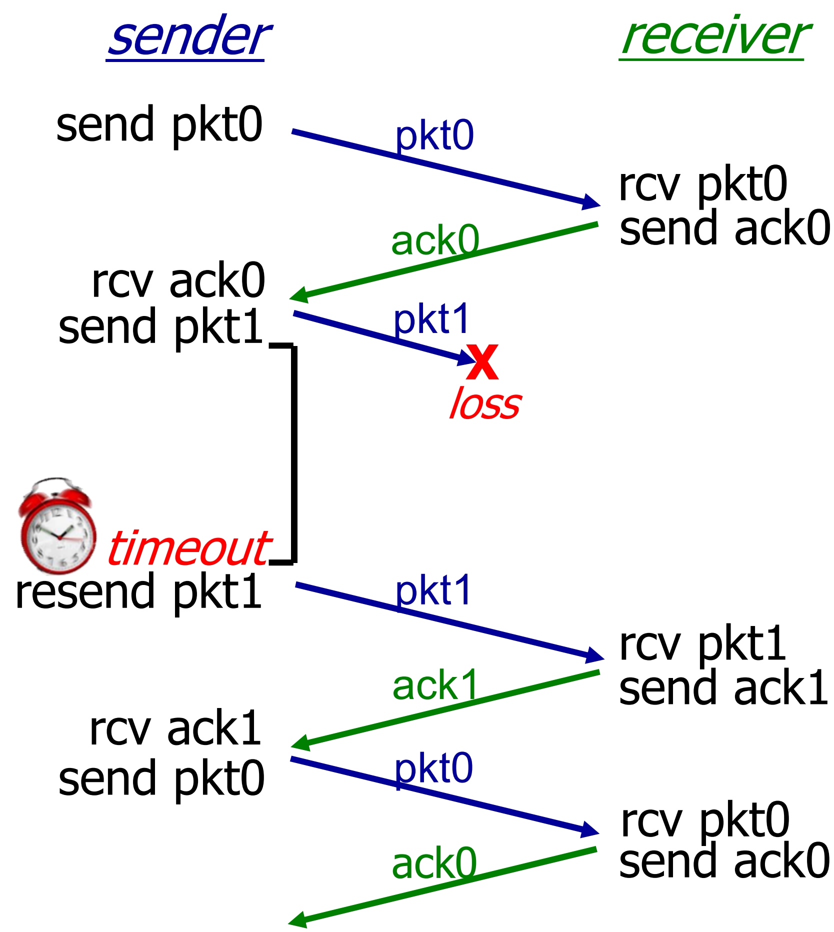
# rdt3.0 sender



# rdt3.0 in action

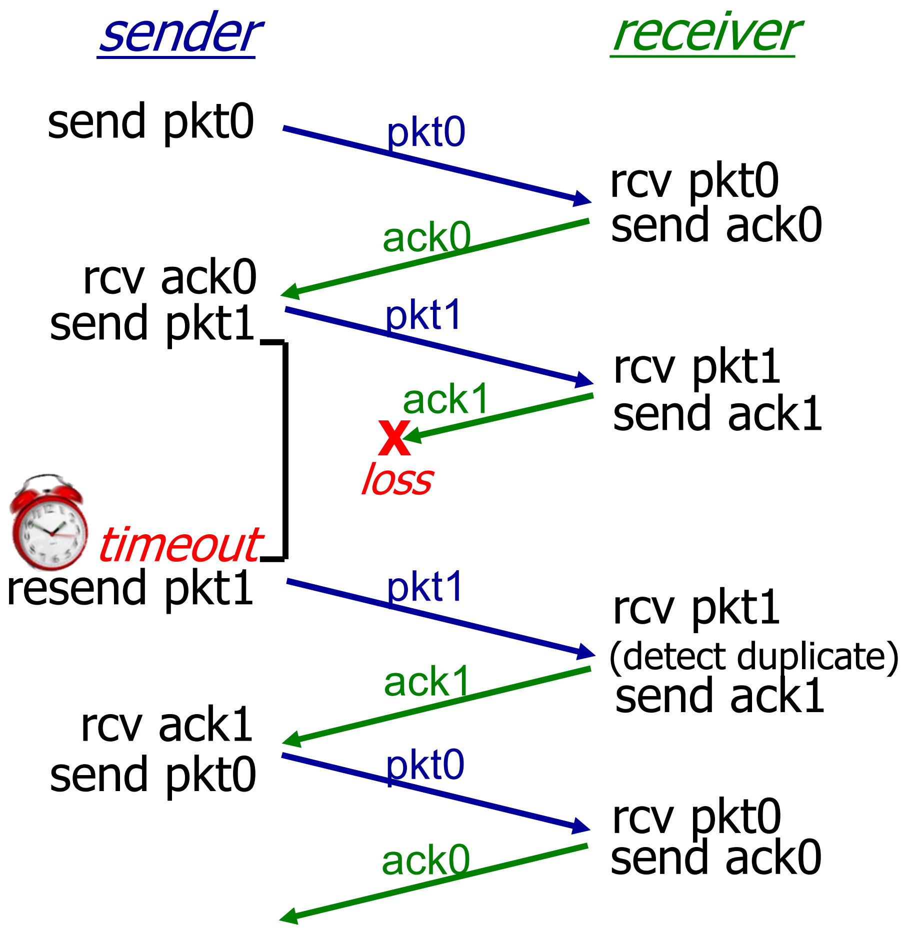



(a) no loss

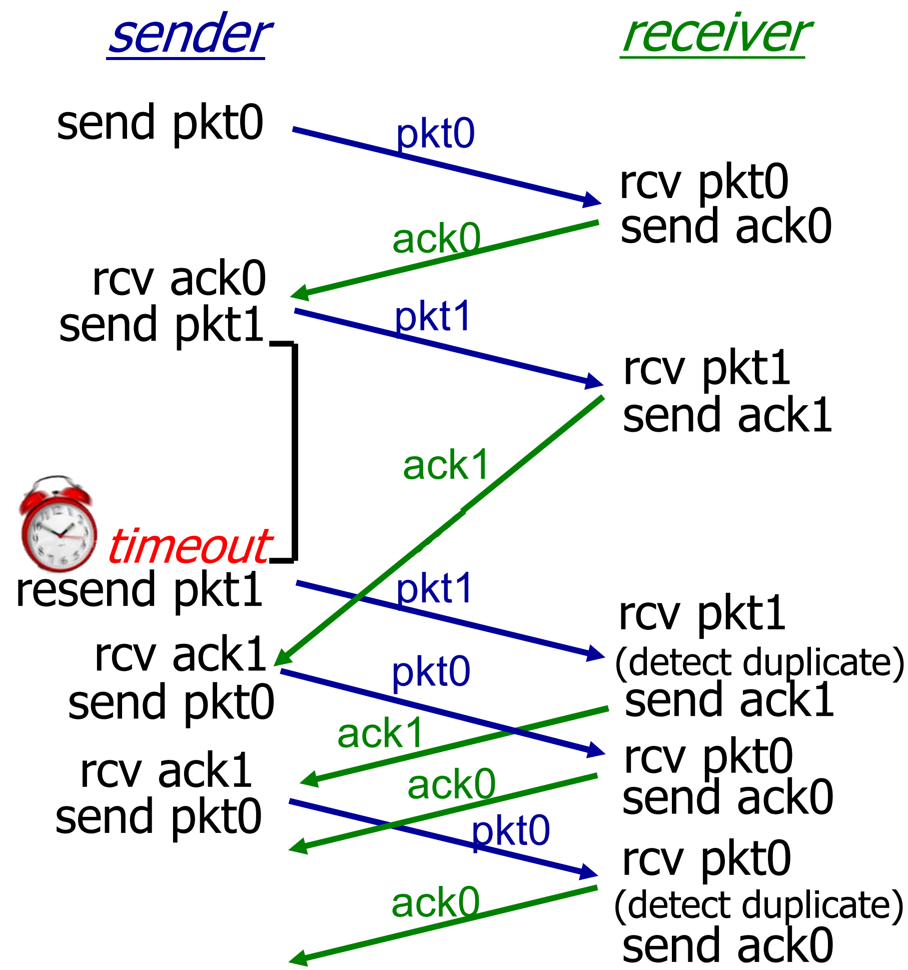



(b) packet loss

# rdt3.0 in action




(c) ACK loss




(d) premature timeout/ delayed ACK