CS 755 – System and Network Architectures and Implementation

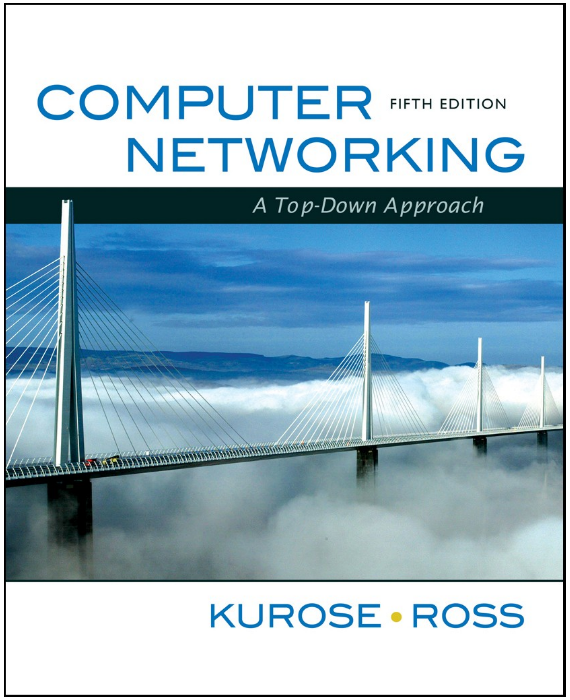
Module 3 - Transport

Martin Karsten

mkarsten@uwaterloo.ca

Notice

Some slides and elements of slides are taken from third-party slide sets. In this module, parts are taken from the Kurose/Ross slide set. See detailed statement on next slide...



A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Overview

- multiplexing, virtual channel
- reliable transmission
- flow and congestion control
- connection management and semantics

Networks – Review

- network graph
- goal: facilitate any-to-any communication
- main concerns
  - routing
  - addressing
  - scalability
- virtualization

Transport

- virtual channel
- end-to-end transmission performance
  - reliable transmission
  - rate control
- connection management

Hop by Hop vs. End to End?

- some services only hop by hop
  - delay control
  - throughput guarantees
- others also end to end
  - multiplexing
  - loss control – reliable transmission
  - rate control
- principle: if possible, use end to end

Multiplexing

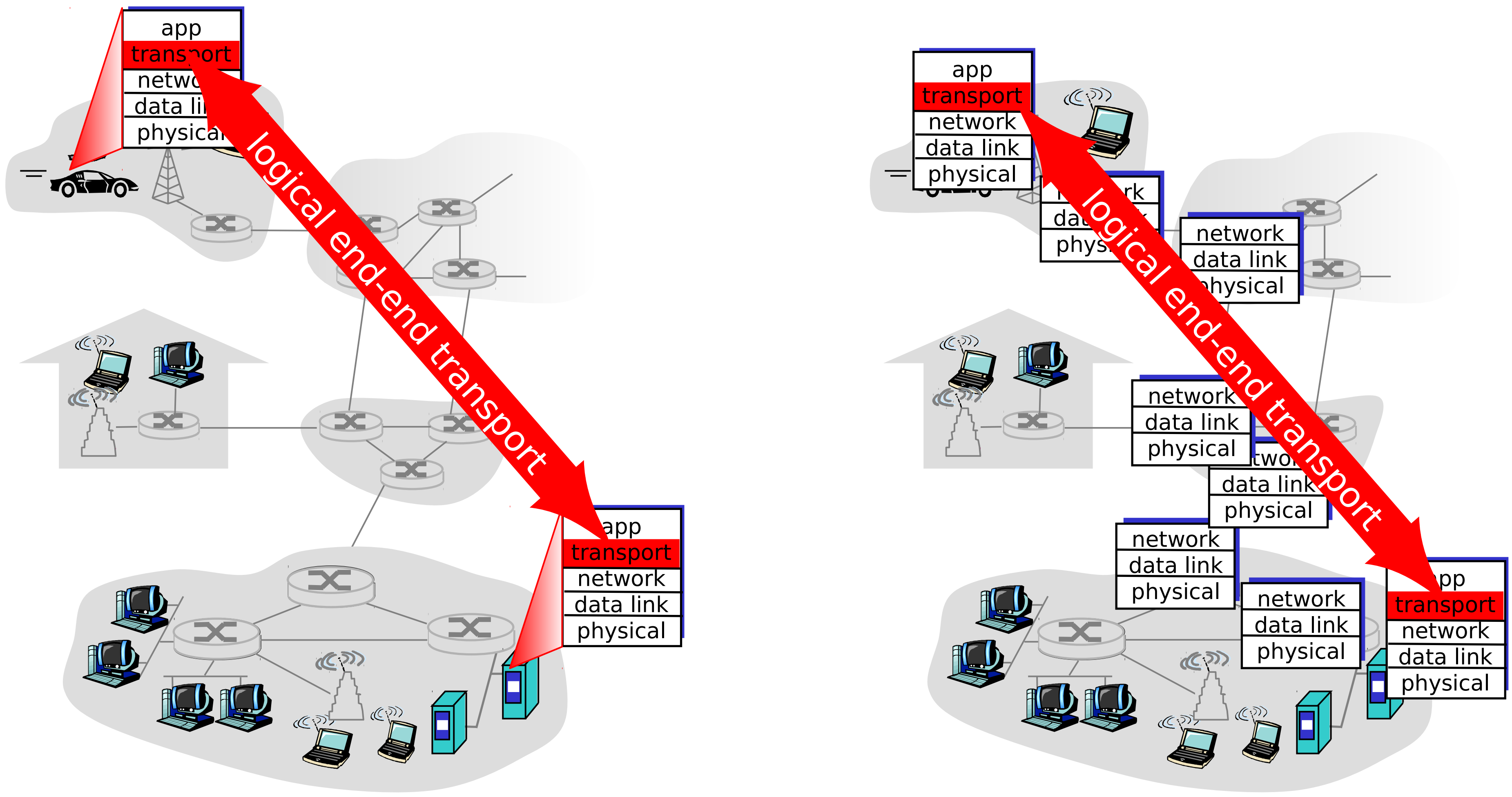
- multiple logical sessions over same channel
- here: IP connectivity provides “virtual channel”
- transport session also provides “virtual channel”
- multiplexing
  - encapsulation / stacking of multiplex label
- demultiplexing
  - forwarding according to multiplex label
  - decapsulation / remove multiplex label

Multiplexing

- Internet addressing convention:
  - IP address – network node address
  - transport *port* – transport session identifier

- Other Approaches
  - virtual circuit – integrated with network layer
  - hop-by-hop transport service
  - session identified locally by *virtual circuit identifier*

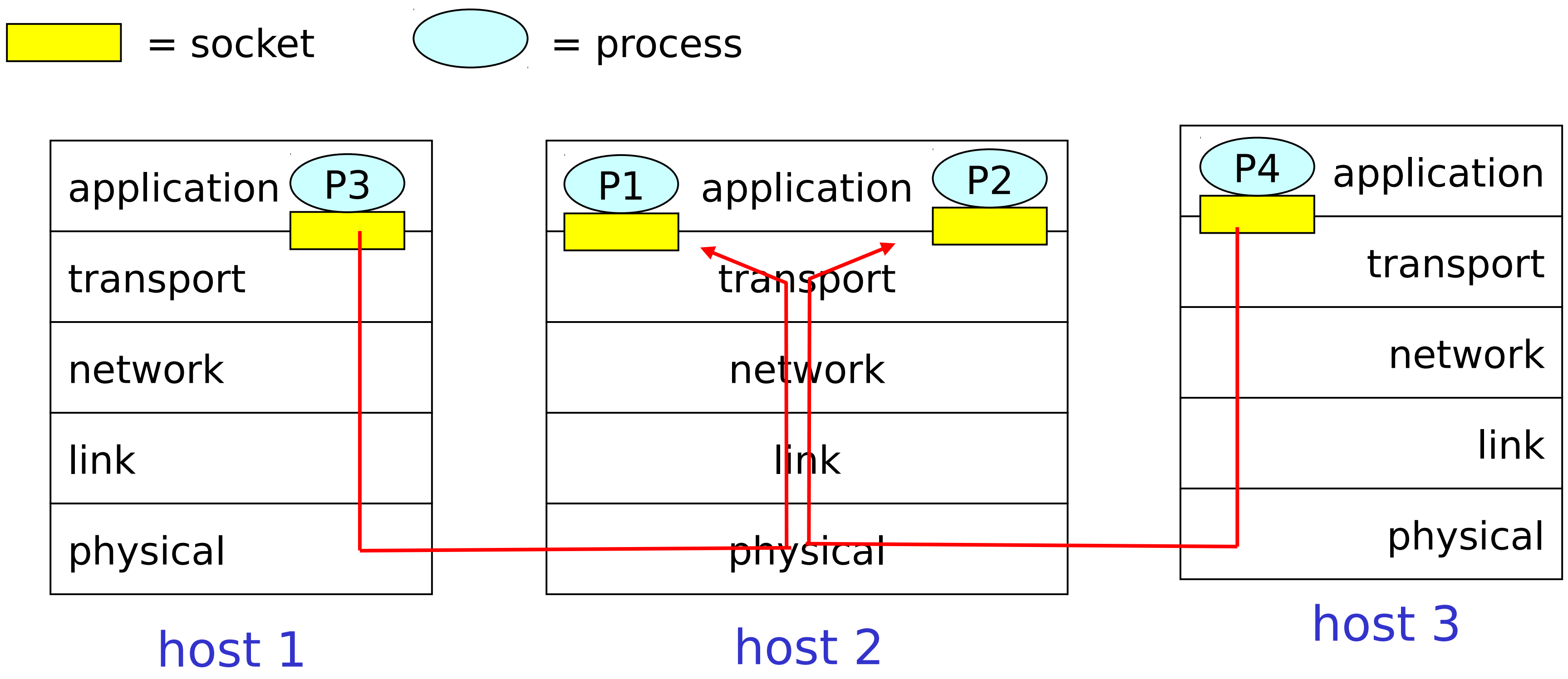
Layered Service



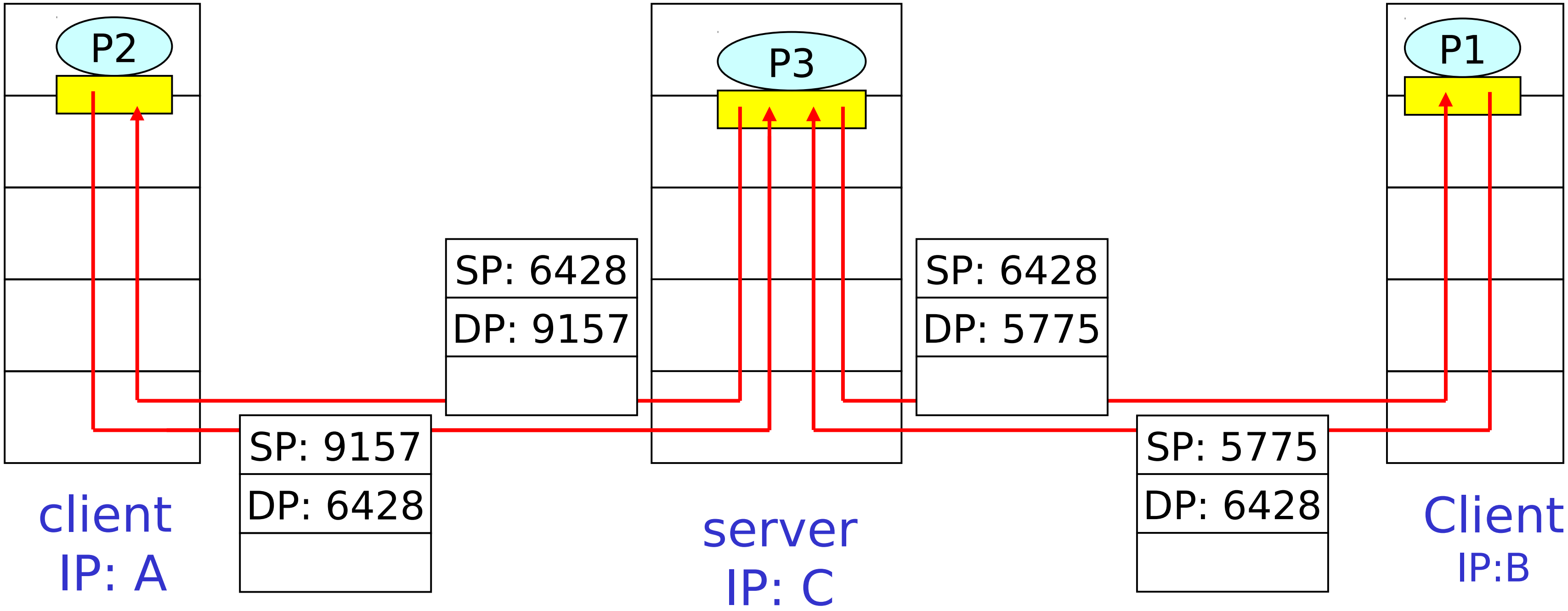
Operating System Integration

- OS implements transport protocol
  - handles asynchronous execution
  - provides send/receive queue at *socket*
    - socket: named communication endpoint
- OS *system calls*
  - create/remove sockets
  - establish names, connections
  - send/receive data

Multiplexing – Multiple Sockets

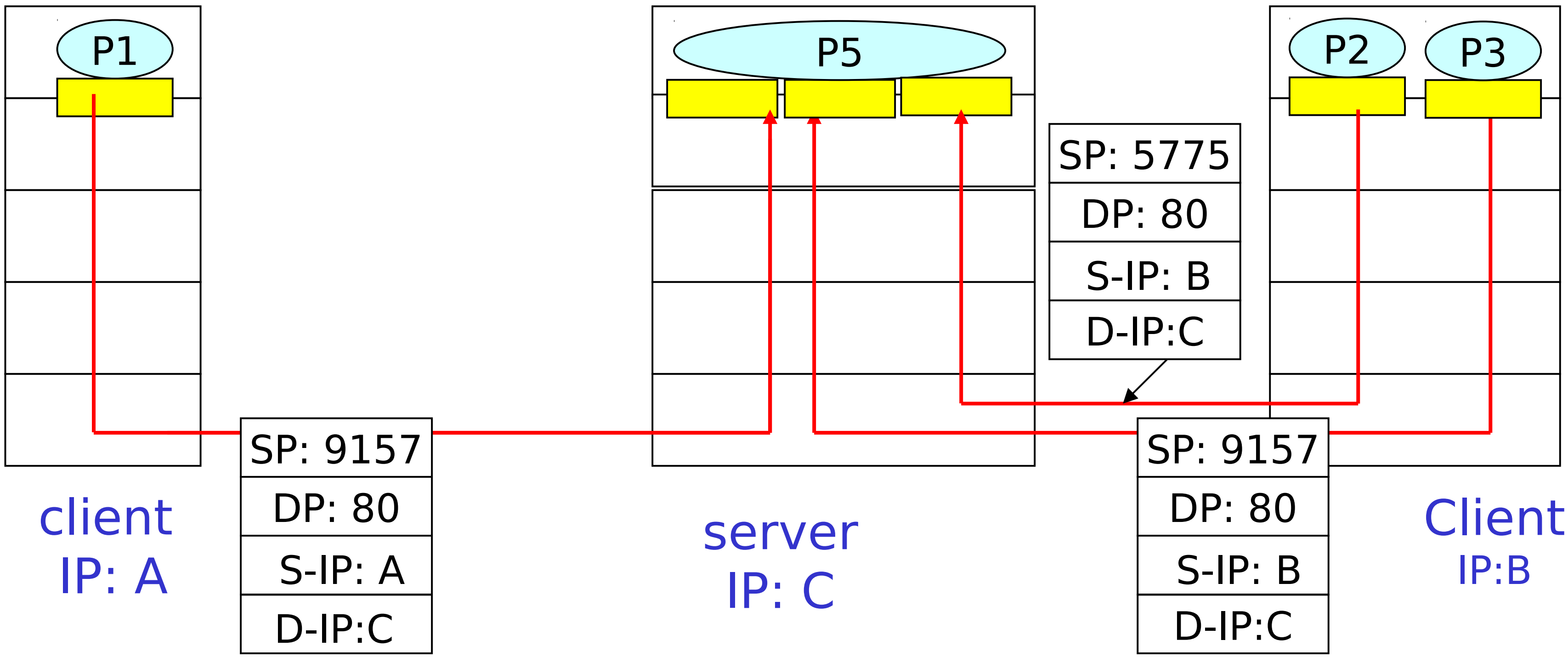


Multiplexing – Single Socket



SP provides “return address”

Multiplexing – Multiple Connections

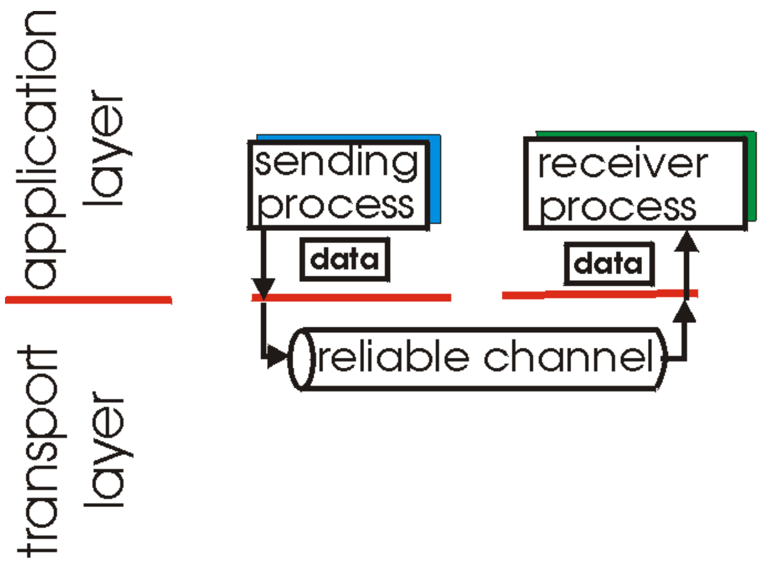


Reliable Transmission

- use acknowledgements to indicate receipt
  - sender knows data has been received
  - BUT: two-army problem
- look at functionality first, then performance

Principles of Reliable data transfer

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

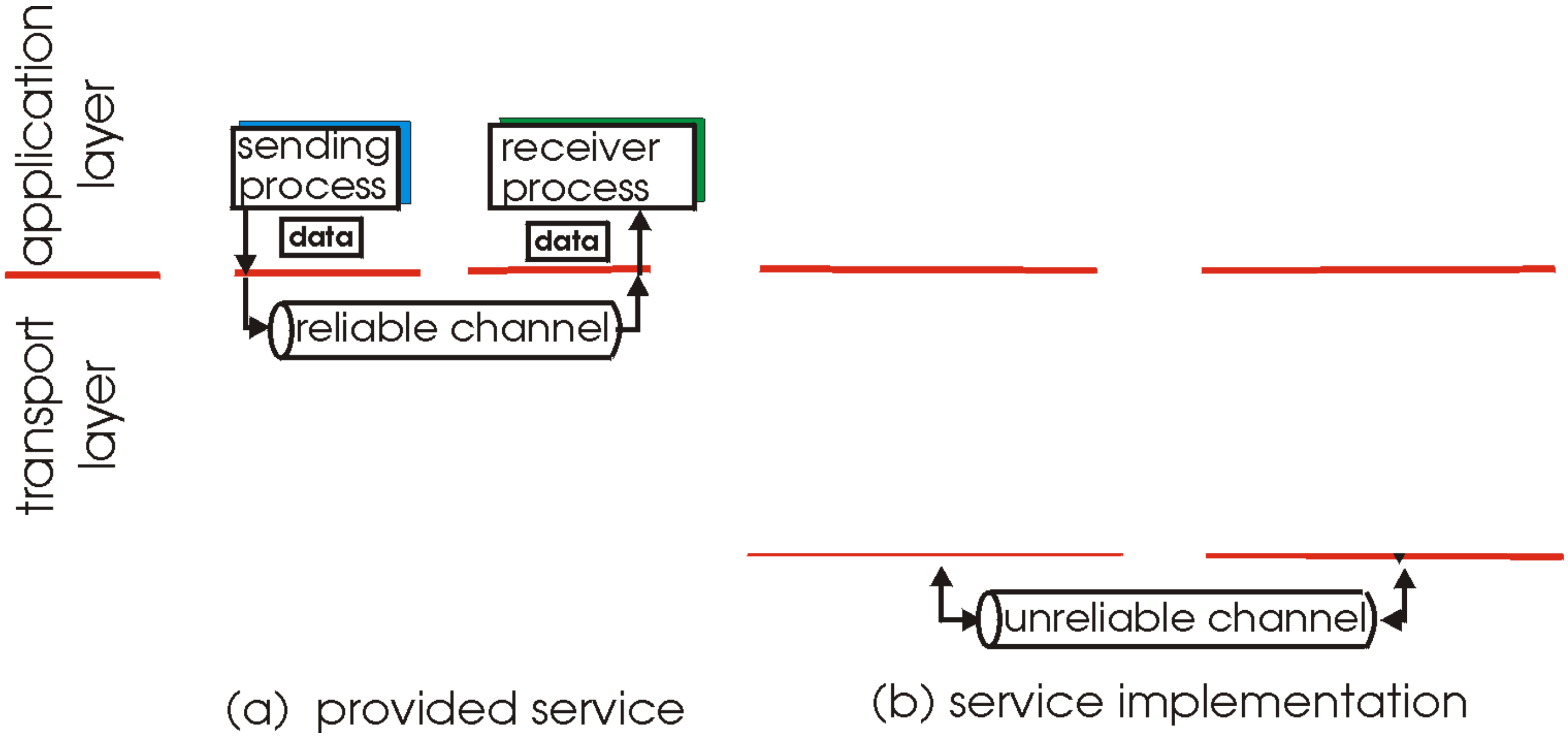


(a) provided service

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

Principles of Reliable data transfer

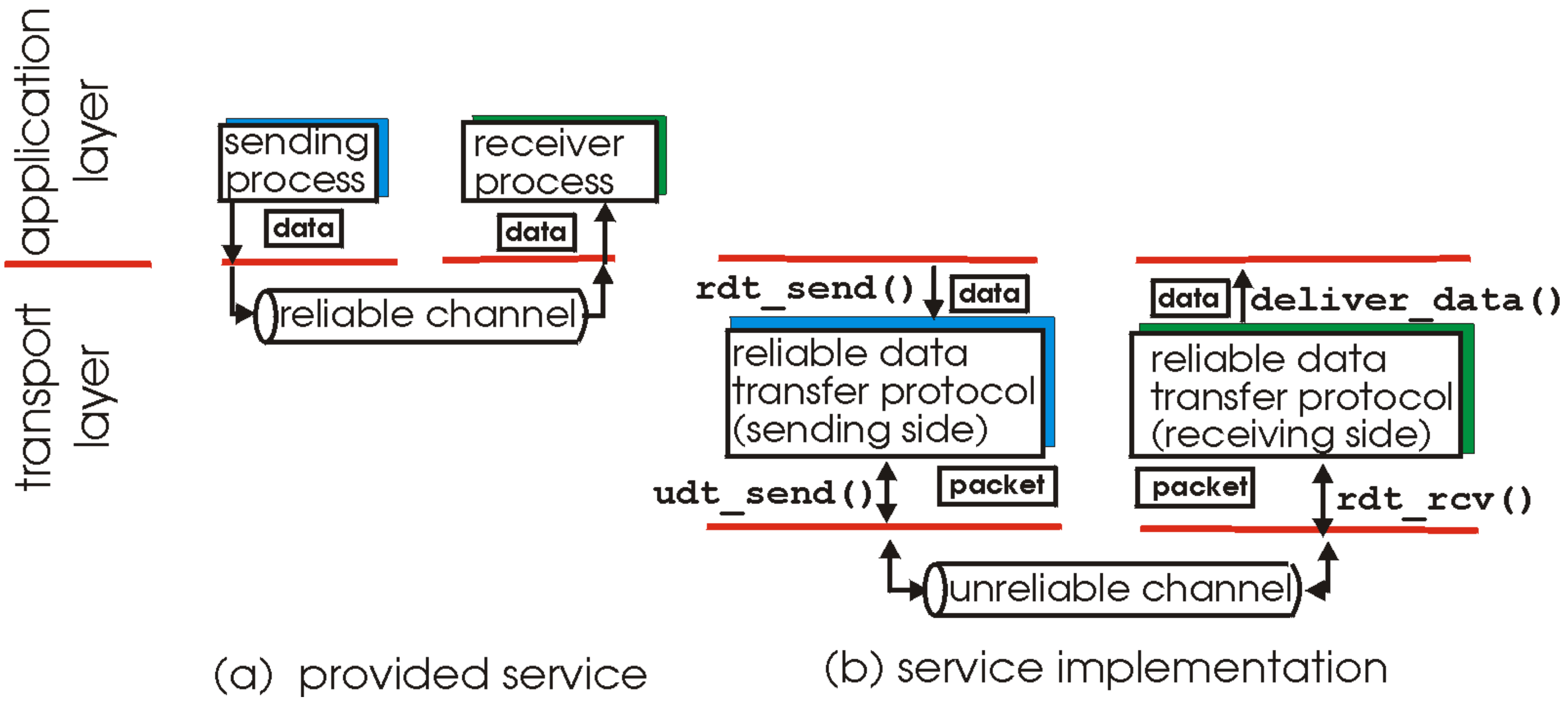
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

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

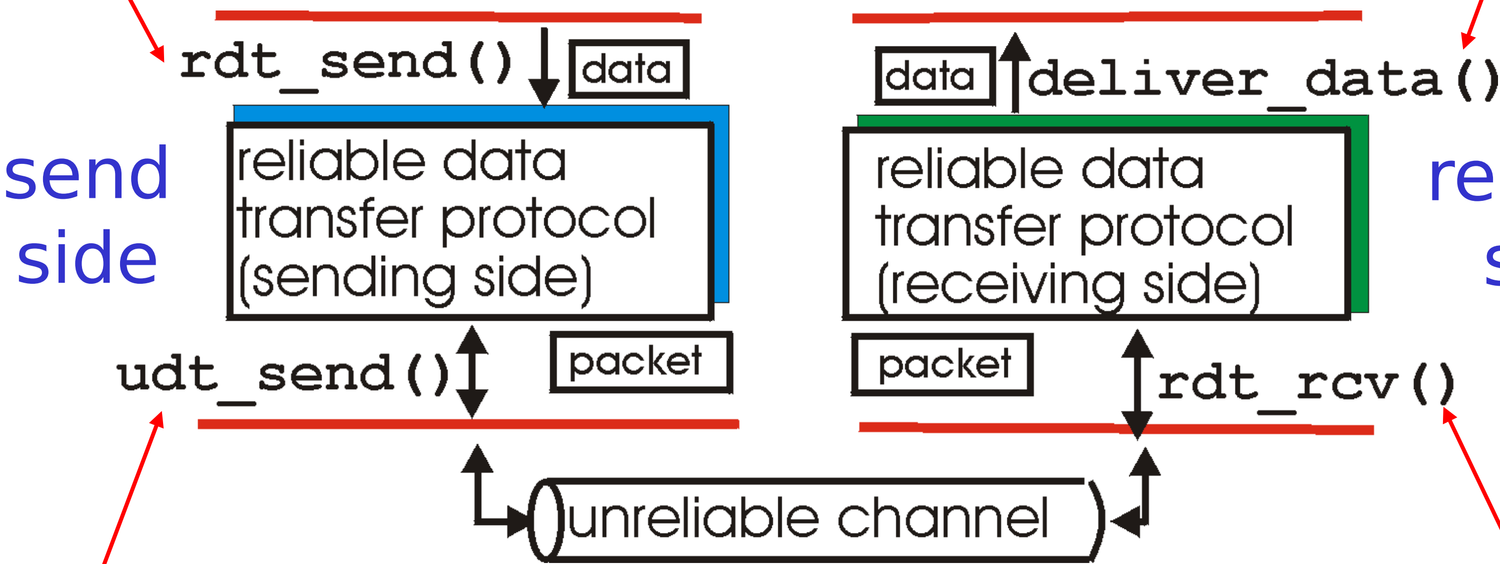


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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 layer



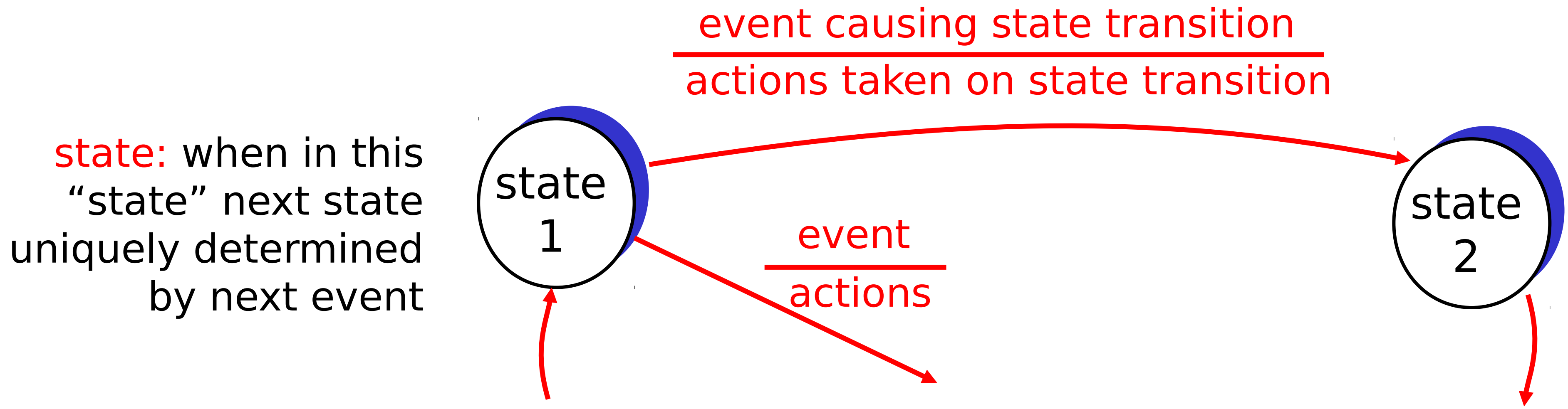
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

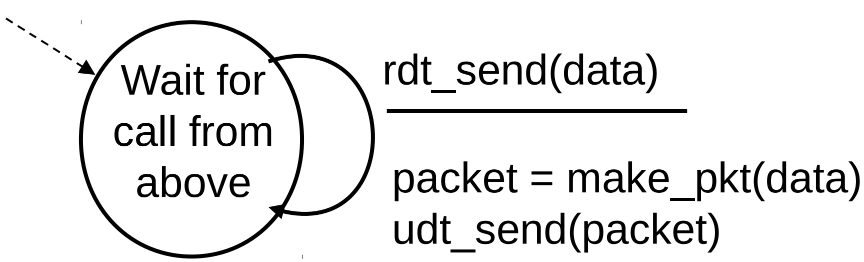
We'll:

- 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

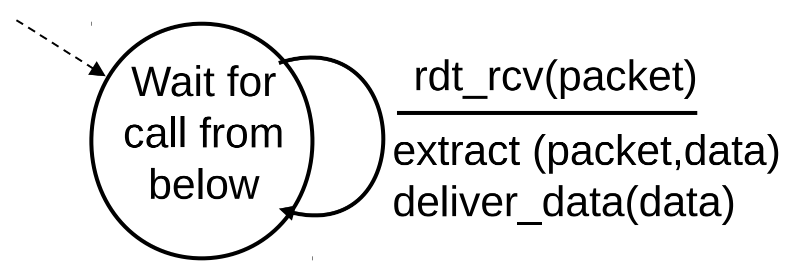


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



sender

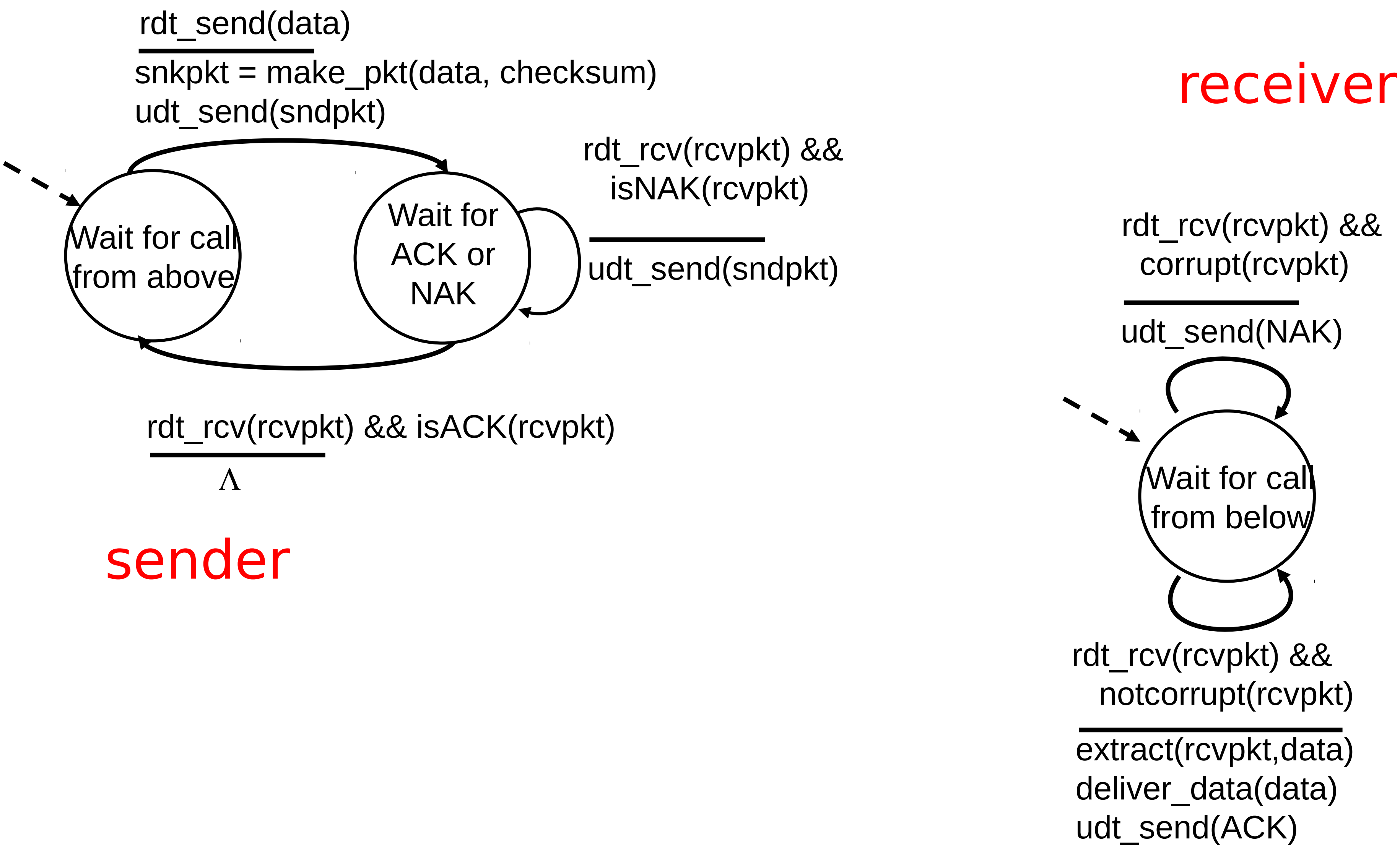


receiver

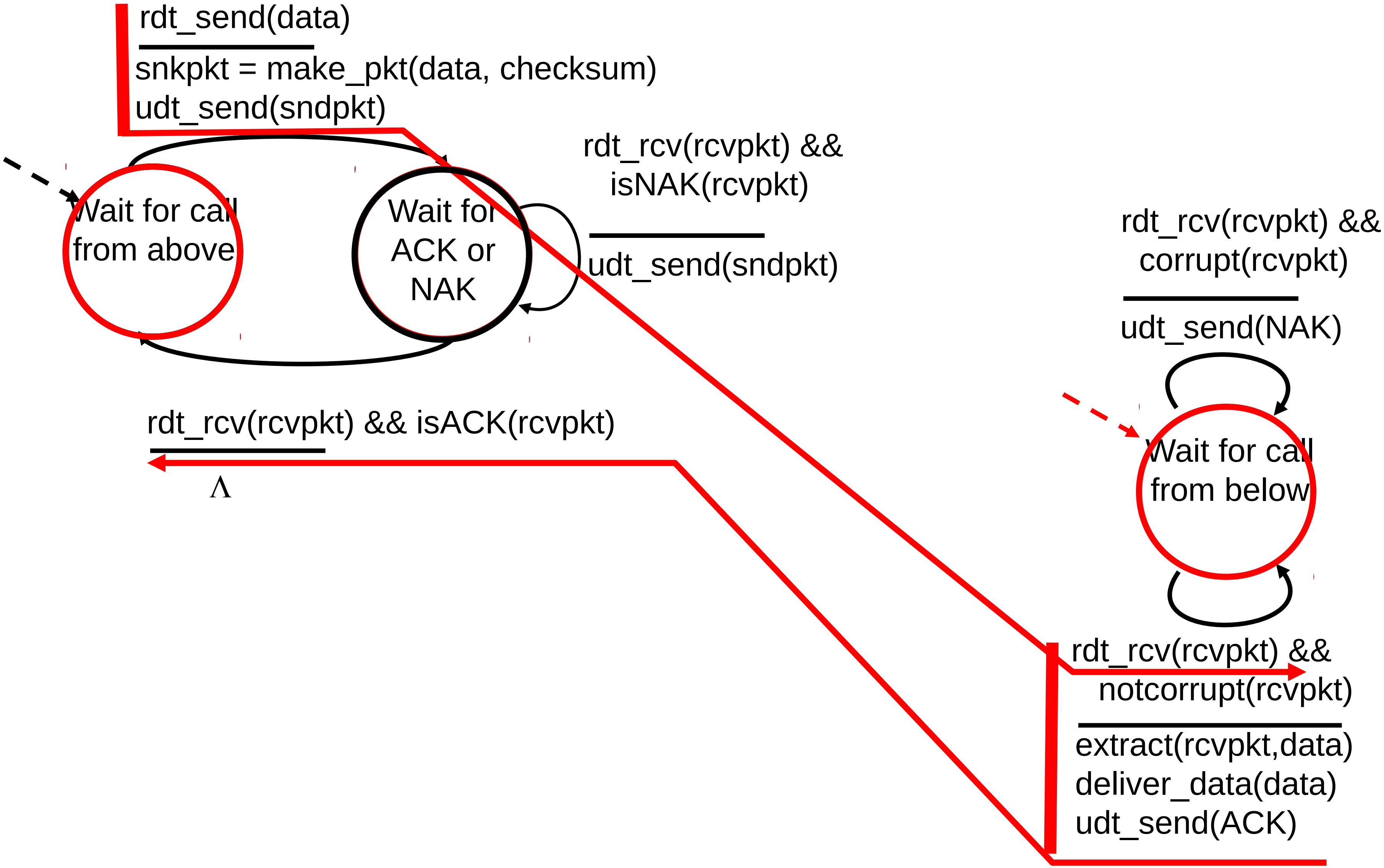
Rdt2.0: channel with bit errors

- 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
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

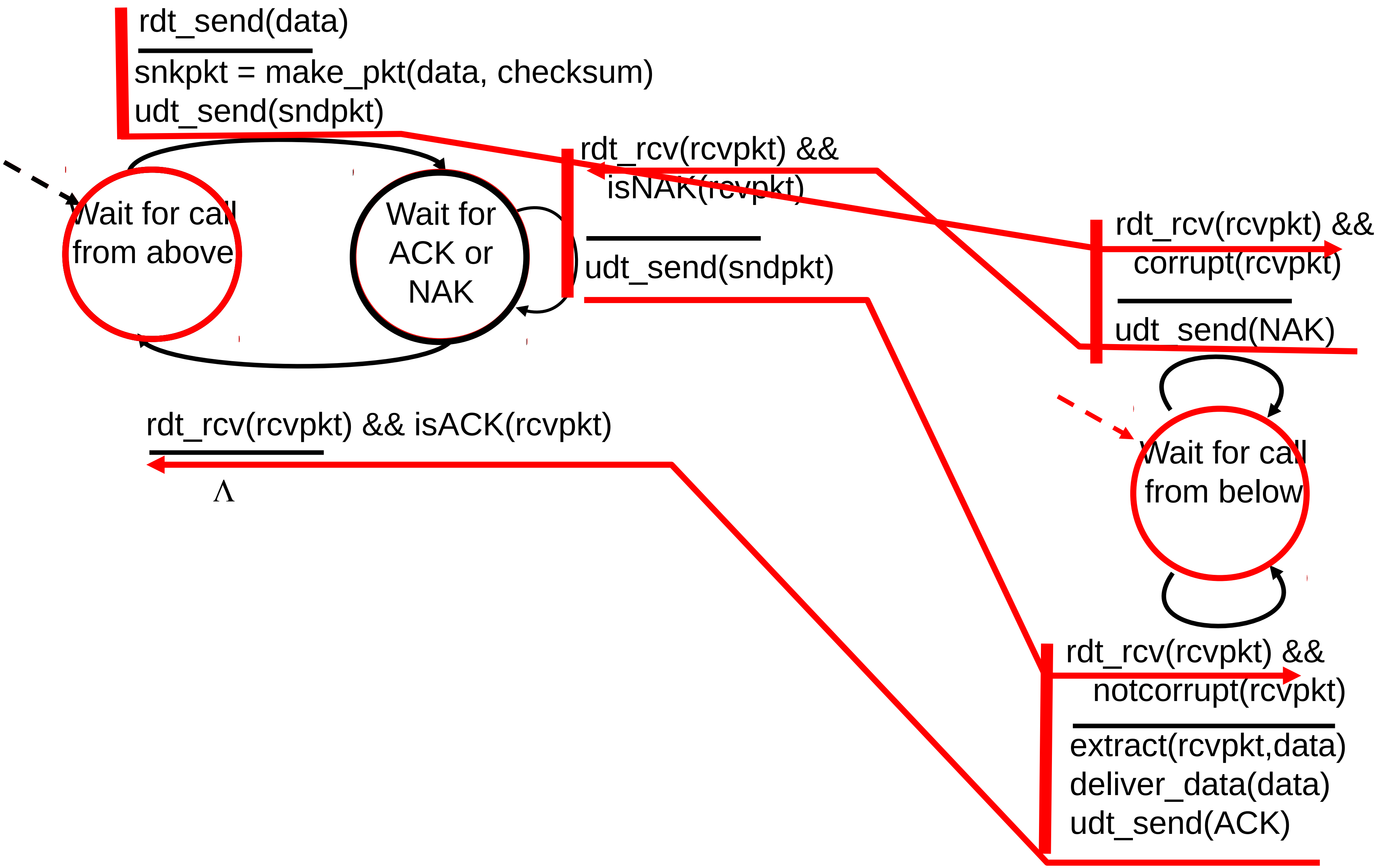
rdt2.0: FSM specification



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

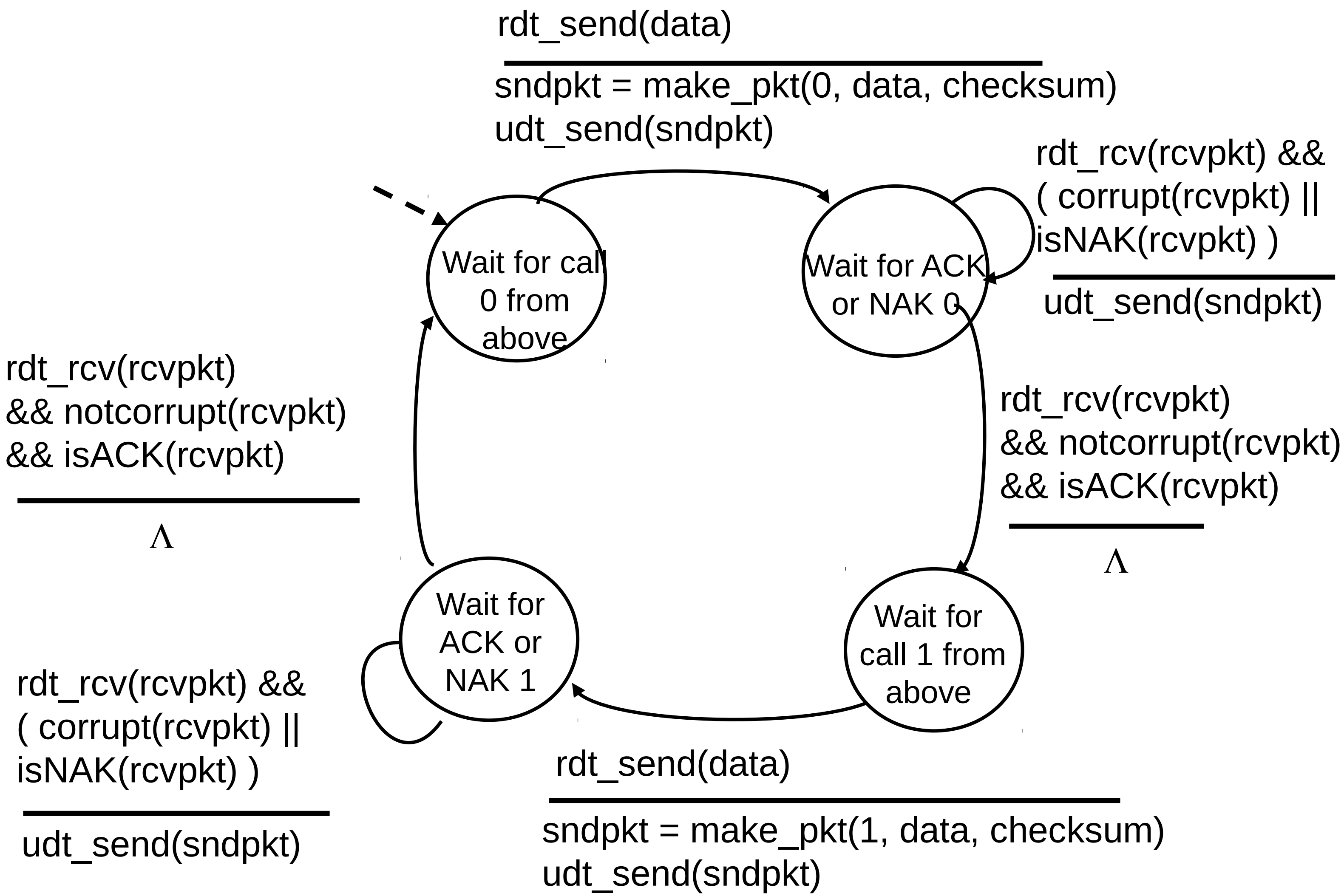
Handling duplicates:

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

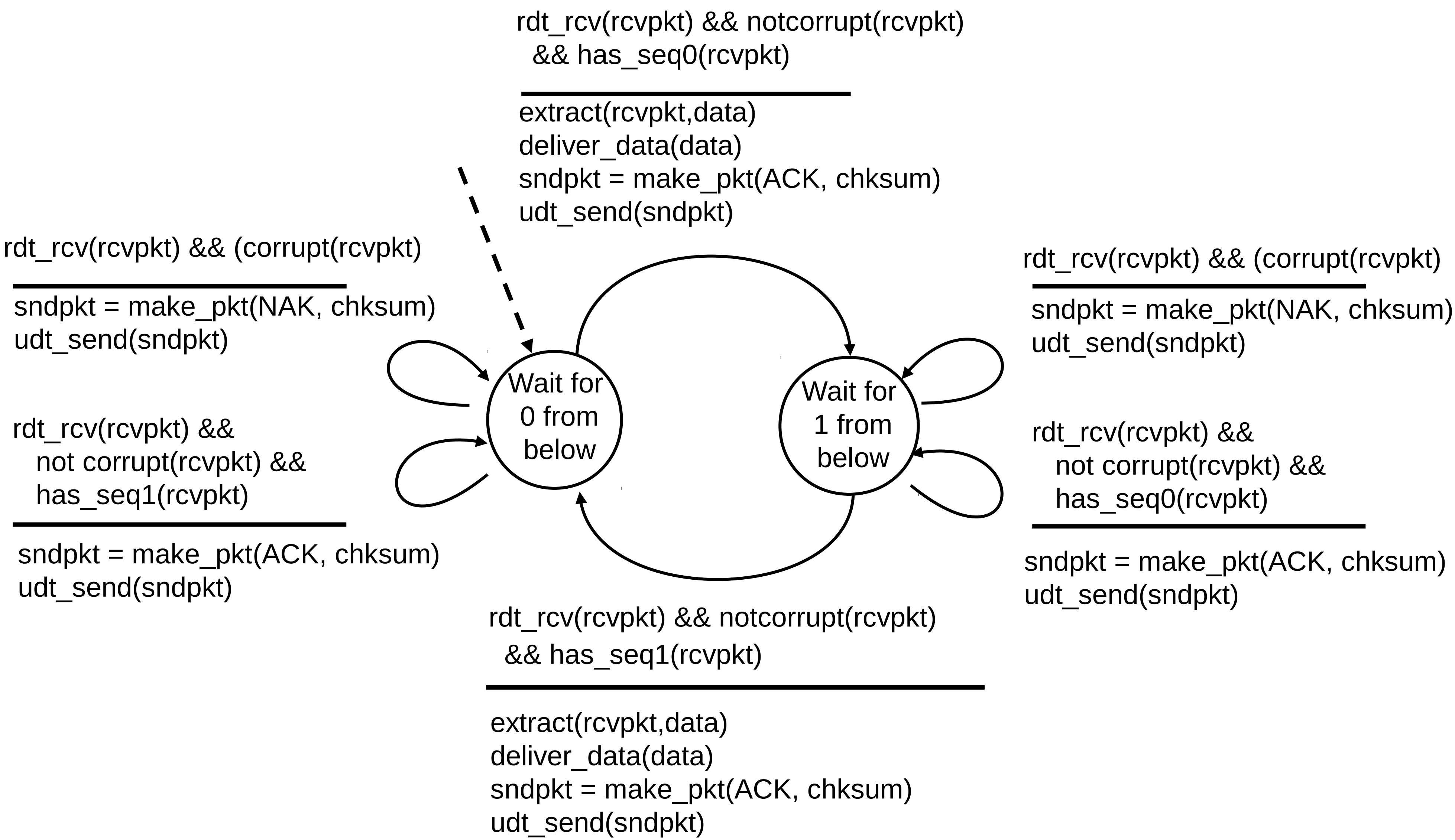
stop and wait

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 "current" pkt has 0 or 1 seq. #

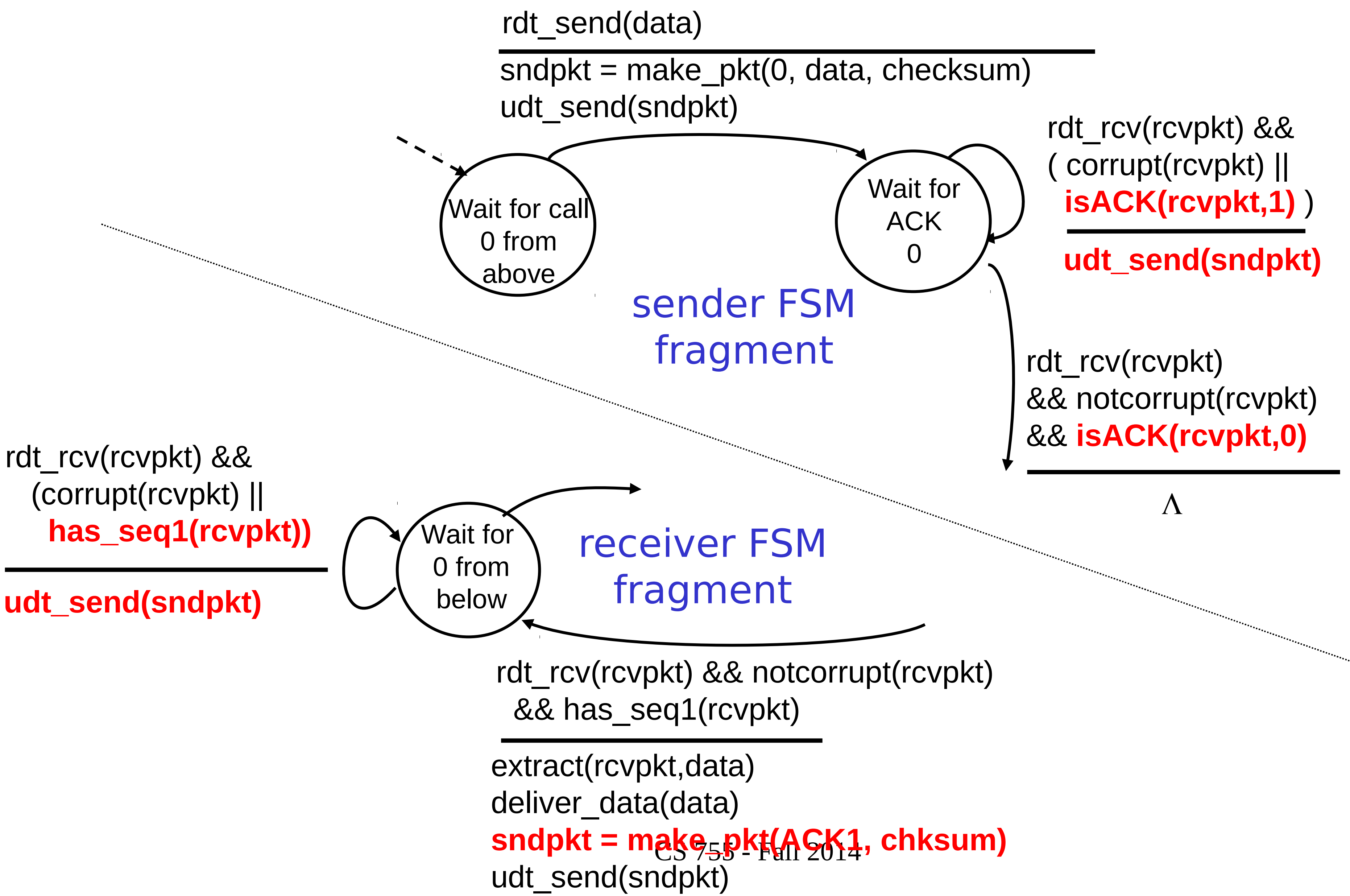
Receiver:

- 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

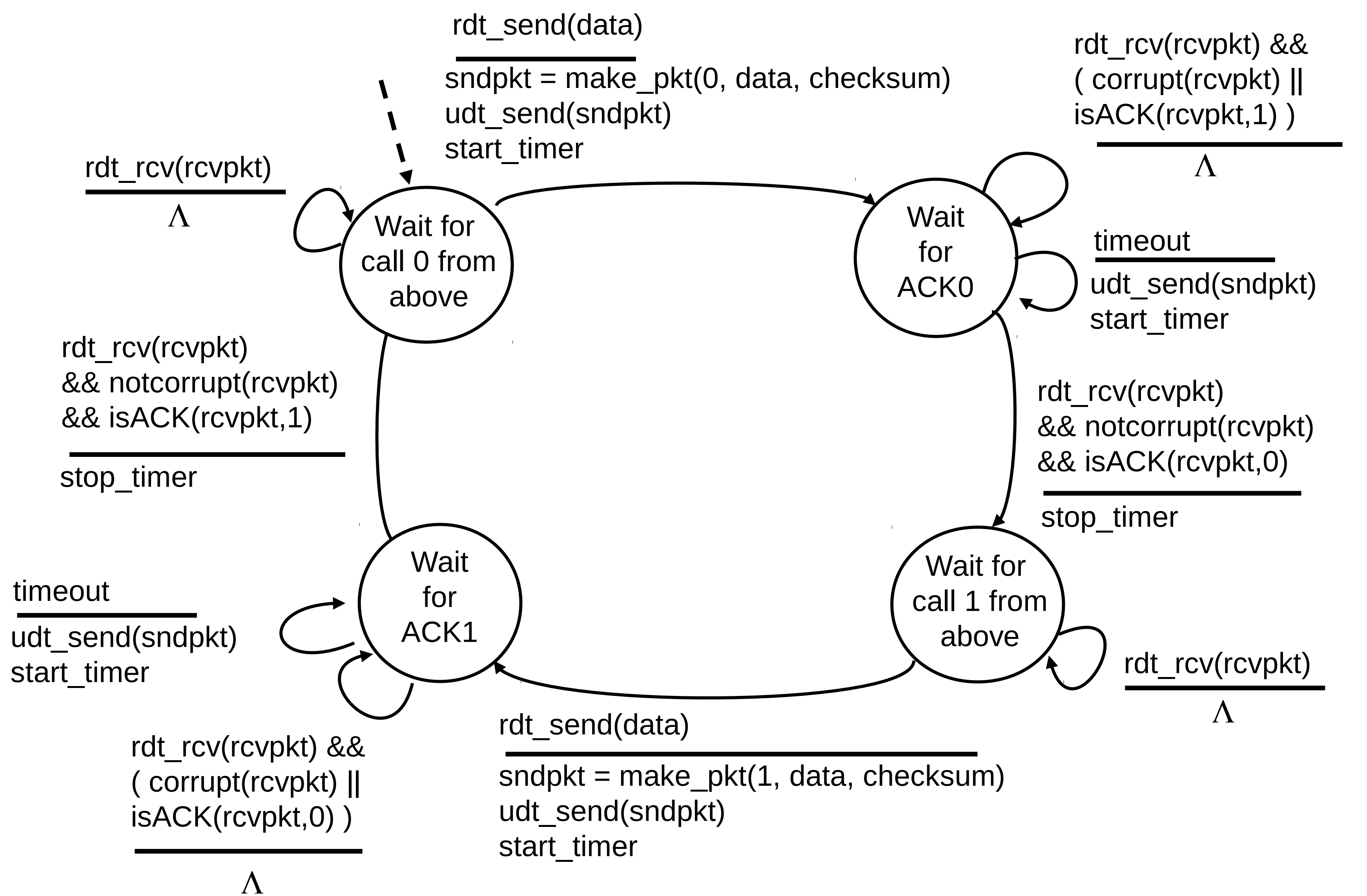
New assumption: underlying channel can also lose packets (data or ACKs)

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

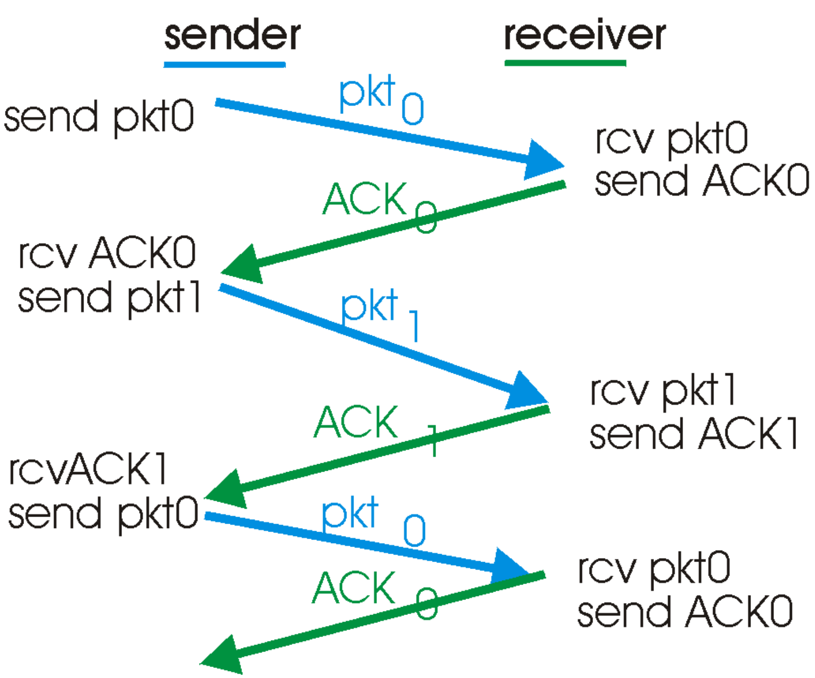
Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

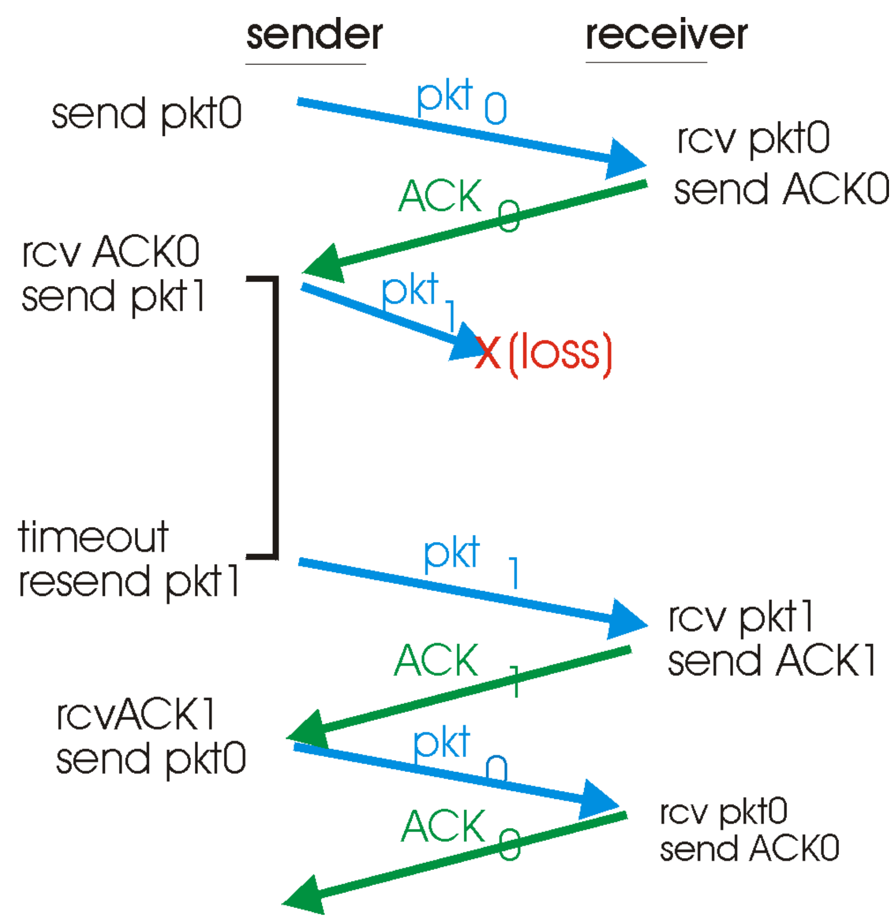
rdt3.0 sender



rdt3.0 in action

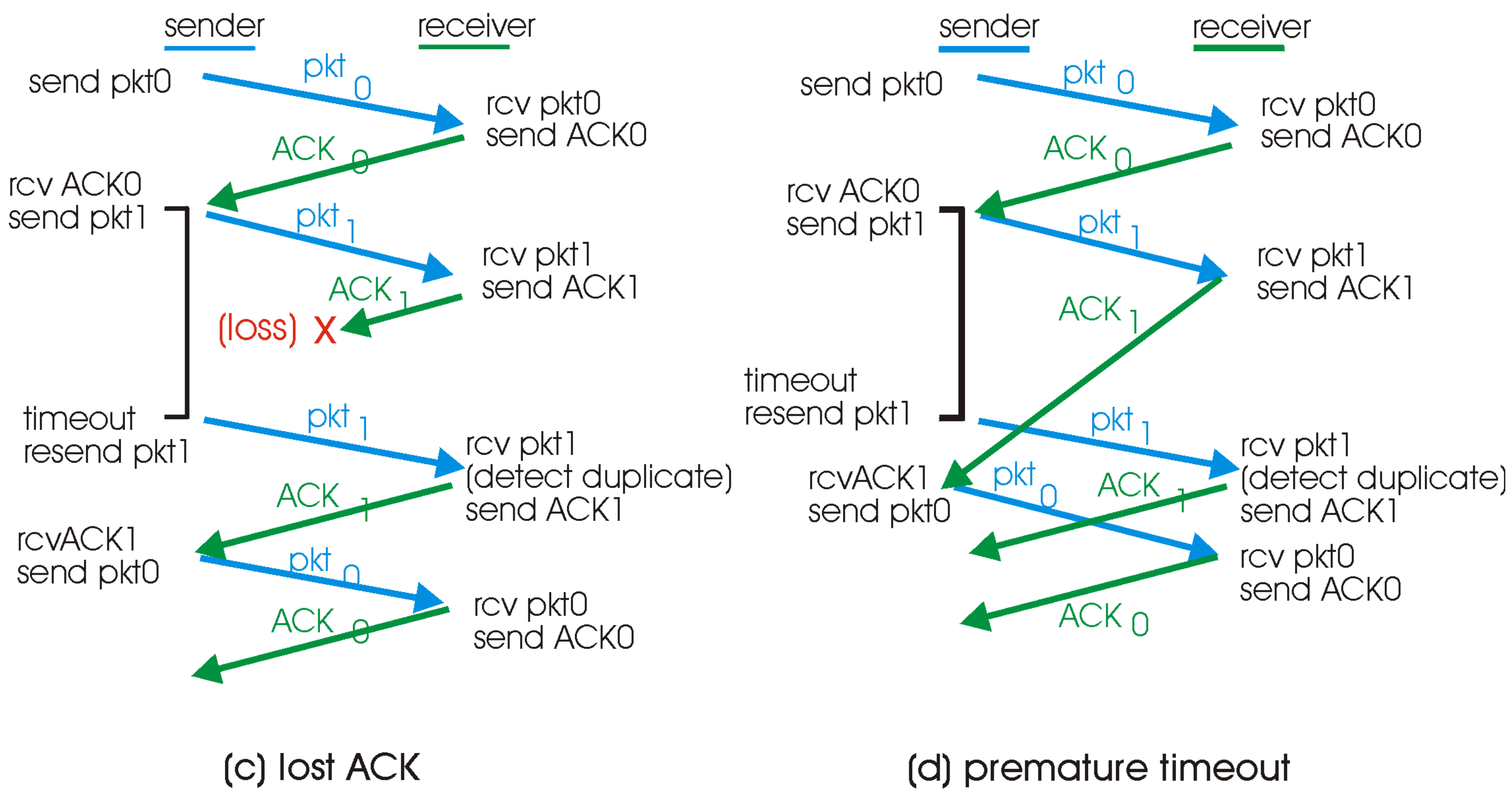


(a) operation with no loss



(b) lost packet

rdt3.0 in action



Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

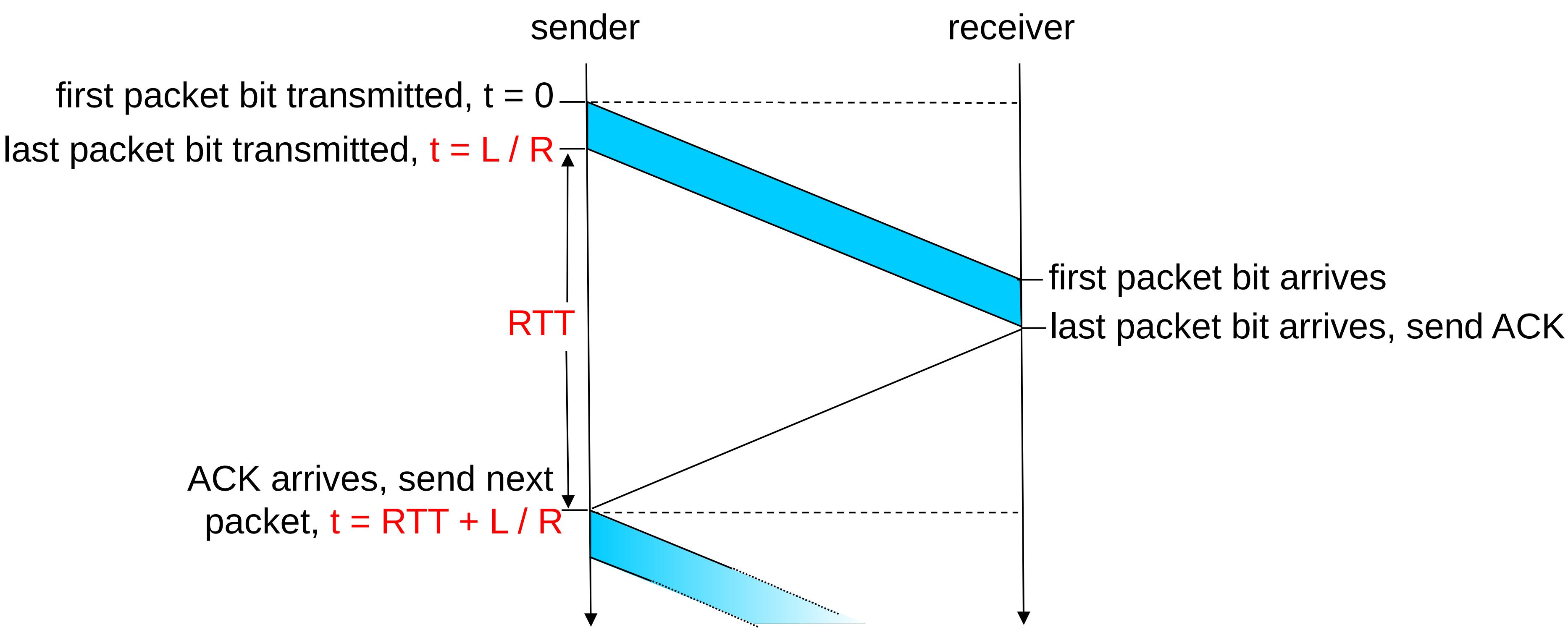
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\text{microseconds}$$

- U_{sender}: utilization – fraction of time sender busy sending

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

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

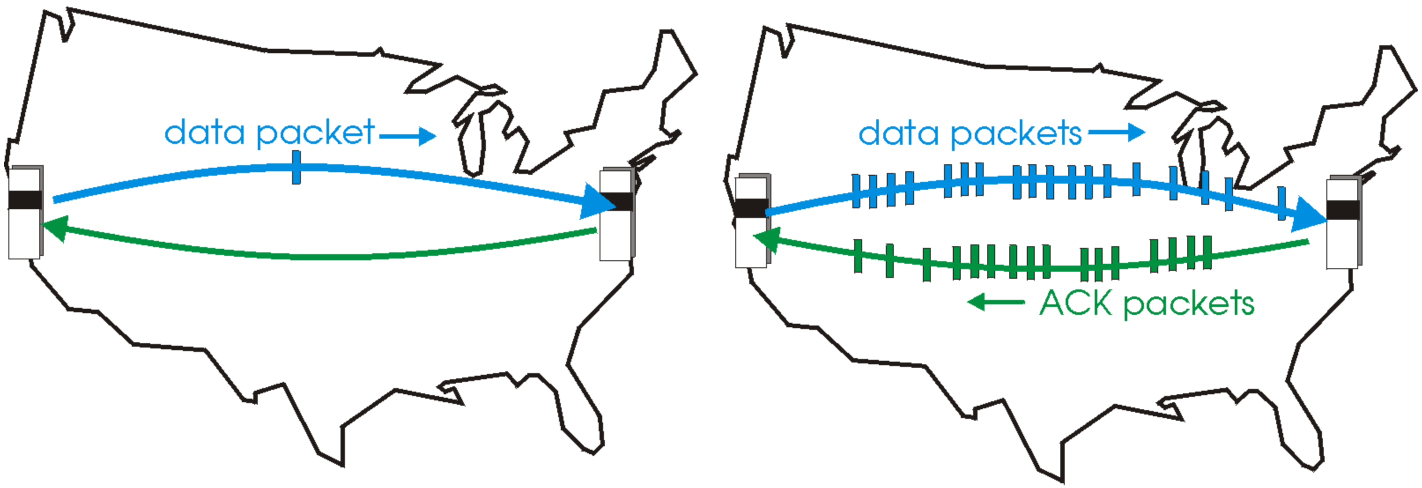


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

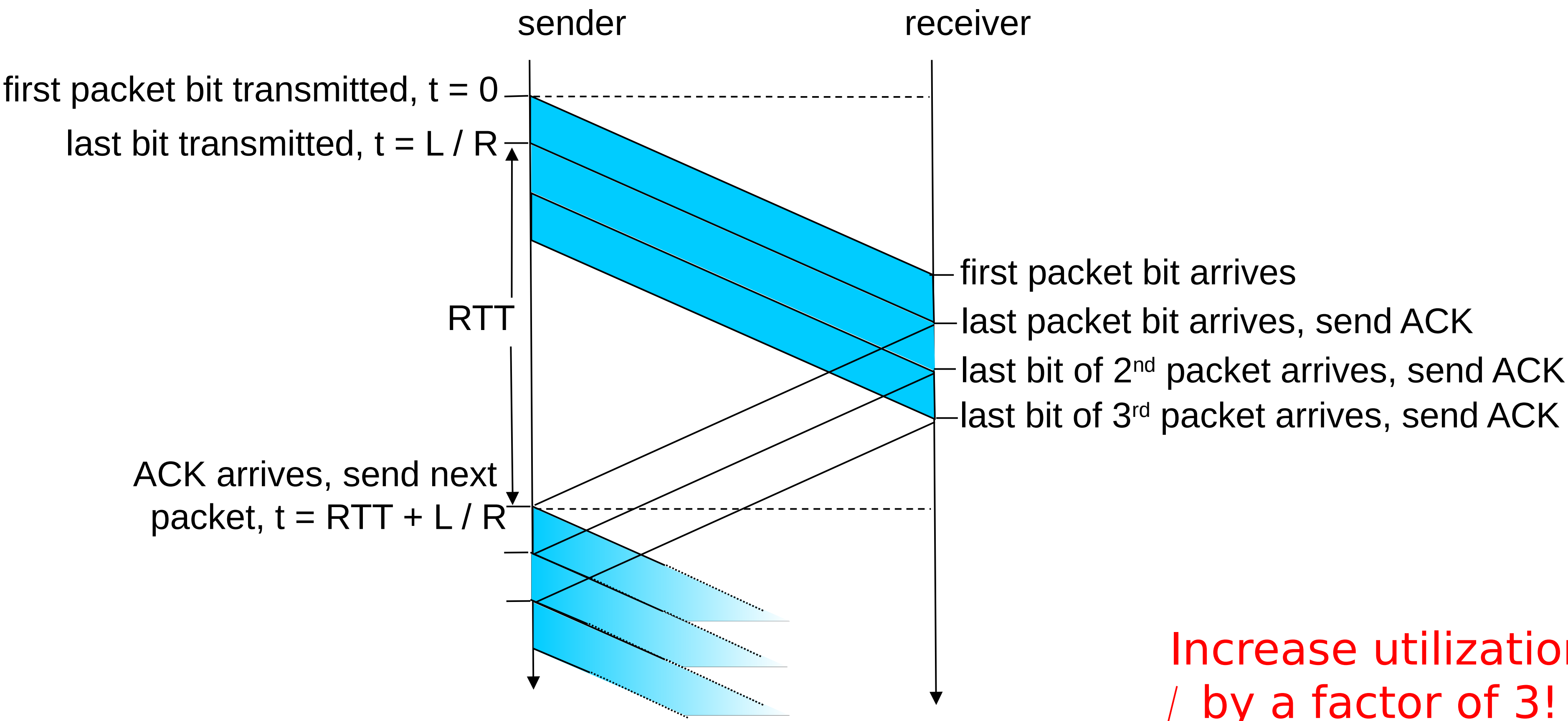


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Pipelining Protocols

Go-back-N: overview

- *sender:* up to N unACKed pkts in pipeline
- *receiver:* only sends cumulative ACKs
  - doesn't ACK pkt if there's a gap
- *sender:* has timer for oldest unACKed pkt
  - if timer expires: retransmit all unACKed packets

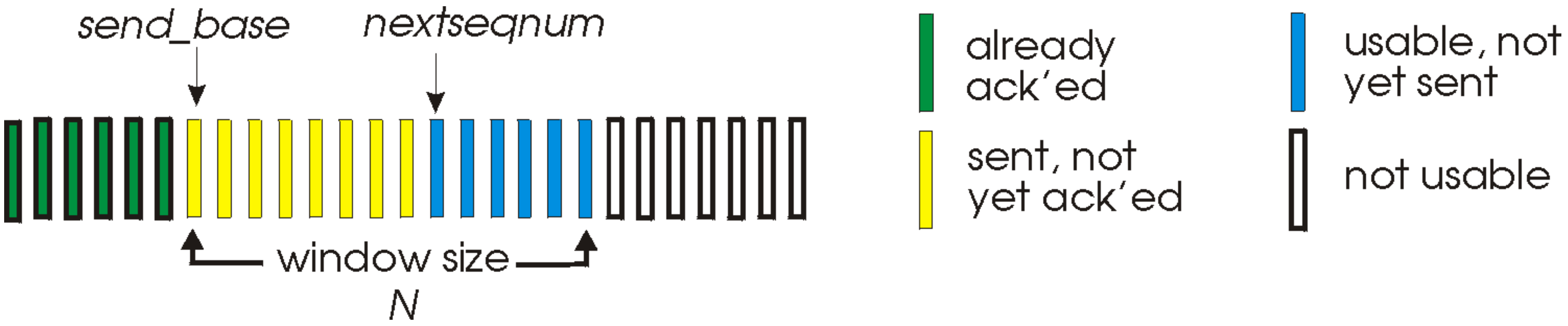
Selective Repeat: overview

- *sender:* up to N unACKed packets in pipeline
- *receiver:* ACKs individual pkts
- *sender:* maintains timer for each unACKed pkt
  - if timer expires: retransmit only unACKed packet

Go-Back-N

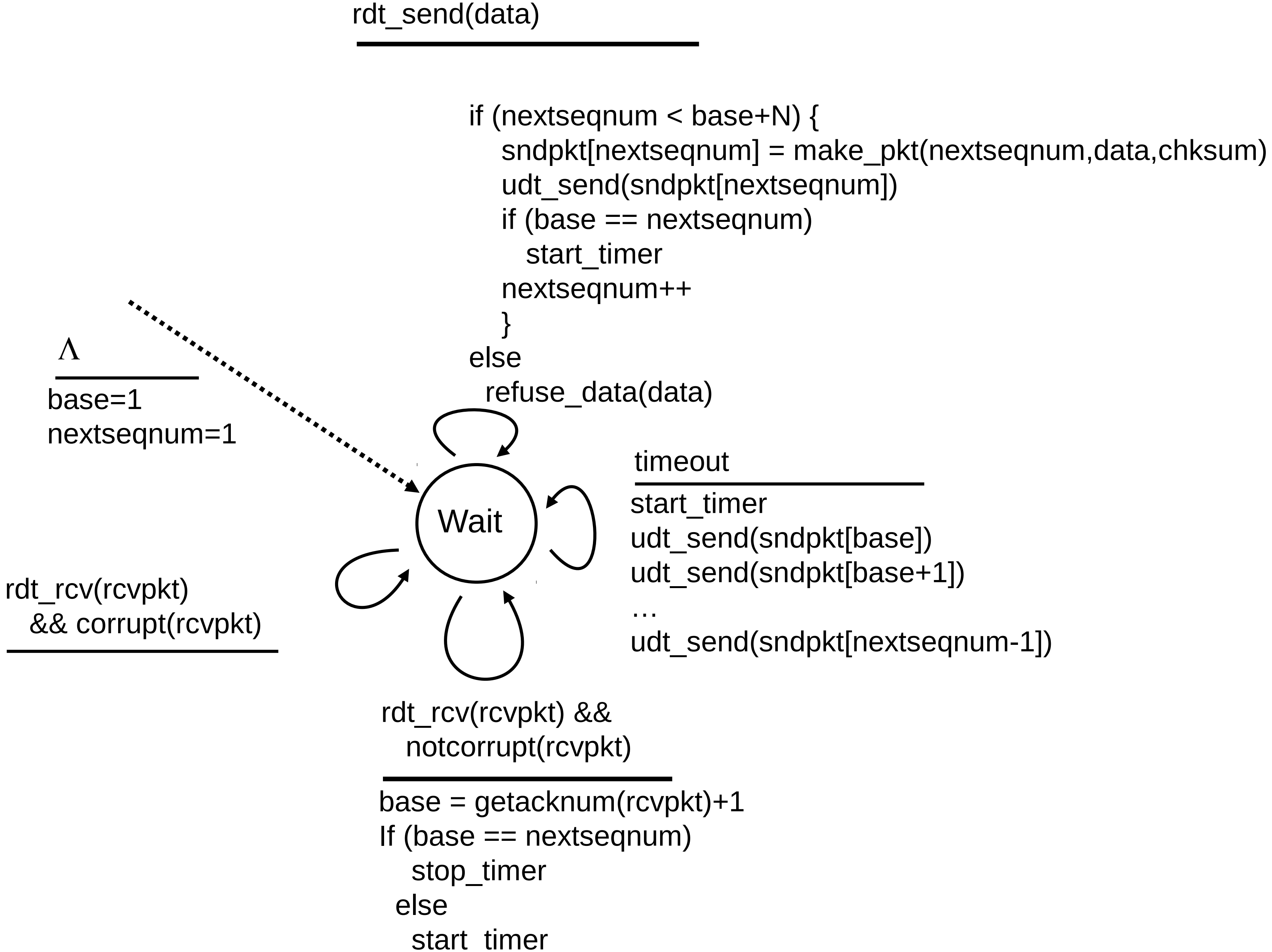
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unACKed pkts allowed

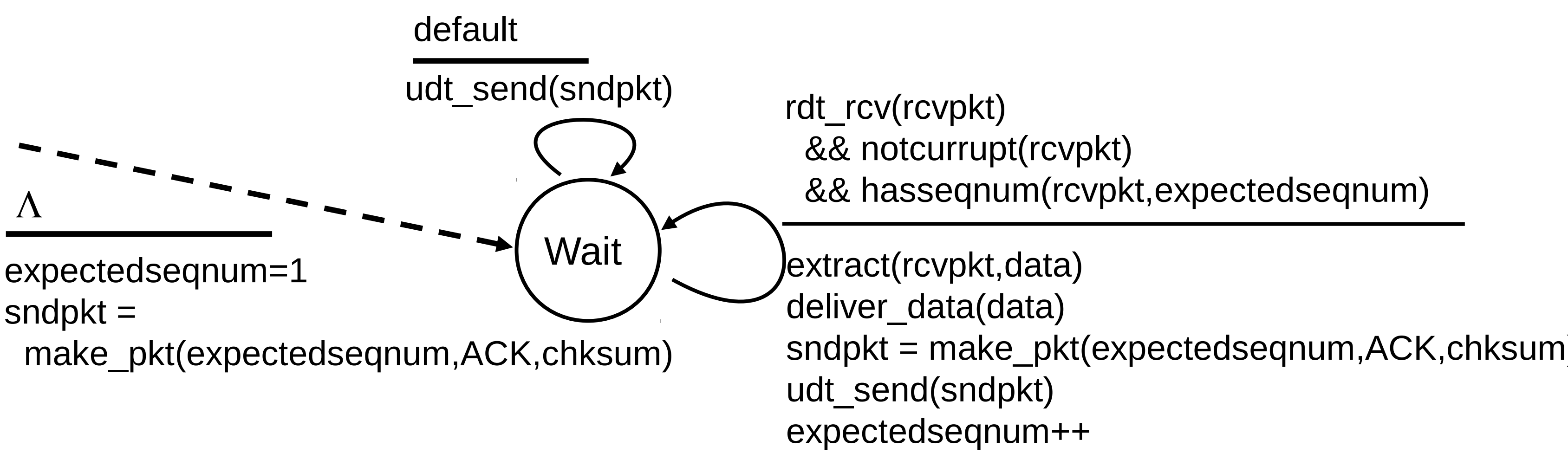


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

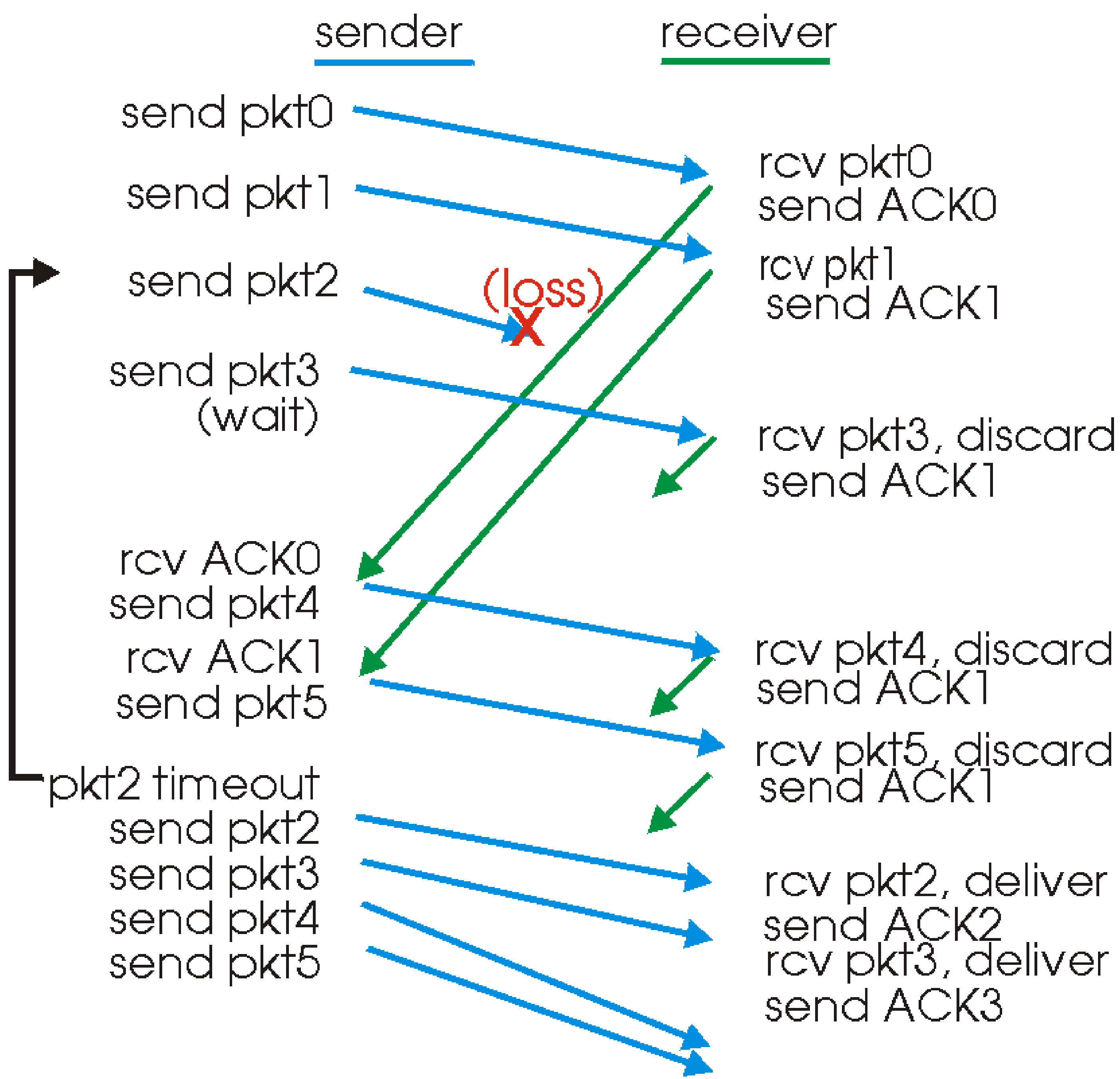


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

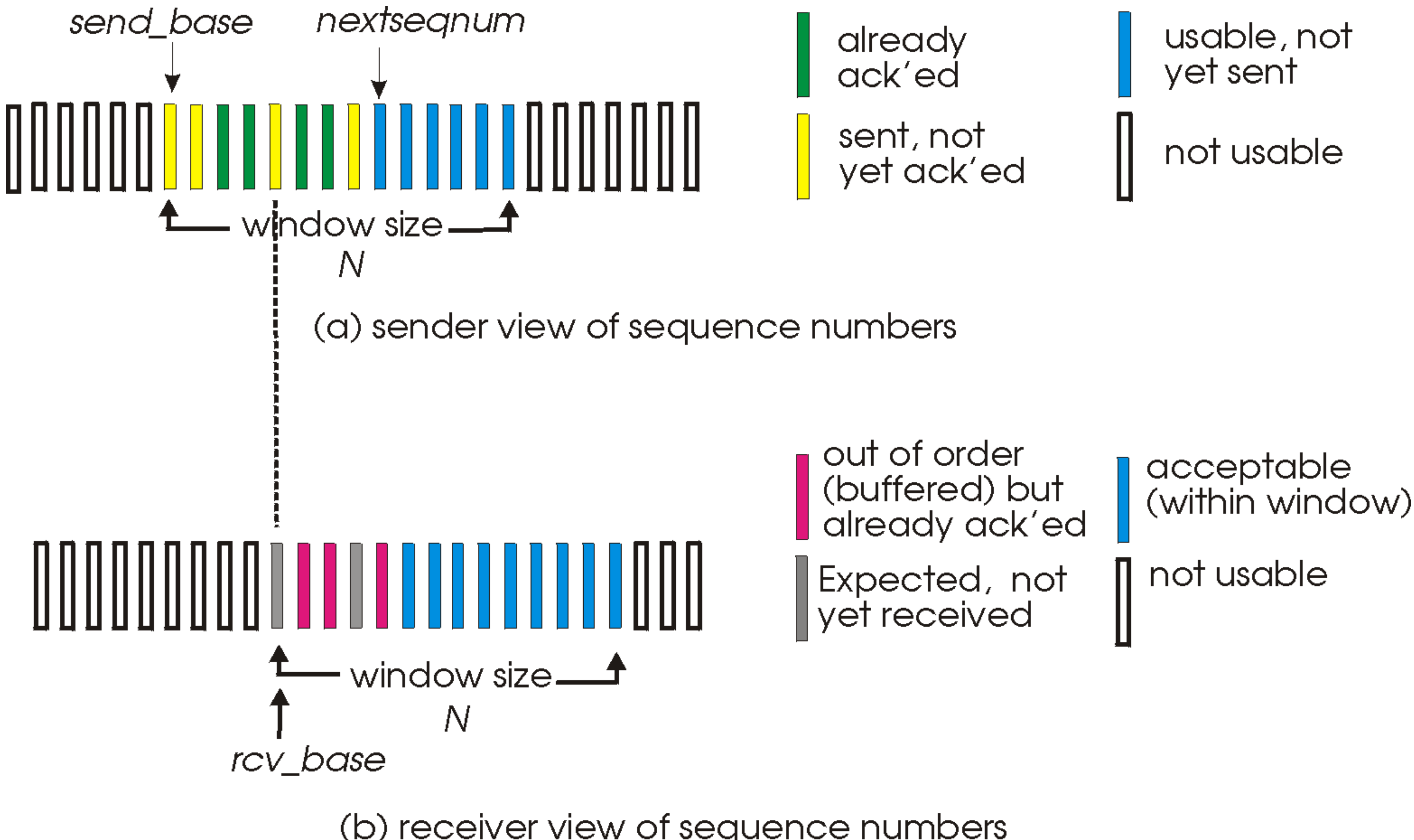
GBN in action



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
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



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]:

- 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

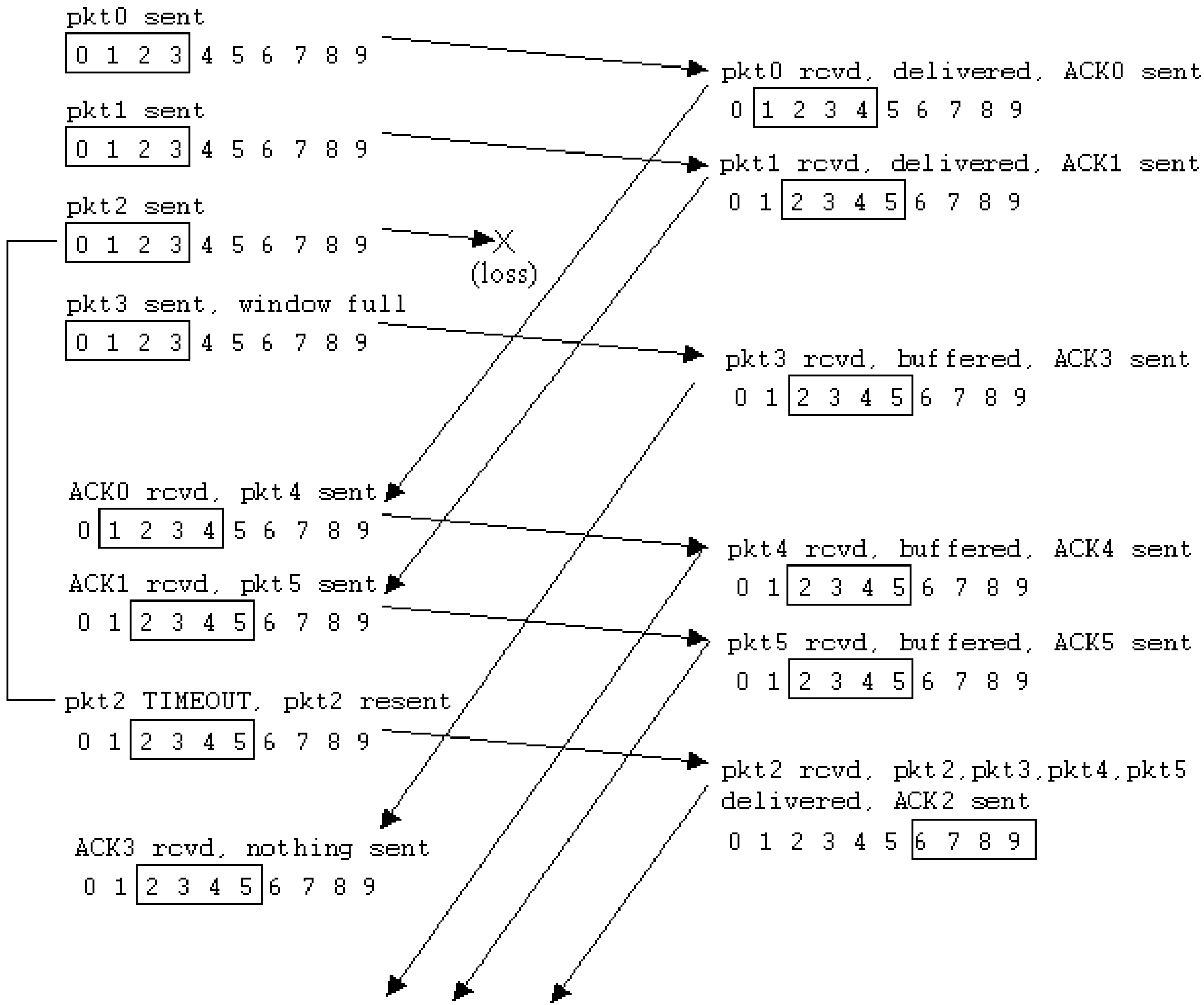
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



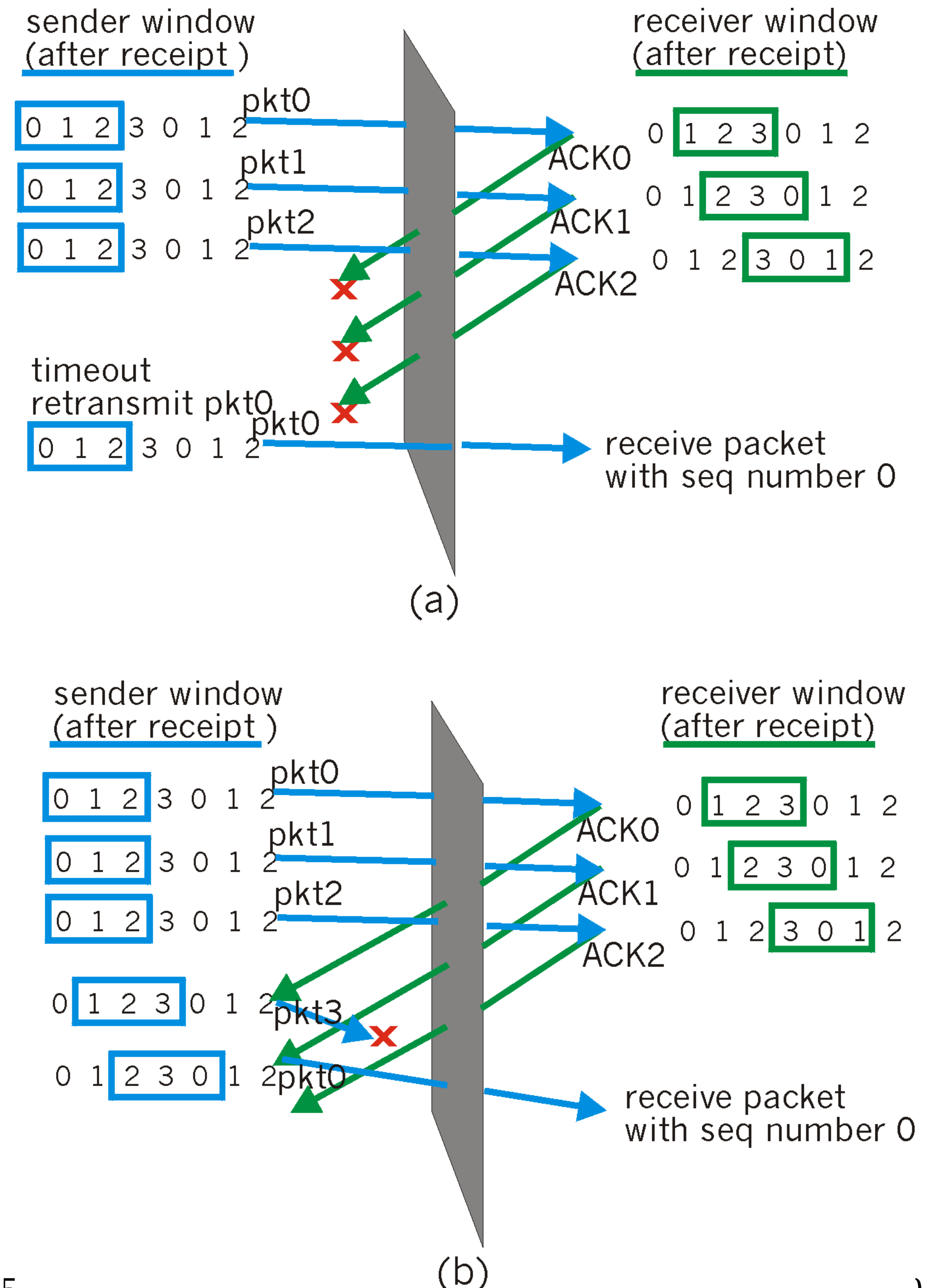
Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

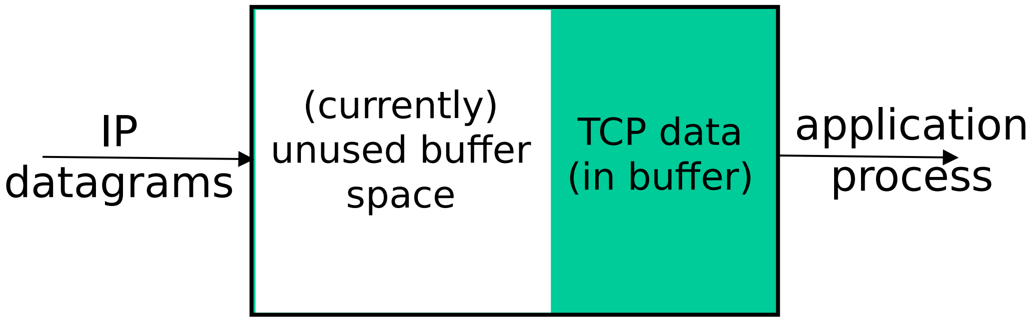


Flow Control: TCP

- receive side of TCP connection has a receive buffer:

flow control

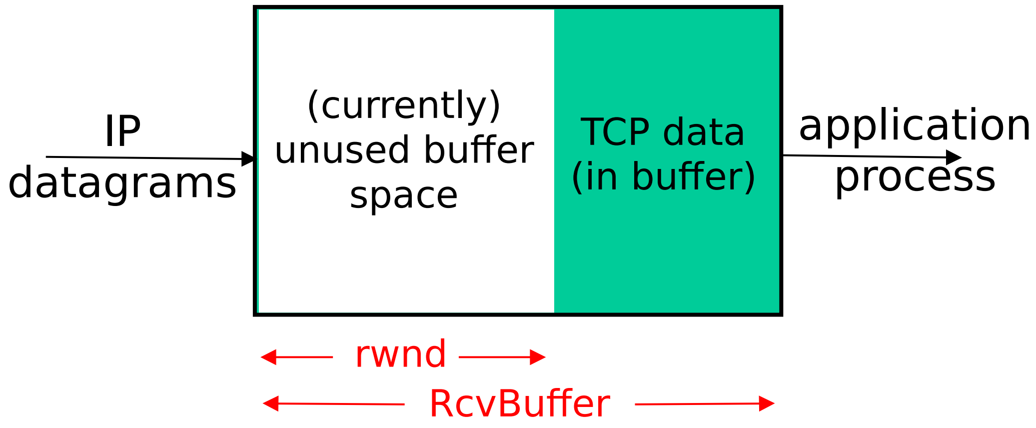
sender won't overflow receiver's buffer by transmitting too much, too fast



- *speed-matching service:* matching send rate to receiving application's drain rate

- app process may be slow at reading from buffer

TCP Flow Control: how it works



(suppose TCP receiver discards out-of-order segments)

- unused buffer space:

= `rwnd`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- receiver: advertises unused buffer space by including **`rwnd`** value in segment header
- sender: limits # of unACKed bytes to `rwnd`
  - guarantees receiver's buffer doesn't overflow

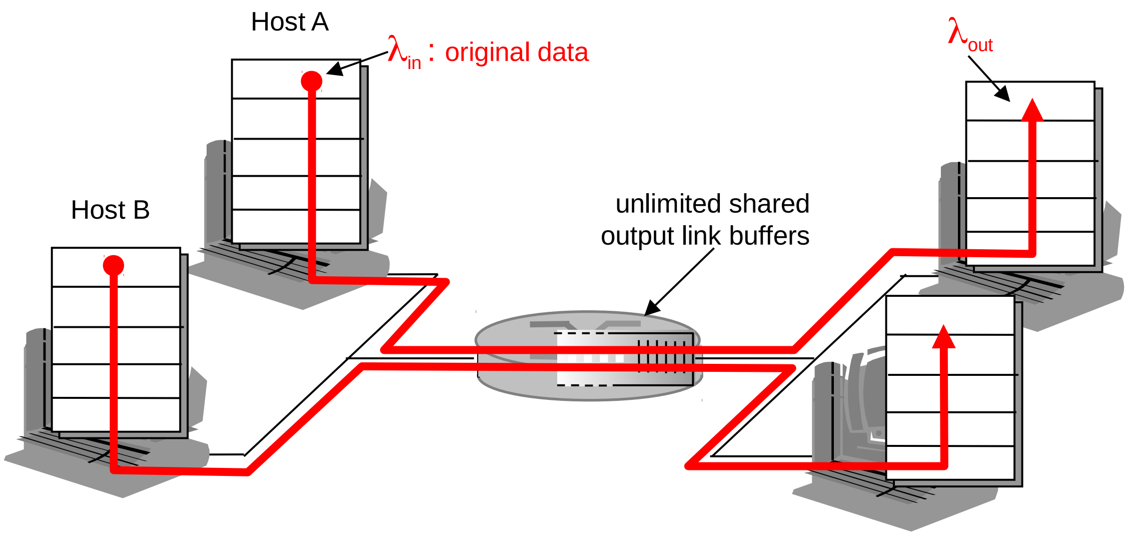
Congestion Control

- decoupled network and transport service: multiple senders might overwhelm routers => packet delay and loss
- certain situations: congestion collapse

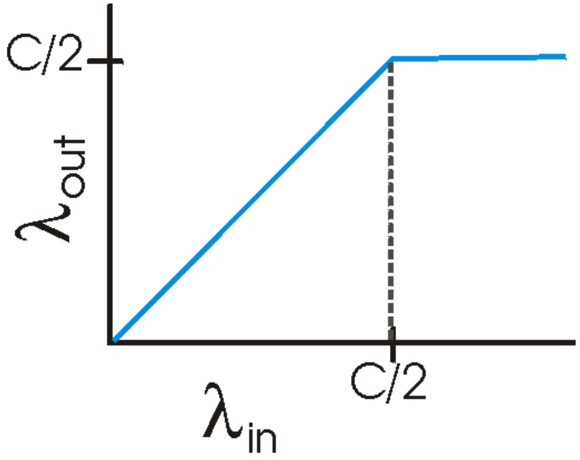
- another goal: “fair” sharing of network resources

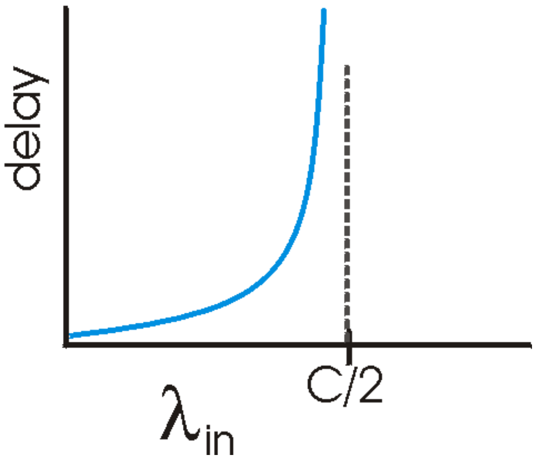
Overload without Reliability

- one router, infinite buffers
- no retransmission



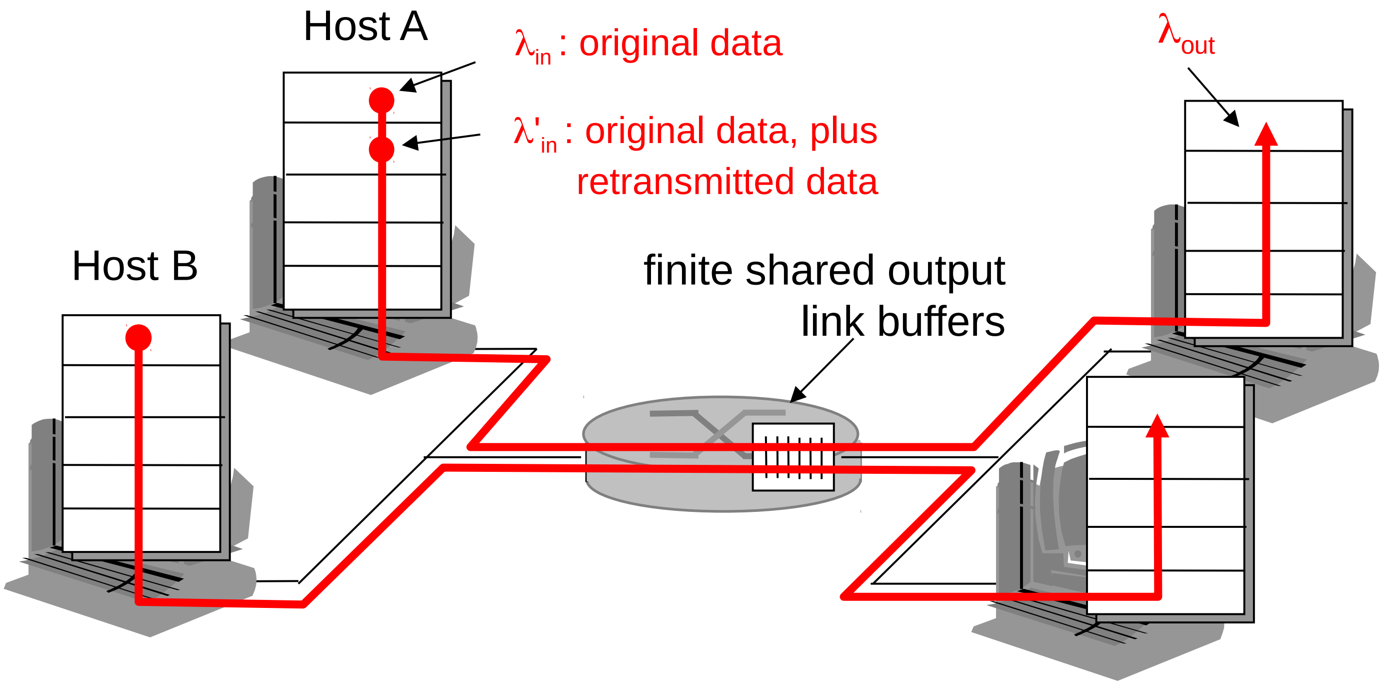
- large delays
- maximum throughput





Overload with Reliability

- one router, finite buffers
- retransmission of lost packets

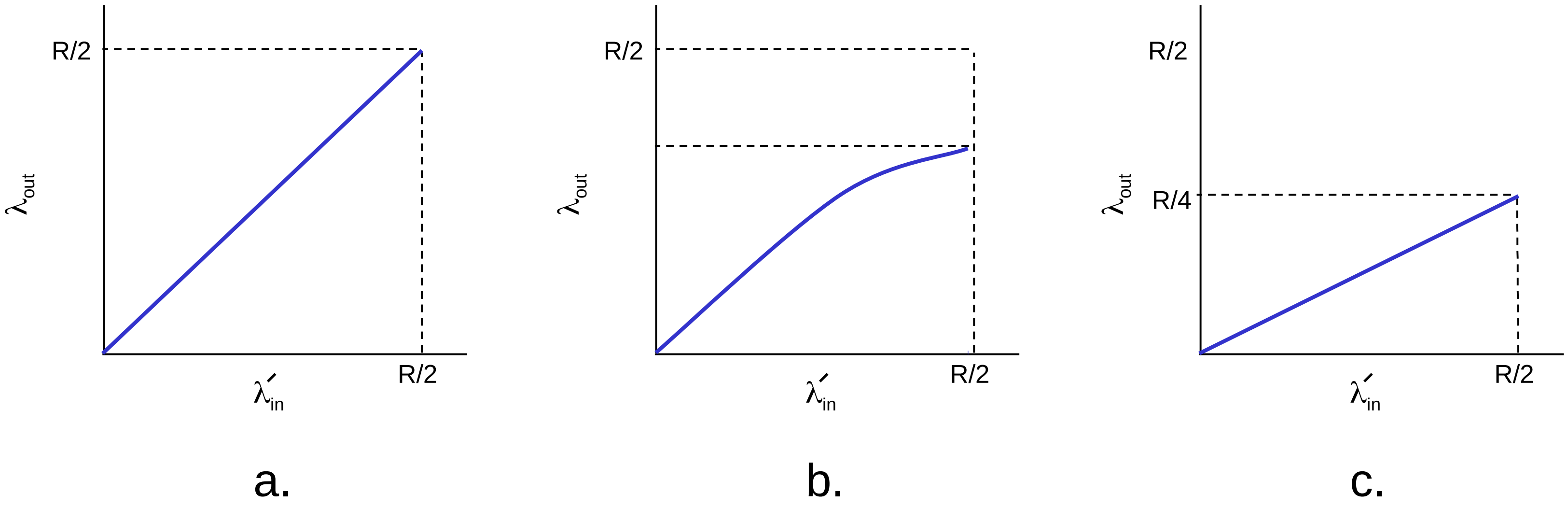


Overload with Reliability

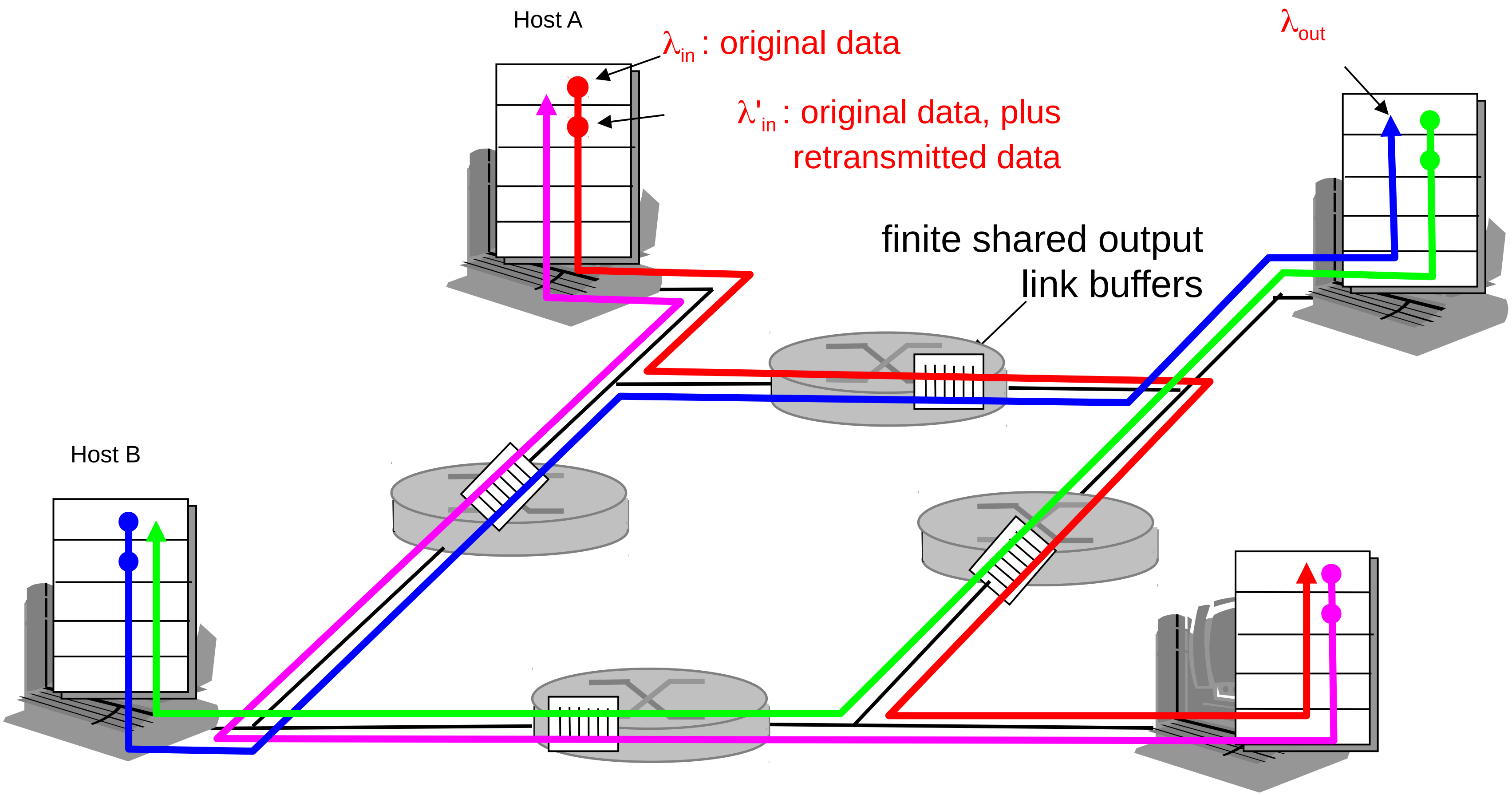
a) perfect send rate

b) finite buffer -> loss & retransmission

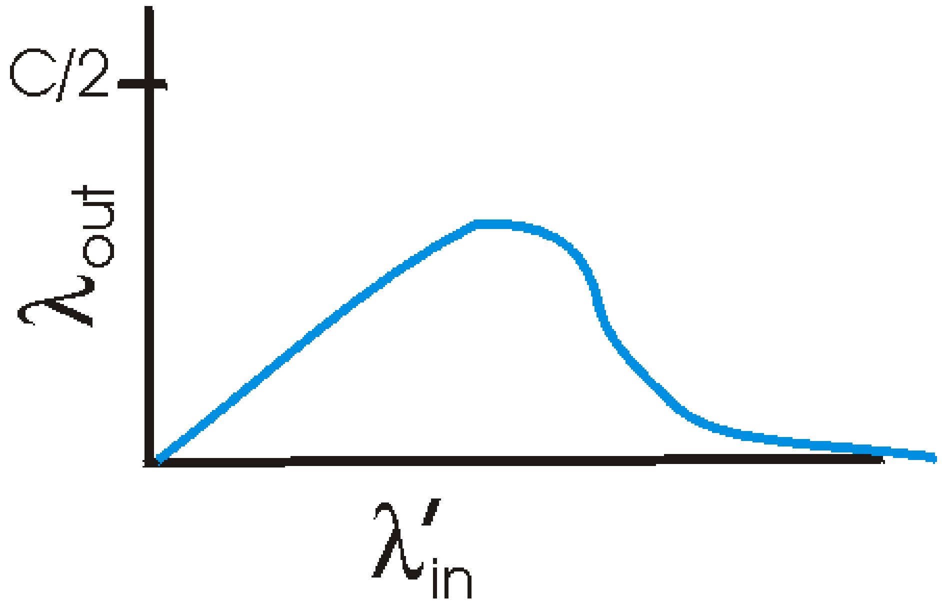
c) retransmission too eager (timeout to small)

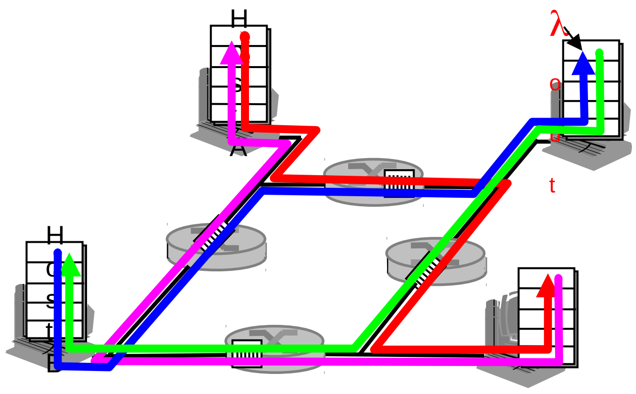


Circular Bottlenecks



Congestion Collapse





- packet drop after upstream bottleneck

  => upstream capacity wasted

Congestion Control

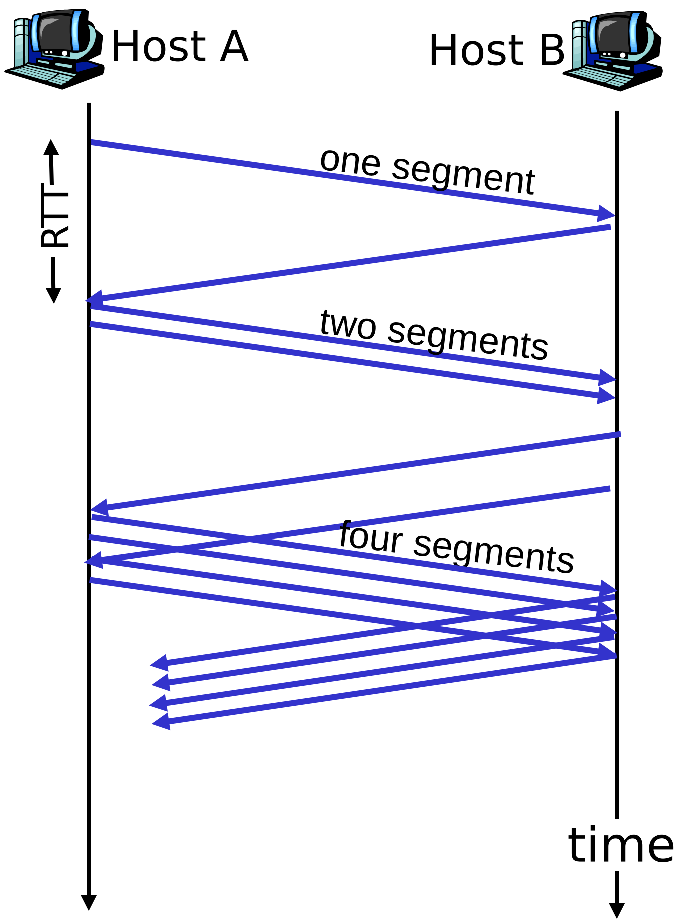
- senders must control rate to avoid permanent network overload
- input signals?
  - direct feedback from network? overhead!
  - indirect feedback through receiver? reaction time!
    - TCP congestion control
    - use drop or packet marking to indicate overload

TCP Congestion Control

- limit depth of pipeline – *congestion window*
- sending rate (roughly): CongWin / RTT
- adjust CongWin based on network feedback
  - sender infers packet loss
    - duplicate ACK -> assume light overload
    - timeout -> assume severe overload
- adaptation regimes/phases
  - *slow start*: start at very small rate, increase fast
  - *congestion avoidance*: hold rate, increase slow

TCP Slow Start

- start with small fixed CongWin
- increase exponentially until first loss
  - double CongWin every RTT, i.e.:
  - increment CongWin for every ACK
- start slow, but increase fast

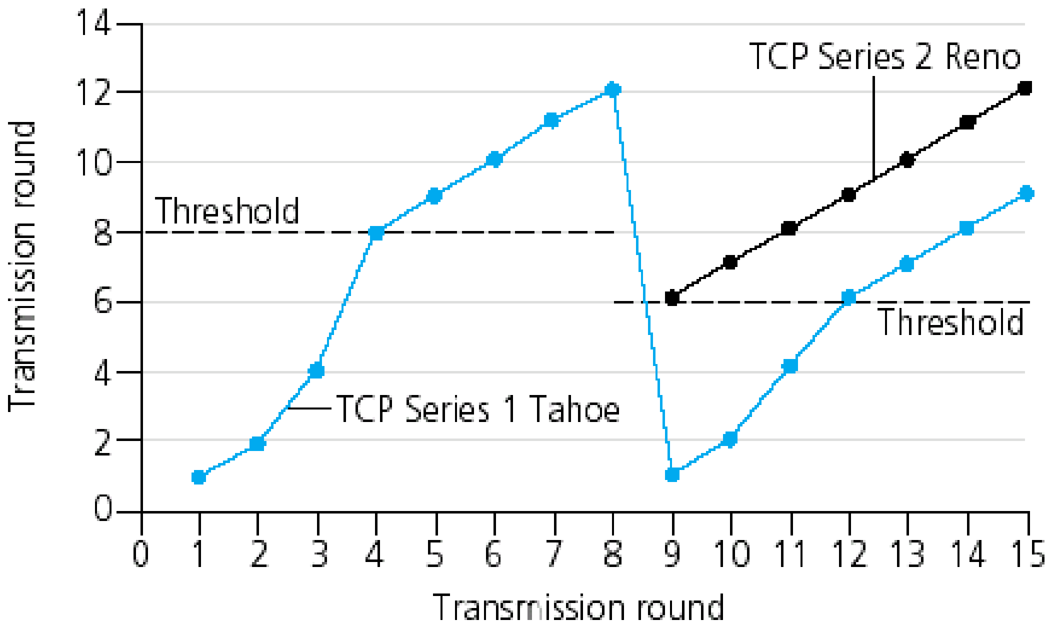


TCP Congestion Avoidance

- regular operation (no loss):
  increase CongWin by fixed amount per RTT
- receiver detects missing segment
  -> send duplicate ACK for previous one
- sender receives 3 duplicate ACKs
  -> reduce CongWin in half
- but after sender timeout:
  -> restart Slow Start procedure

TCP Rate Control

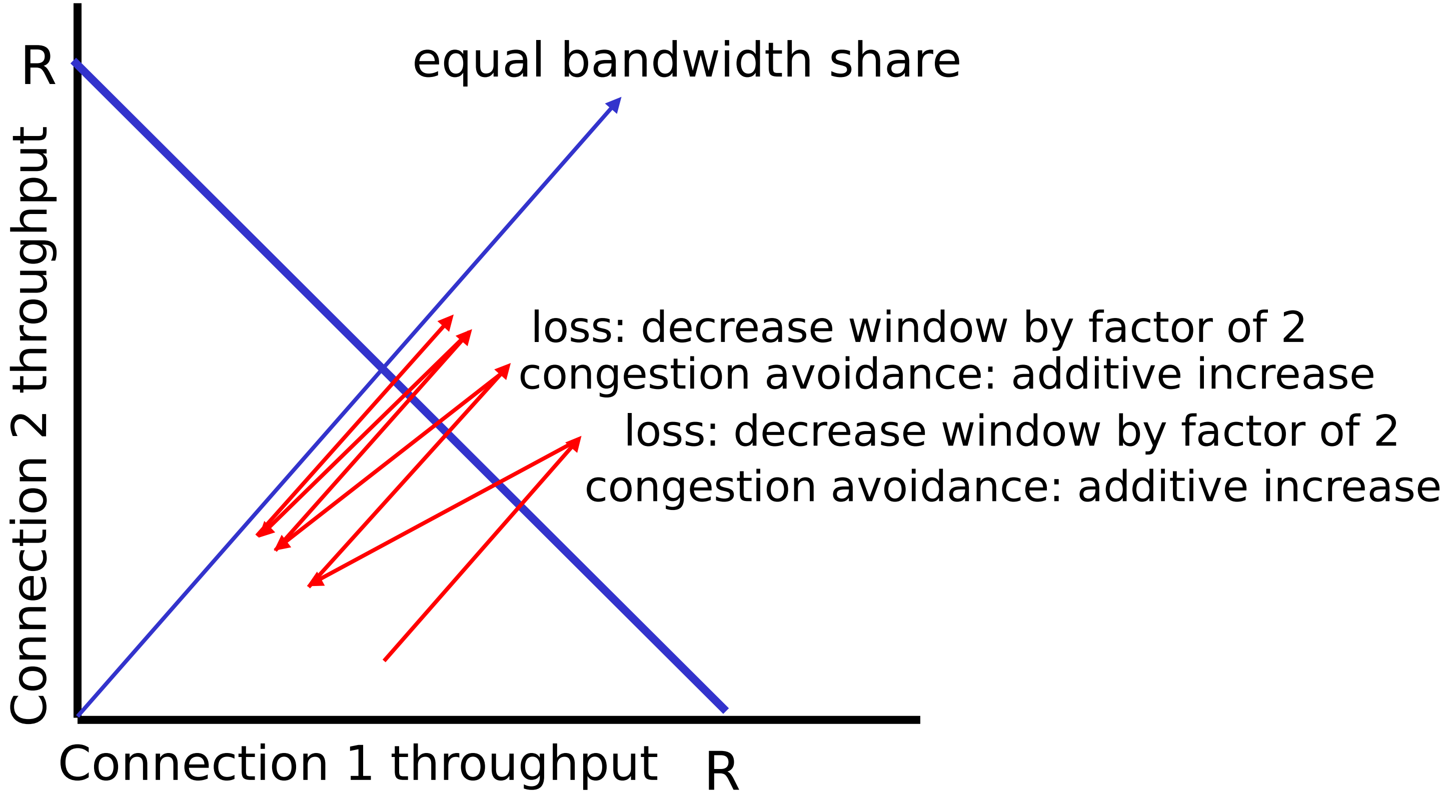
- Slow Start -> Congestion Avoidance
  - based on threshold (CongWin/2 of last timeout)



TCP Fairness

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



TCP Discussion

- hybrid of Go-Back-N and Selective Repeat
  - SACK: more precise acknowledgements (limited)
- reduction of CongWin -> pause sending
  - until ACKs catch up with outstanding data
- refinements
  - fast retransmit & fast recovery -> resume sending faster during dupack losses
  - keep sending at ACK-clocked pace

TCP Discussion

- TCP fairness relies on configuration values
  - initial window for slow start
  - additive increase during congestion avoidance

-> problem with highspeed / long-delay links

- more agile congestion control -> robustness?
- per-session fairness?

- lots of other approaches in the literature
  - very little real-world adoption

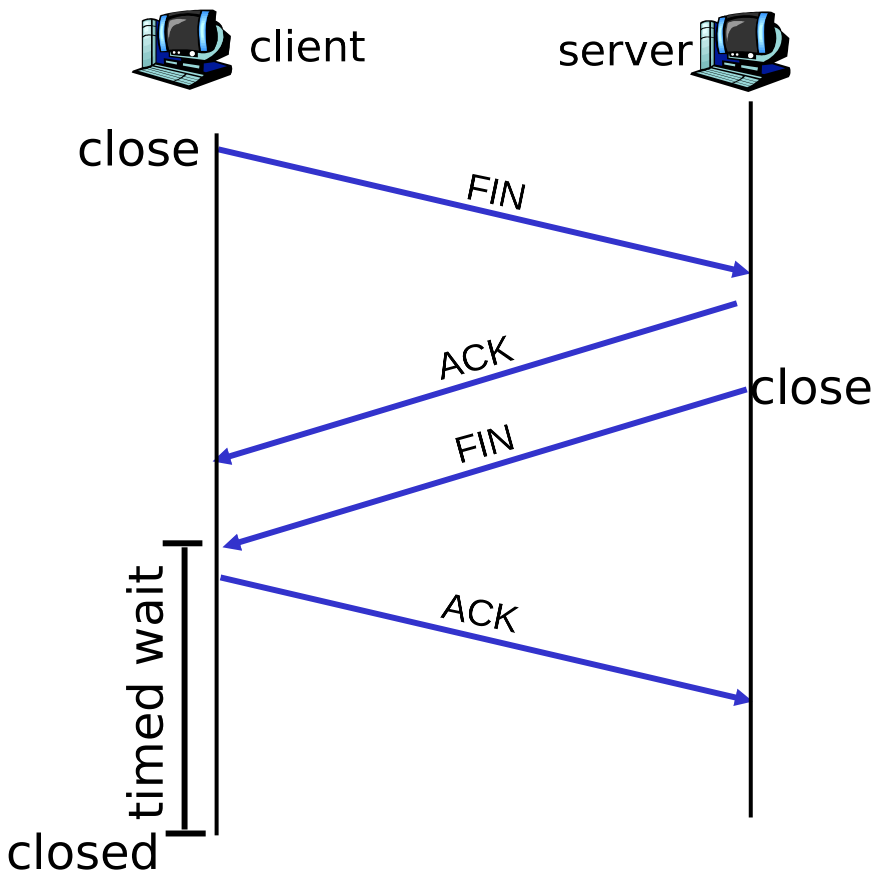
Connection Management: TCP

- Connection Establishment: 3-Way Handshake
- Step 1: initiator sends SYN to responder
  - sets up initial variables, e.g., sequence number
- Step 2: responder responds with SYNACK
  - sets up initial variables, e.g., sequence number
  - responder allocates internal buffer
- Step 3: initiator responds with ACK
  - might already send data along

Connection Management: TCP

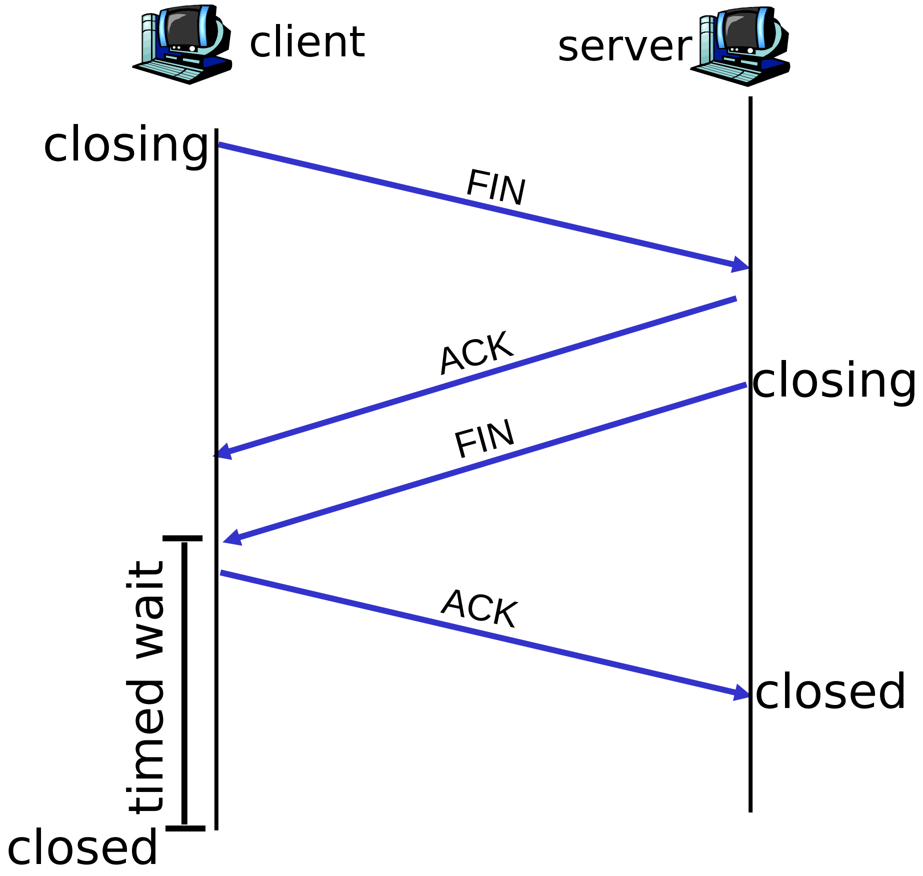
- Connection Teardown – Be Aware of Reliability!
- Step 1: client sends TCP FIN to server
- Step 2: server responds with ACK & sends FIN

- FIN -> no more data
- ACK -> all data received



Connection Management: TCP

- Step 3: client responds with ACK
  - enters “TIMED WAIT”
  - in case ACK is lost
- Step 4: server receives ACK
  - connection closed



Interface Semantics

- message interface
  - message boundaries preserved across transport
- byte-stream interface
  - message boundaries not preserved
  - simpler and more flexible for implementation