Transport Layer

EECS 489 Computer Networks

http://www.eecs.umich.edu/courses/eecs489/w07

Z. Morley Mao

Wednesday Jan 24, 2007

Acknowledgement: Some slides taken from Kurose&Ross and Katz&Stoica

Adminstrivia

- Homework 1 was due yesterday – 1/23
  - You can use your late days
- PA1 has been posted
  - A simplified Web server
  - You need to find a project partner for this assignment
  - If you want to work in groups of three or by yourself, please email us to get permission.
- Reading assignment for this week
  - Chapter 3 of the book
  - You should have read Chapter 1 and 2.

Discussion on email

- Why do we have so much spam?
  - How would you design the email system to prevent spam?
- How does anonymous email work?

Discussion on DNS

- Is it easy to attack the DNS system?
- Why is DNS caching good?
- Why is DNS caching bad?
- DNS is “exploited” for server load balancing, how?
  - Local DNS servers are usually close to local clients
- If you were to design DNS differently today, how would you?
  - Any problems with the current DNS system?

Discussion on P2P

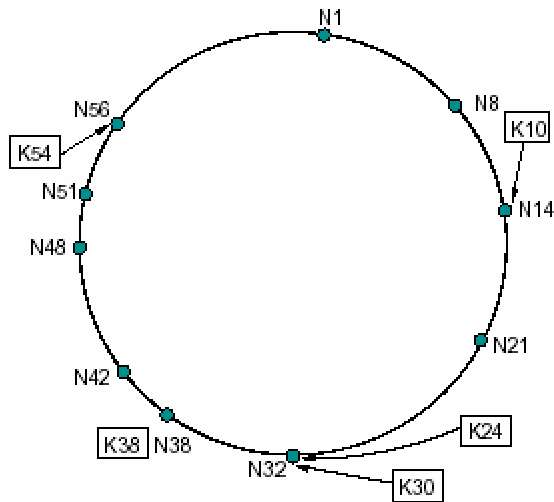
- How would you design a P2P system that is scalable, decentralized, and guarantees the location of the files?
- One solution: DHT (Distributed Hash Table)
  - Guarantees that you can find the file
  - Mapping between the file and the node ID
  - Consistent hashing function assigns each node and key an m-bit identifier using SHA-1 base hash function.

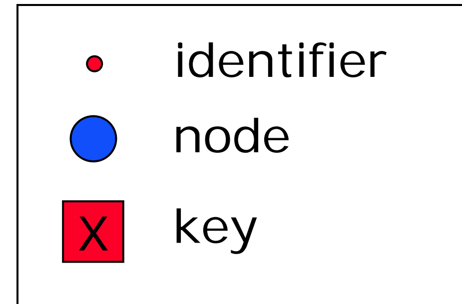
Chord protocol

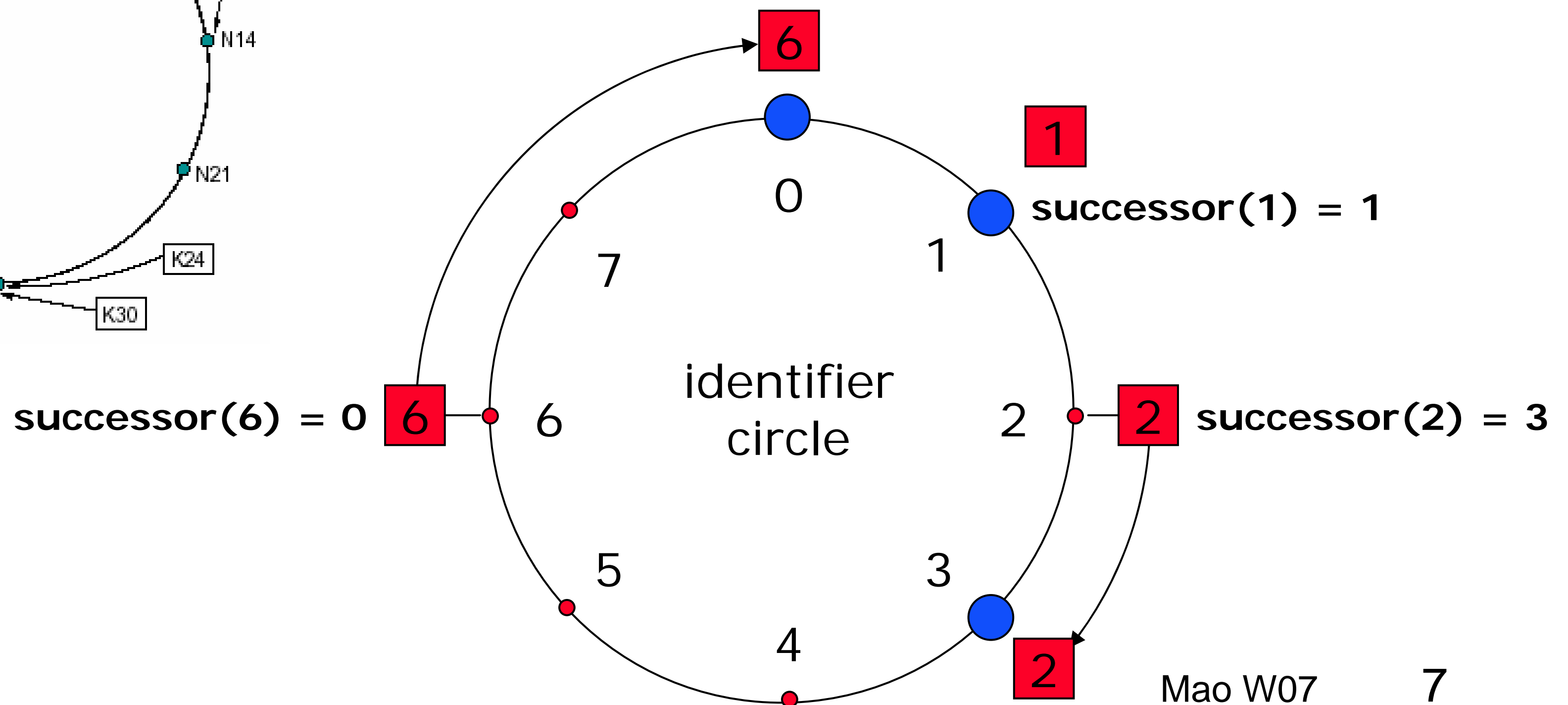
- Consistent hashing function assigns each node and key an m-bit identifier using SHA-1 base hash function.
- Node's IP address is hashed.
- Identifiers are ordered on a identifier circle modulo 2^m called a chord ring.
- *succesor(k)* = first node whose identifier is >= identifier of k in identifier space.

Chord protocol

For m = 6, # of identifiers is 64.
The following Chord ring has 10 nodes and stores 5 keys.
The successor of key 10 is node 14.

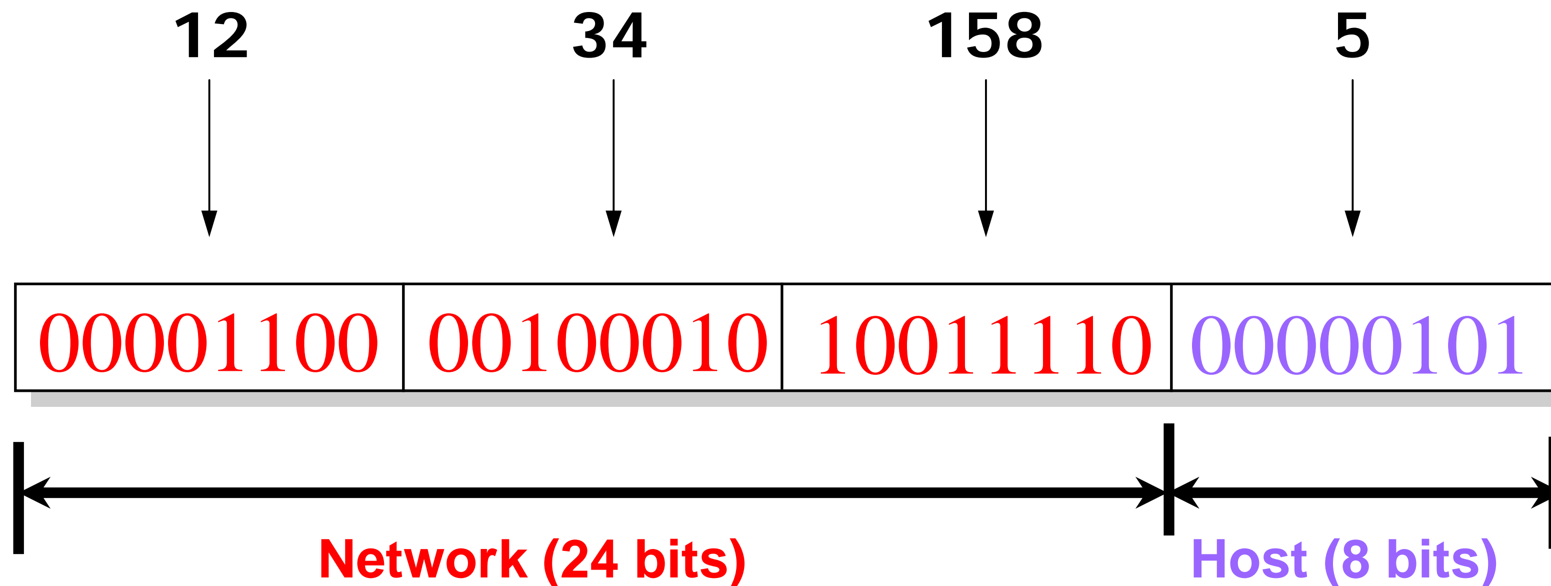






IP Addressing

- 32-bit number in dotted-quad notation (12.34.158.5)
- Divided into network & host portions (left and right)
- 12.34.158.0/24 is a 24-bit prefix with 2^8 addresses



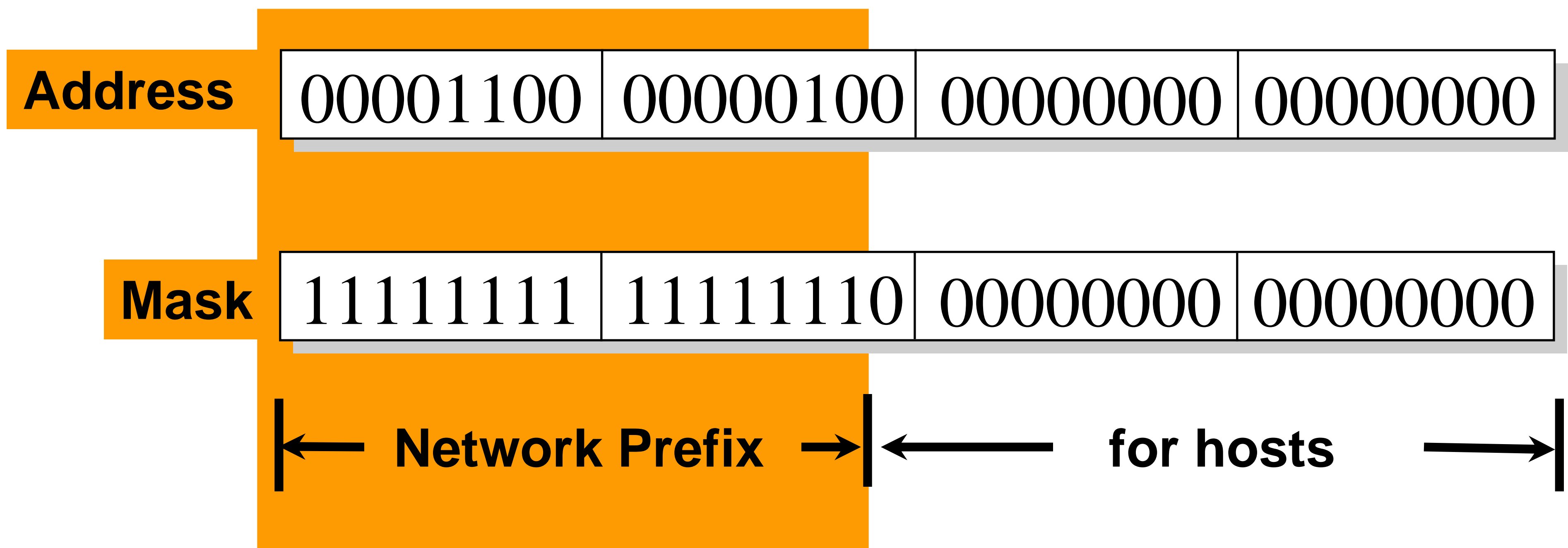
Some History: Why Dotted-Quad Notation?

- In the olden days…
  - Class A: 0*
    - Very large /8 blocks (e.g., MIT has 18.0.0.0/8)
  - Class B: 10*
    - Large /16 blocks (e.g,. UM has 141.213.0.0/16)
  - Class C: 110*
    - Small /24 blocks (e.g., AT&T Labs has 192.20.225.0/24)
  - Class D: 1110*
    - Multicast groups
  - Class E: 11110*
    - Reserved for future use (sounds a bit scary…)
- And then, address space became scarce…

Classless Inter-Domain Routing (CIDR)

Use two 32-bit numbers to represent a network.
Network number = IP address + Mask

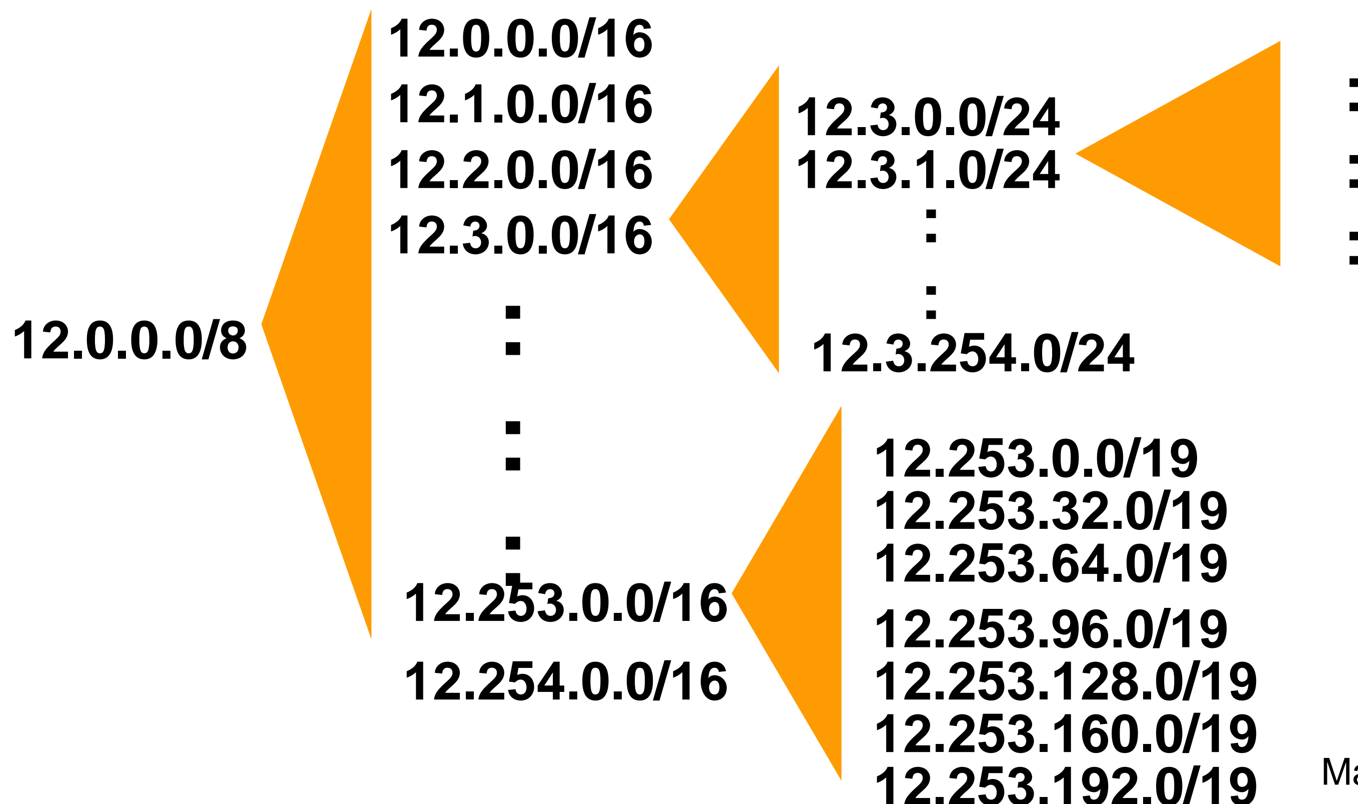
IP Address : 12.4.0.0 IP Mask: 255.254.0.0



Usually written as 12.4.0.0/15

CIDR = Hierarchy in Address Allocation

- Prefixes are key to Internet scalability
  - Address allocation by ARIN/RIPE/APNIC and by ISPs
  - Routing protocols and packet forwarding based on prefixes
  - Today, routing tables contain ~150,000-200,000 prefixes



Figuring Out Who Owns an Address

- Address registries
  - Public record of address allocations
  - ISPs should update when giving addresses to customers
  - However, records are notoriously out-of-date
- Ways to query
  - UNIX: “whois –h whois.arin.net 128.112.136.35”
  - http://www.arin.net/whois/
  - http://www.geektools.com/whois.php
  - …

Example Output for 141.213.4.5 (galileo.eecs.umich.edu)

OrgName: University of Michigan
OrgID: UNIVER-118
Address: IT Communications Services
Address: 4251 Plymouth Road
City: Ann Arbor
StateProv: MI
PostalCode: 48105-2785
Country: US

NetRange: 141.213.0.0 - 141.213.255.255
CIDR: 141.213.0.0/16
NetName: UMNET3
NetHandle: NET-141-213-0-0-1
Parent: NET-141-0-0-0-0
NetType: Direct Assignment
NameServer: SRVR8.ENGIN.UMICH.EDU
NameServer: SRVR7.ENGIN.UMICH.EDU
NameServer: DNS2.ITD.UMICH.EDU

There is more…

Comment: Abuse contact for 141.213.128.0/17 is abuse@umich.edu.
Comment: For DMCA info see http://www.umich.edu/~itua/copyright/
RegDate: 1990-08-02
Updated: 2003-03-27

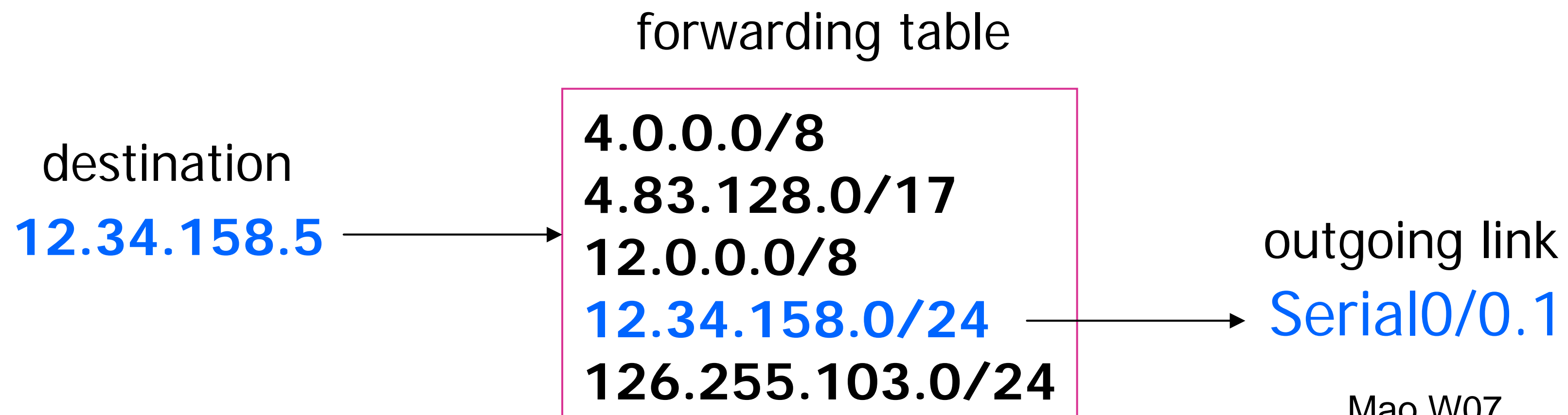
AbuseHandle: CEAC-ARIN
AbuseName: College of Engineering Abuse Contact
AbusePhone: +1-734-936-2486
AbuseEmail: abuse@engin.umich.edu

TechHandle: PMK5-ARIN
TechName: Killey, Paul M.
TechPhone: +1-734-763-4910
TechEmail: paul@engin.umich.edu

OrgTechHandle: UA11-ORG-ARIN
OrgTechName: UMnet Administration
OrgTechPhone: +1-734-647-4200
OrgTechEmail: umnet-admin@umich.edu

Longest Prefix Match Forwarding

- Forwarding tables in IP routers
  - Maps each IP prefix to next-hop link(s)
- Destination-based forwarding
  - Packet has a destination address
  - Router identifies longest-matching prefix
  - Cute algorithmic problem: very fast lookups



How are packets forwarded?

- Routers have forwarding tables
  - Map prefix to outgoing link(s)
- Entries can be statically configured
  - E.g., “map 12.34.158.0/24 to Serial0/0.1”
- But, this doesn’t adapt
  - To failures
  - To new equipment
  - To the need to balance load
  - …
- That is where routing protocols come in… [more on this in the next lectures]

Discussions

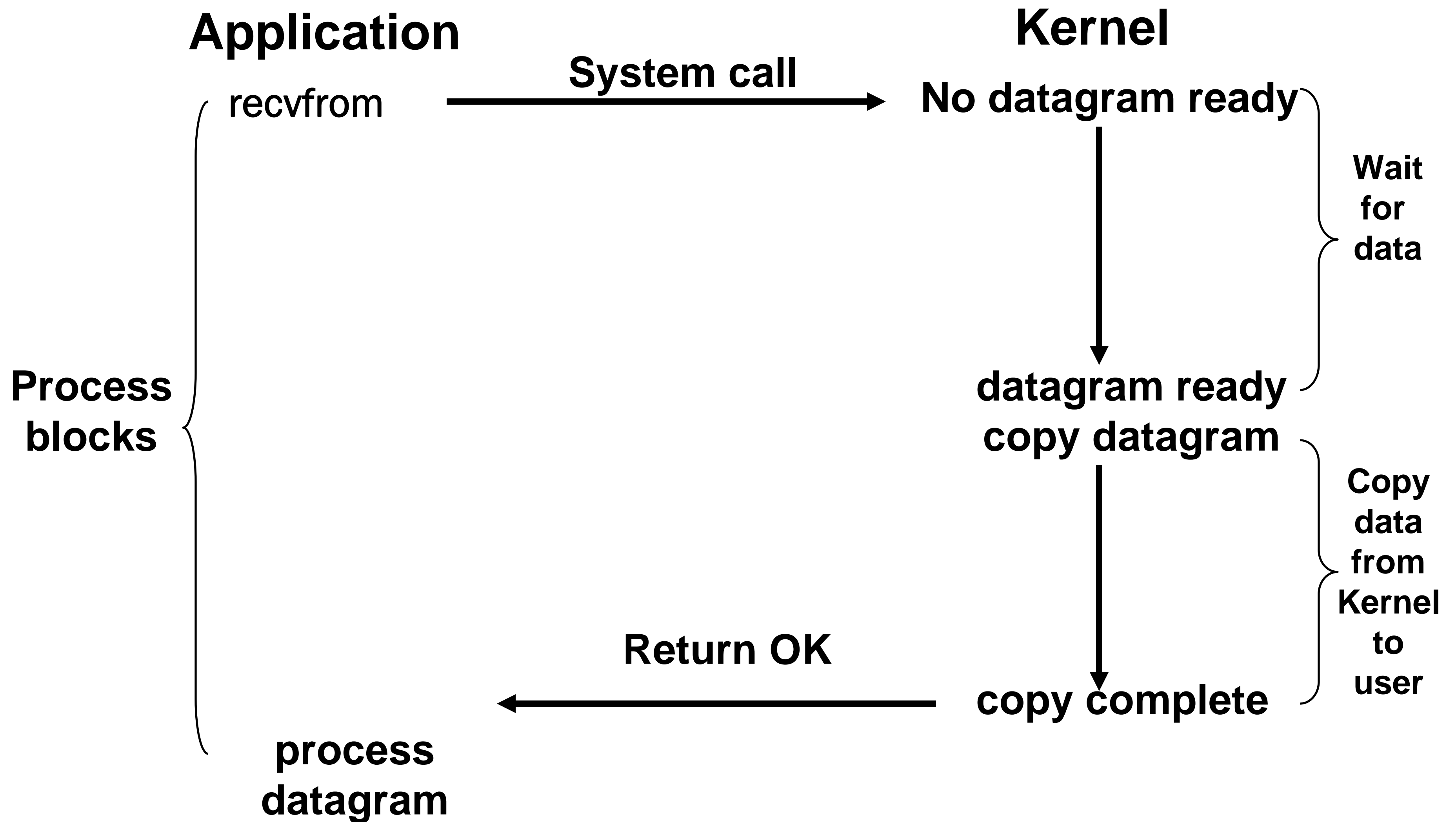
- IP address space scarcity
  - What can we do about it?
- Increased IP address fragmentation
- Does an IP address identify the actual user?
- How does one achieve mobility while maintaining the same IP address?

I/O Models

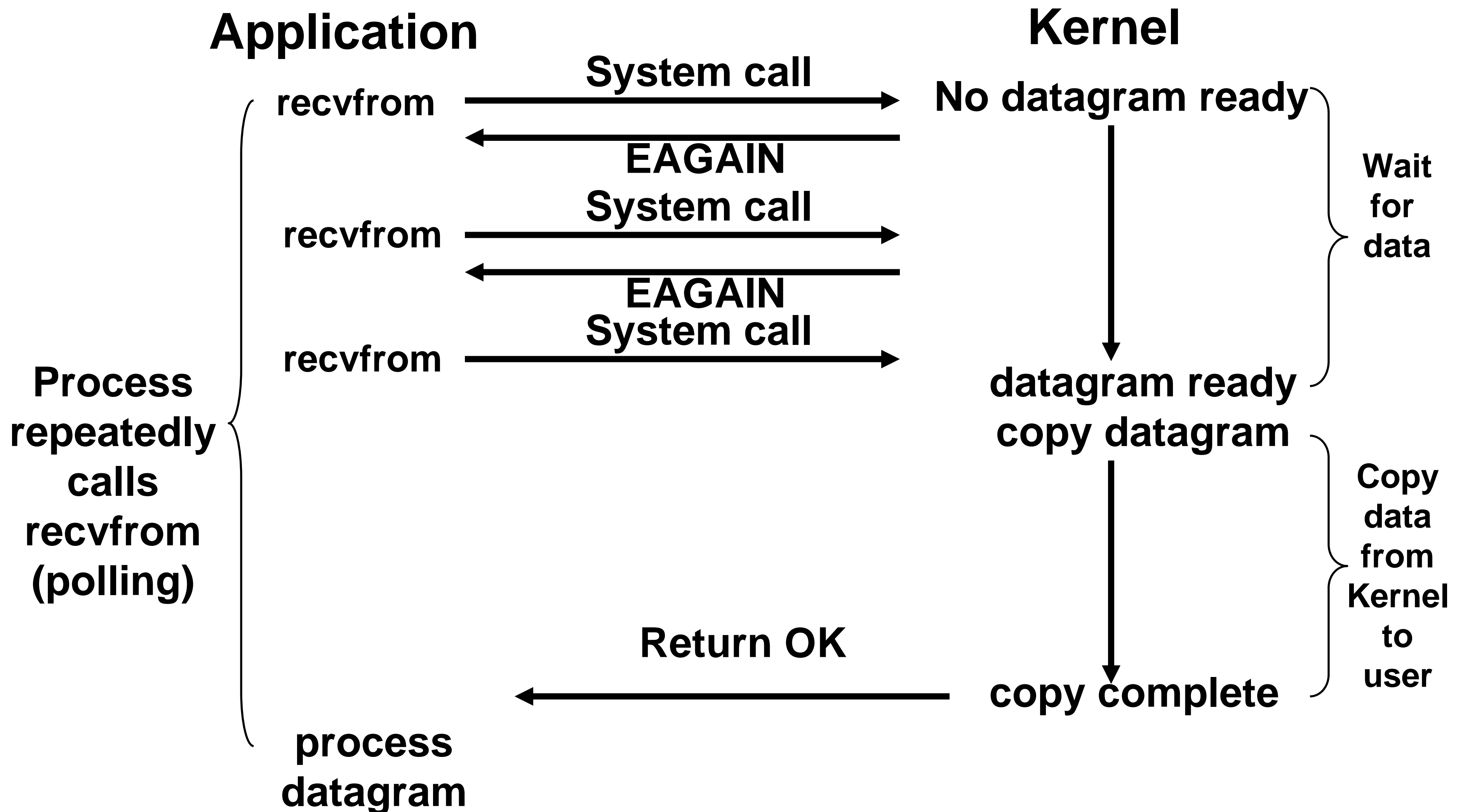
- Five different I/O models
  - Blocking I/O
  - Nonblocking I/O
  - I/O multiplexing (select and poll)
    - Threads with blocking I/O
  - Signal driven I/O (SIGIO)
  - Asynchronous I/O (aio_* functions)
    - Not widely supported

  (Blocking, Nonblocking, I/O multiplexing, Signal driven: **Synchronous**)
- Blocking vs. nonblocking (system call)
  - Whether the system call blocks until data is ready
- Synchronous vs. asynchronous (I/O operation)
  - Whether the I/O operation causes requesting process to be blocked

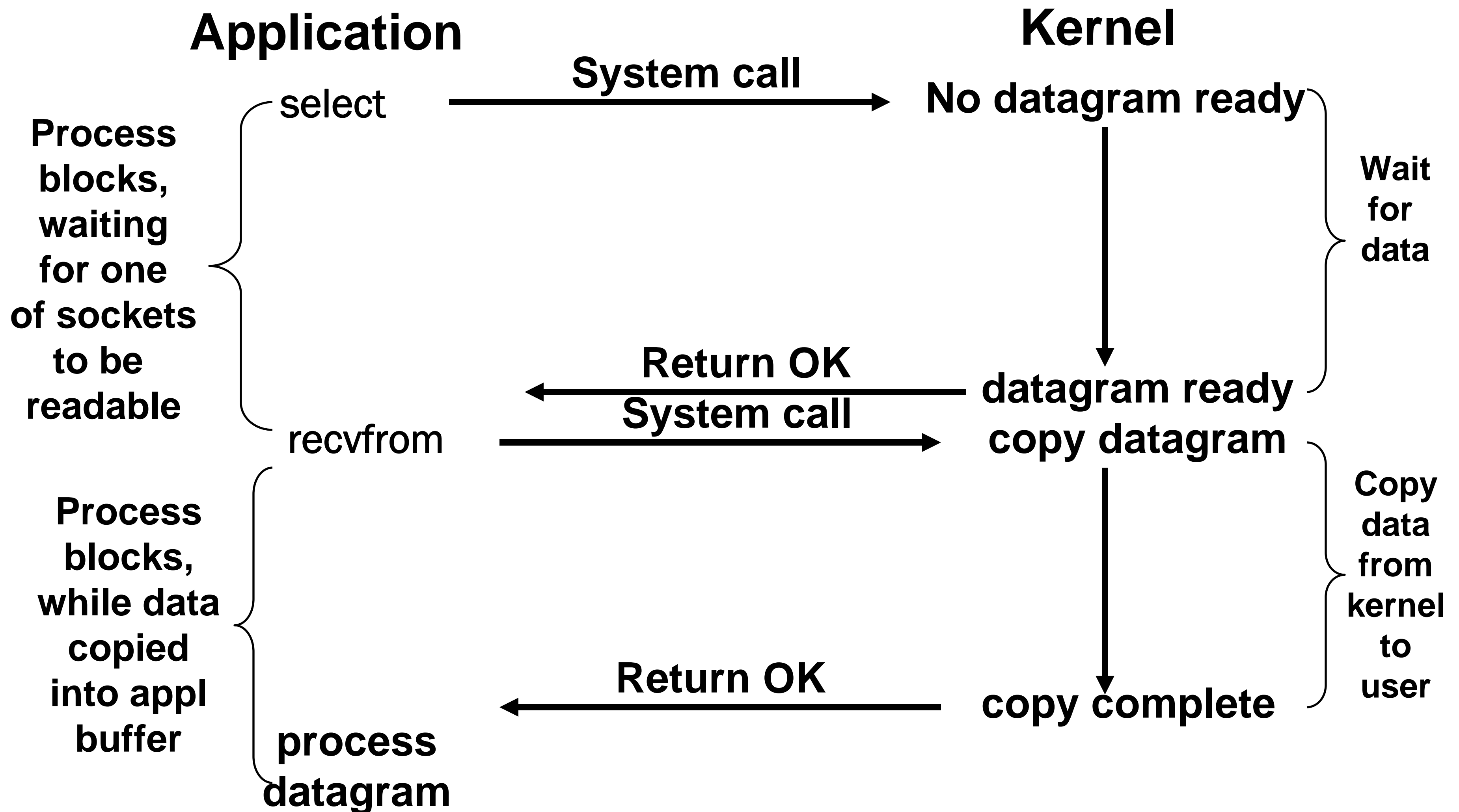
Blocking I/O Model



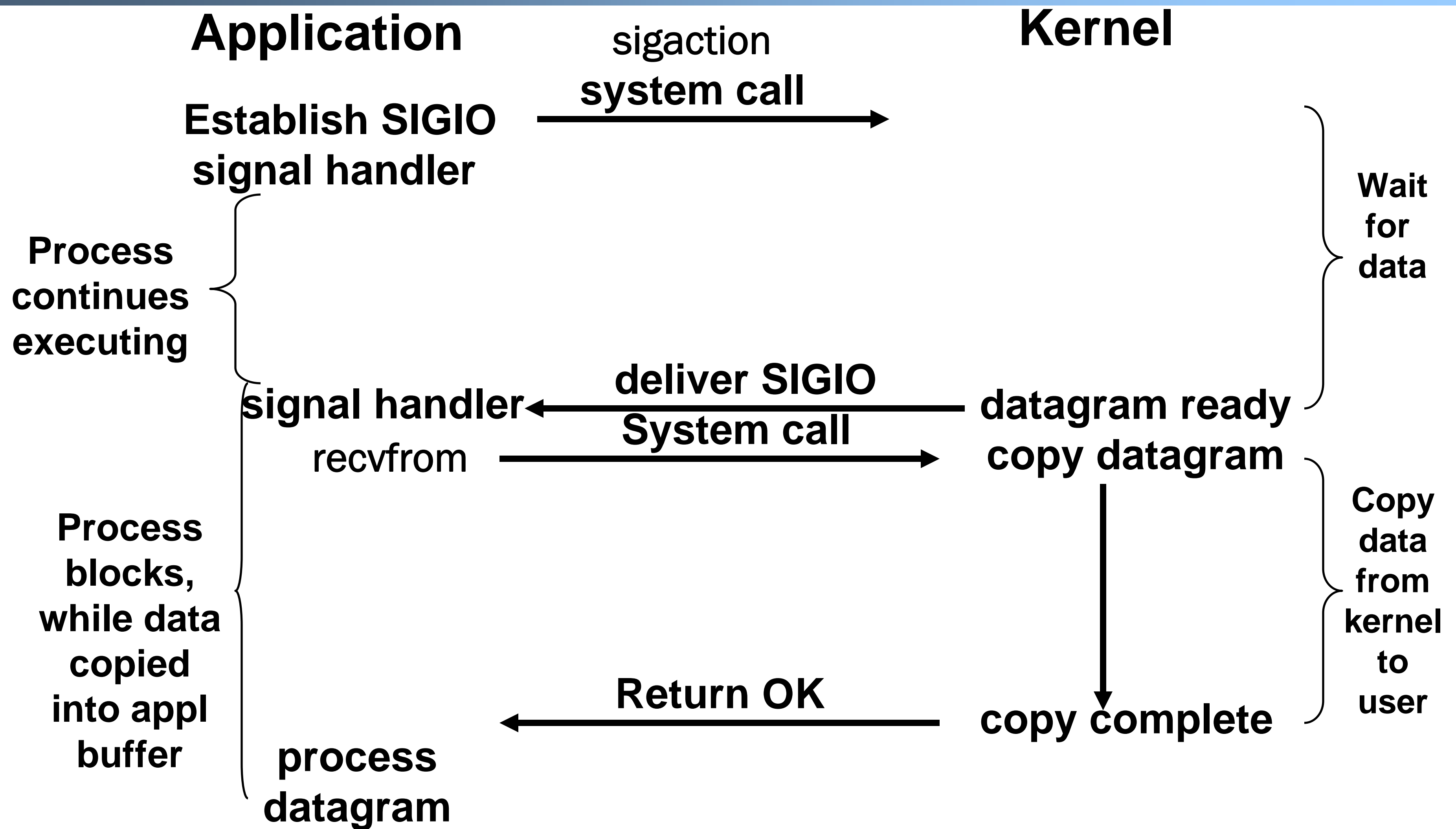
Nonblocking I/O Model



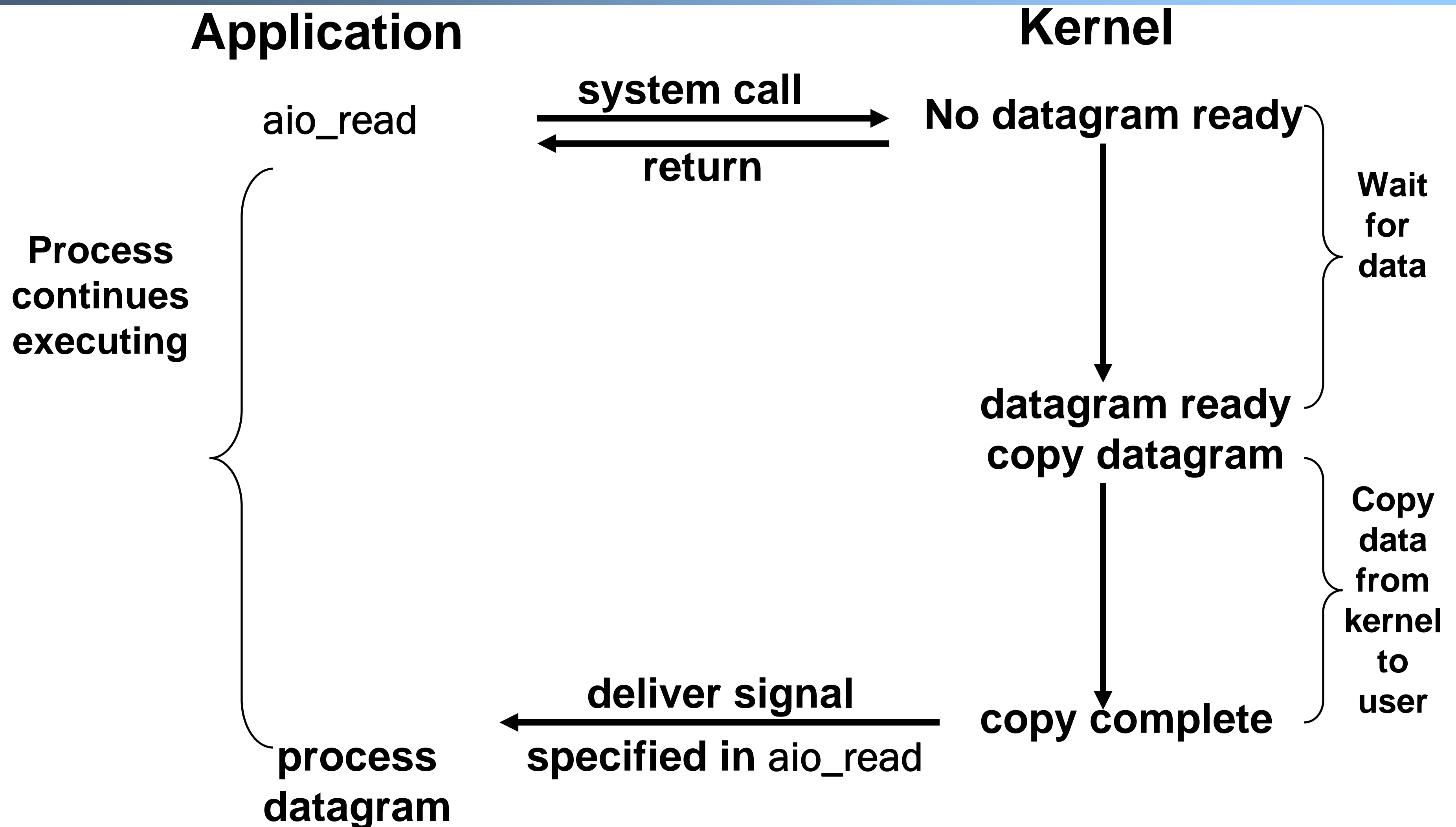
I/O Mutiplexing Model



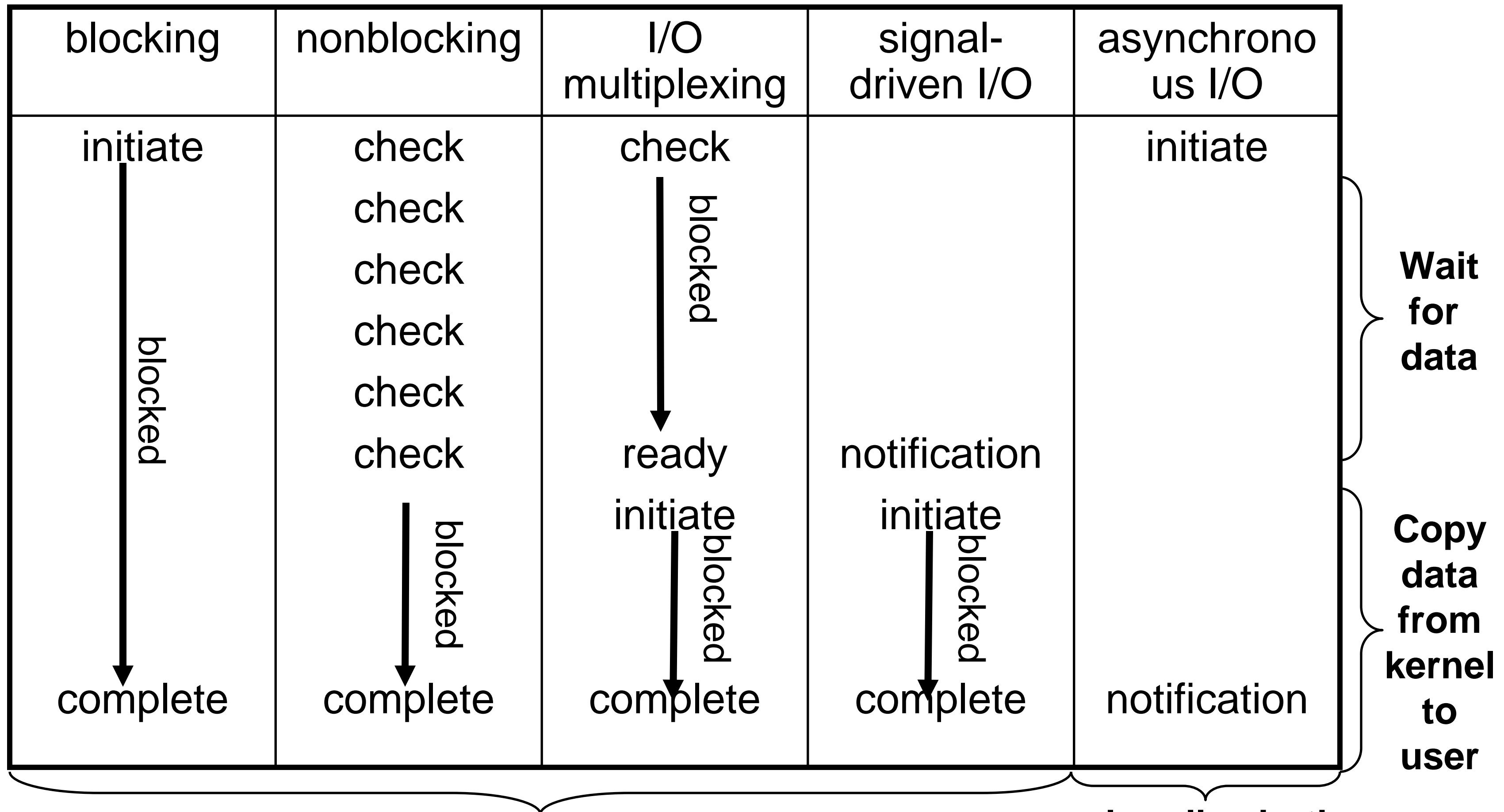
Signal-driven I/O Model



Asynchronous I/O Model

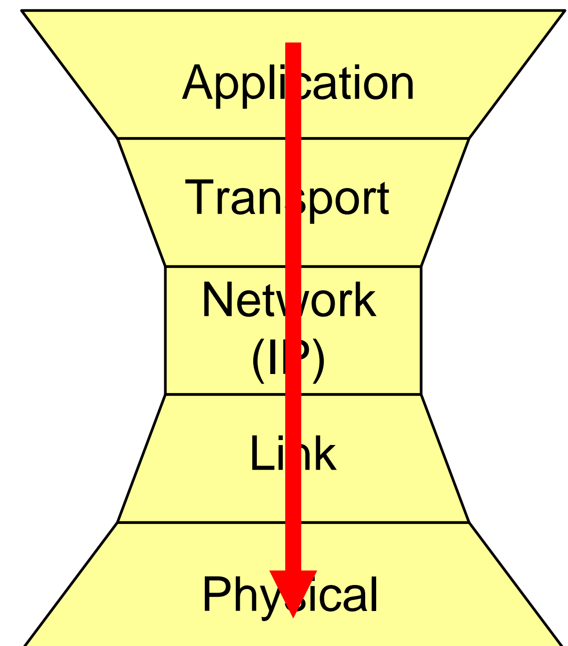


Comparison



Top-down

- New approach (E.g., Kurose & Ross) – start from the application layer all the way down to the physical layer
- Advantages – goals are very clear → start from application needs
- Disadvantages – harder to understand some assumptions made about lower layers (e.g., packet losses in the Internet are because of congestion)



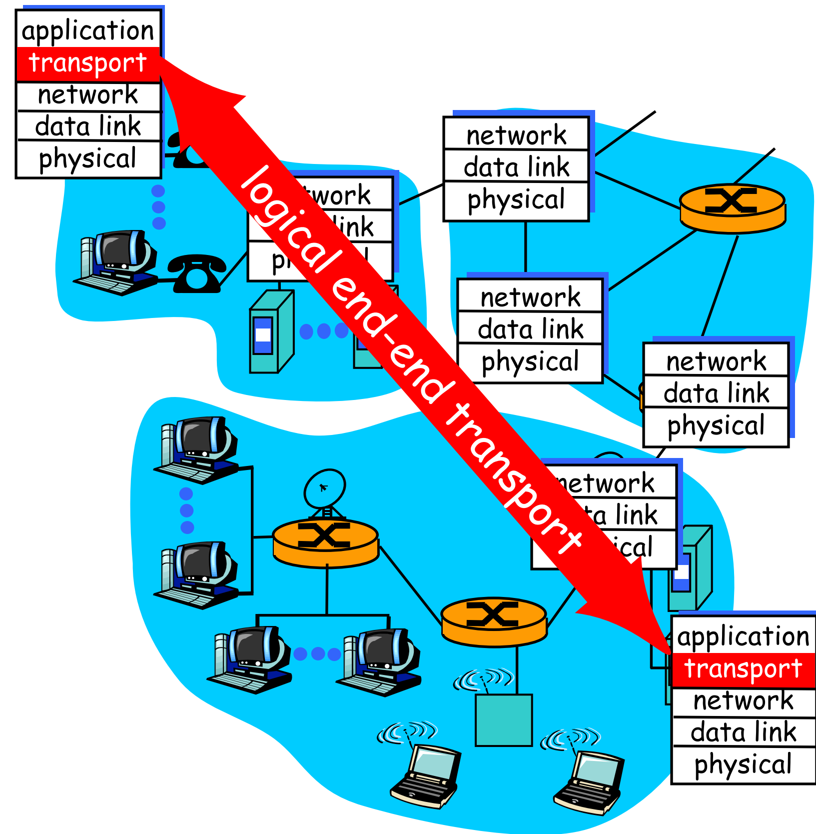
Transport Layer

Our goals:

- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services
- *Q: what is an example property that network layer does not have, transport layer provides? And vice versa?*

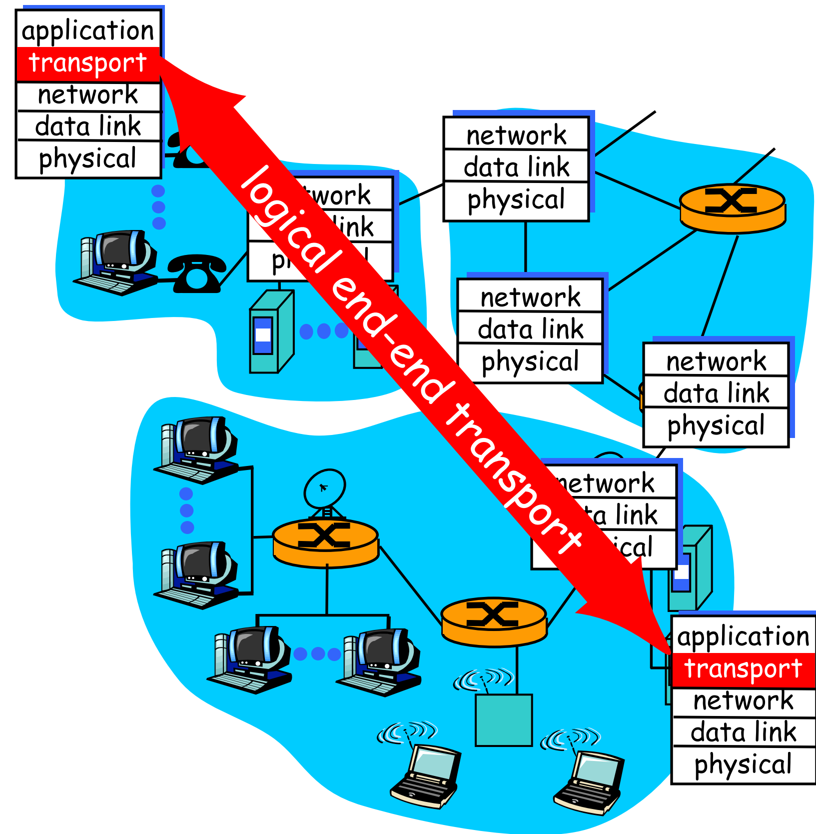
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



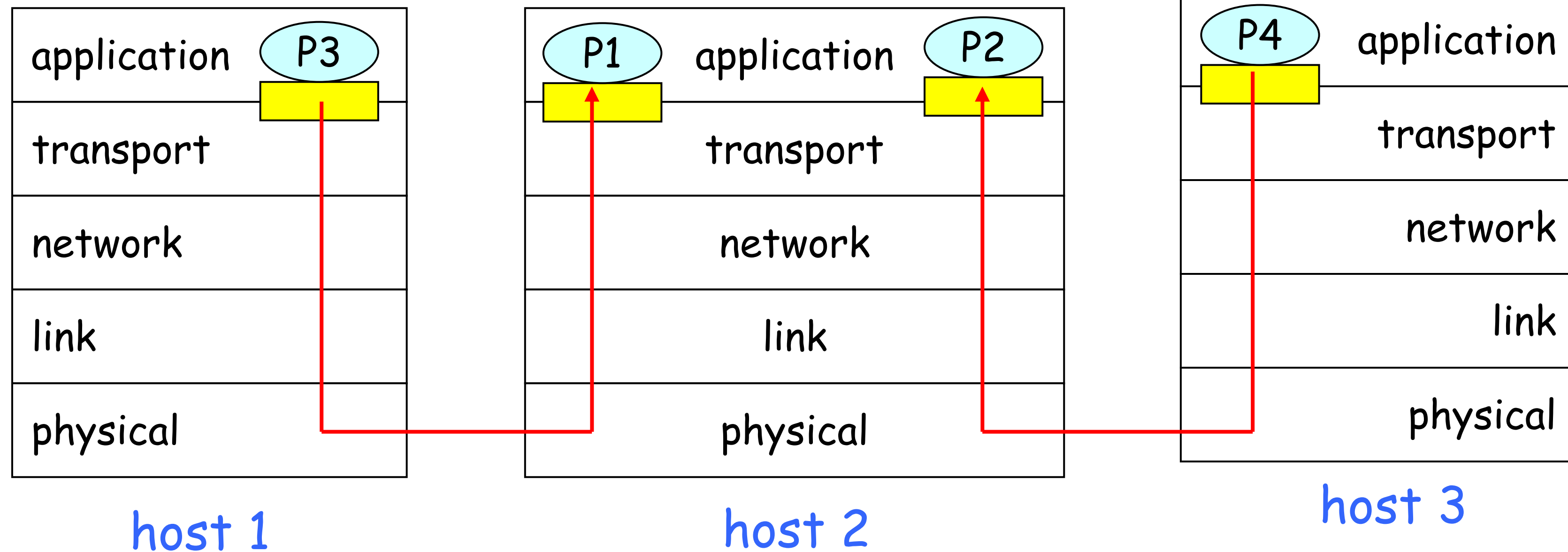
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

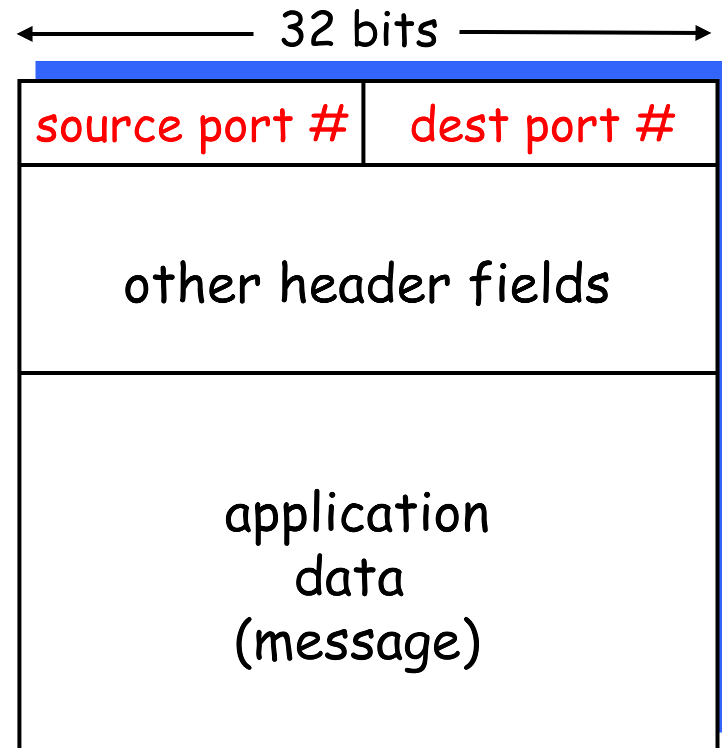
Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 =
  new DatagramSocket(99111);
DatagramSocket mySocket2 =
  new DatagramSocket(99222);
```

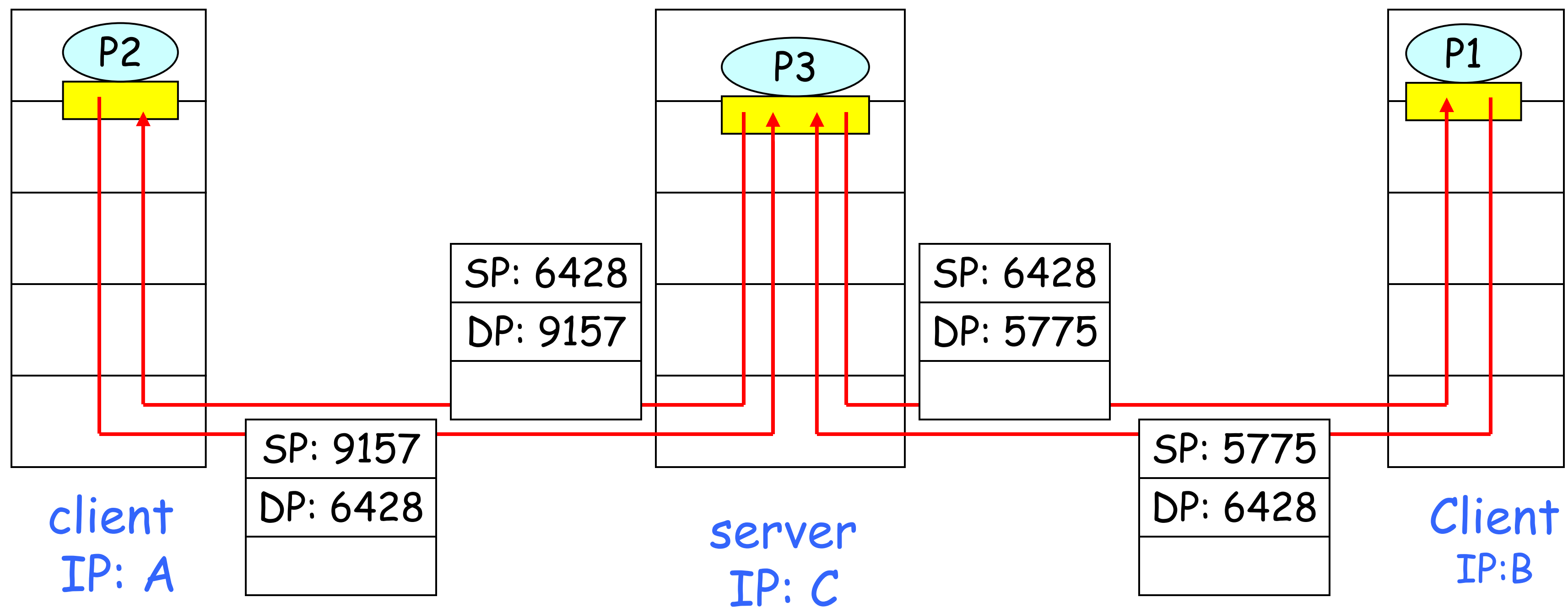
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

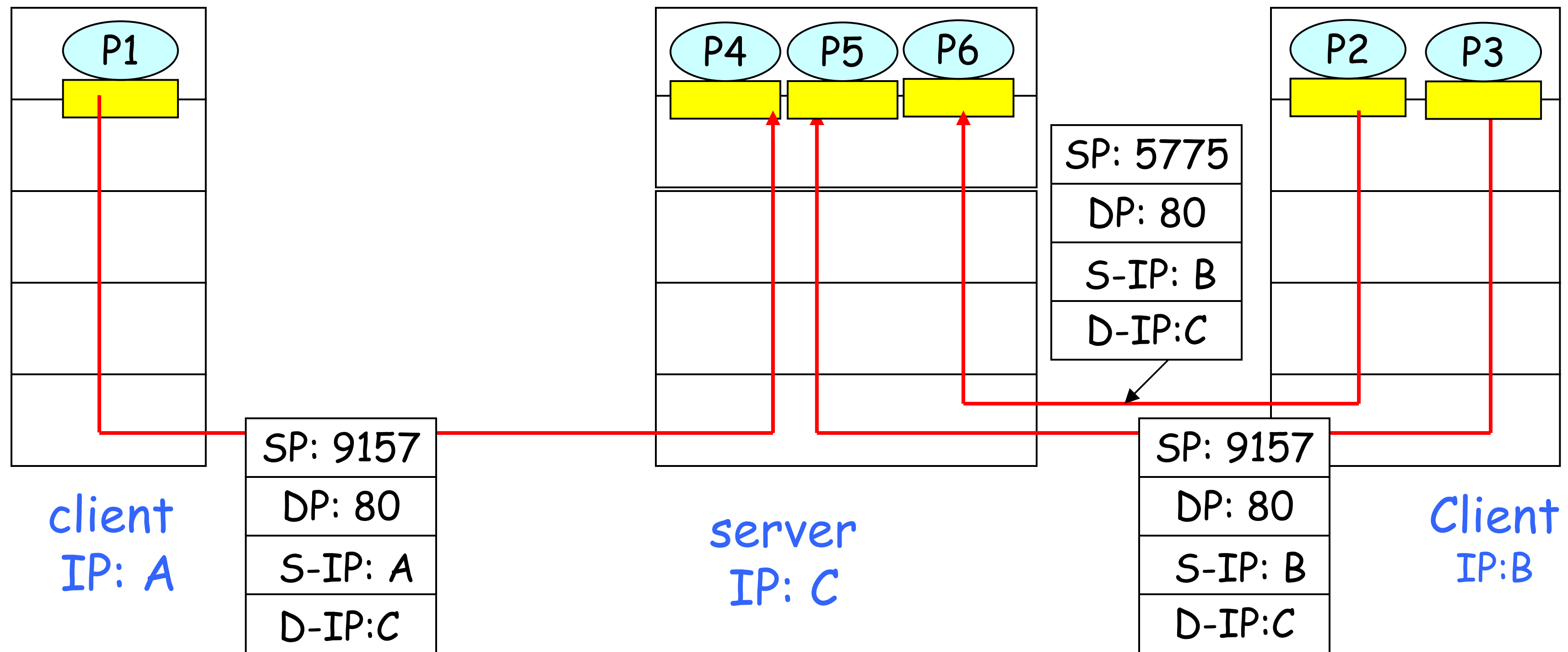


SP provides "return address"

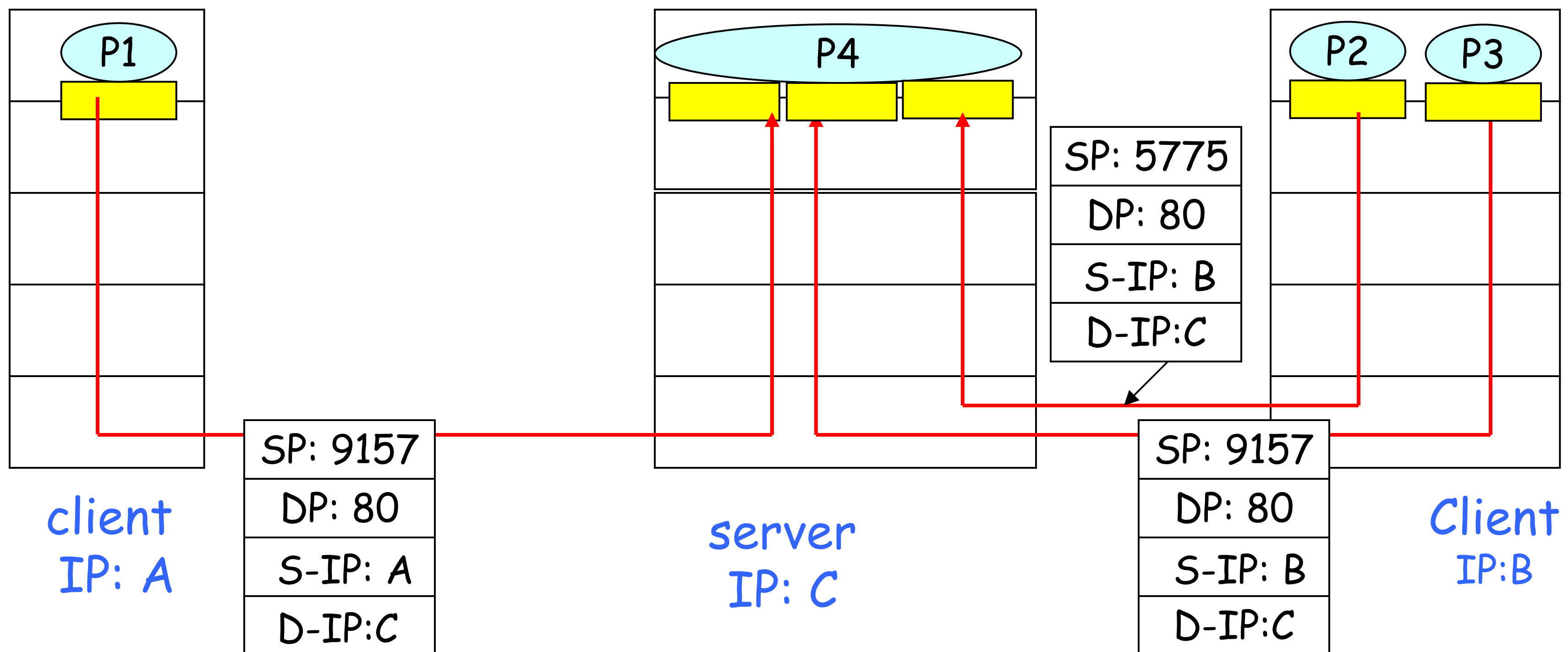
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



UDP: User Datagram Protocol [RFC 768]

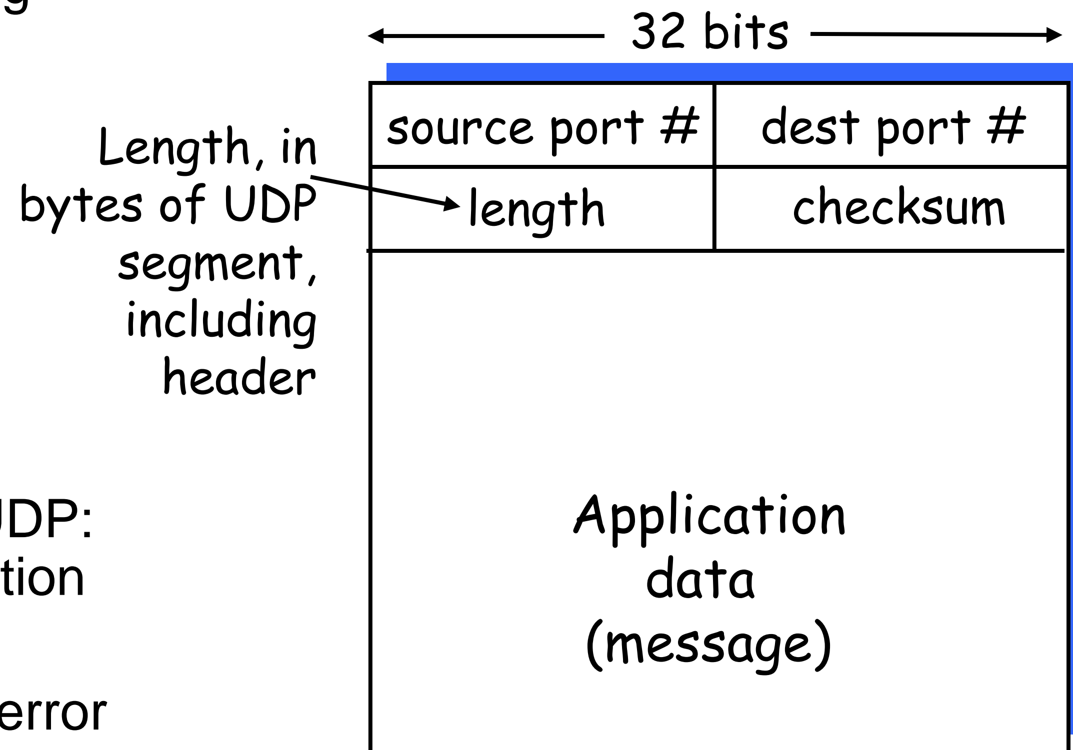
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP

- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- Other UDP uses
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format

UDP checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

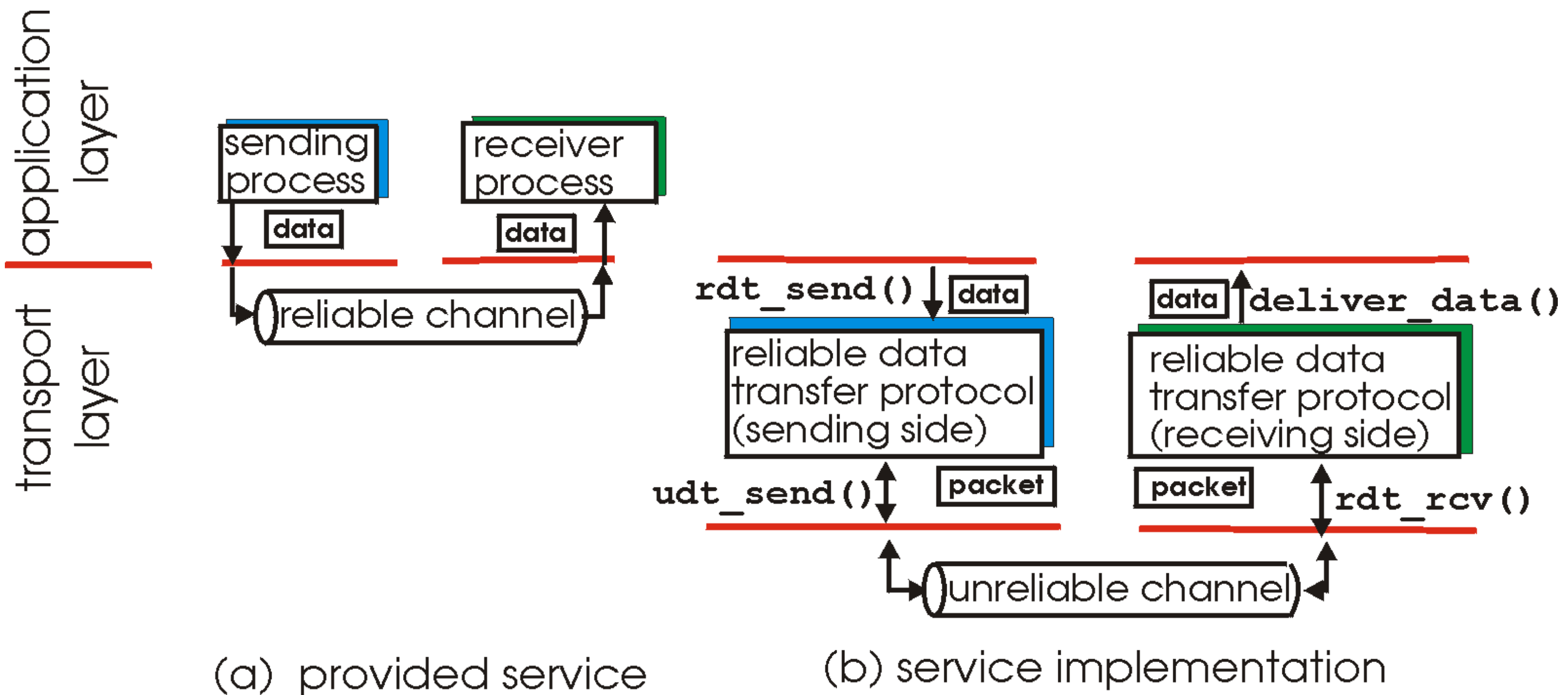
Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Principles of Reliable data transfer

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

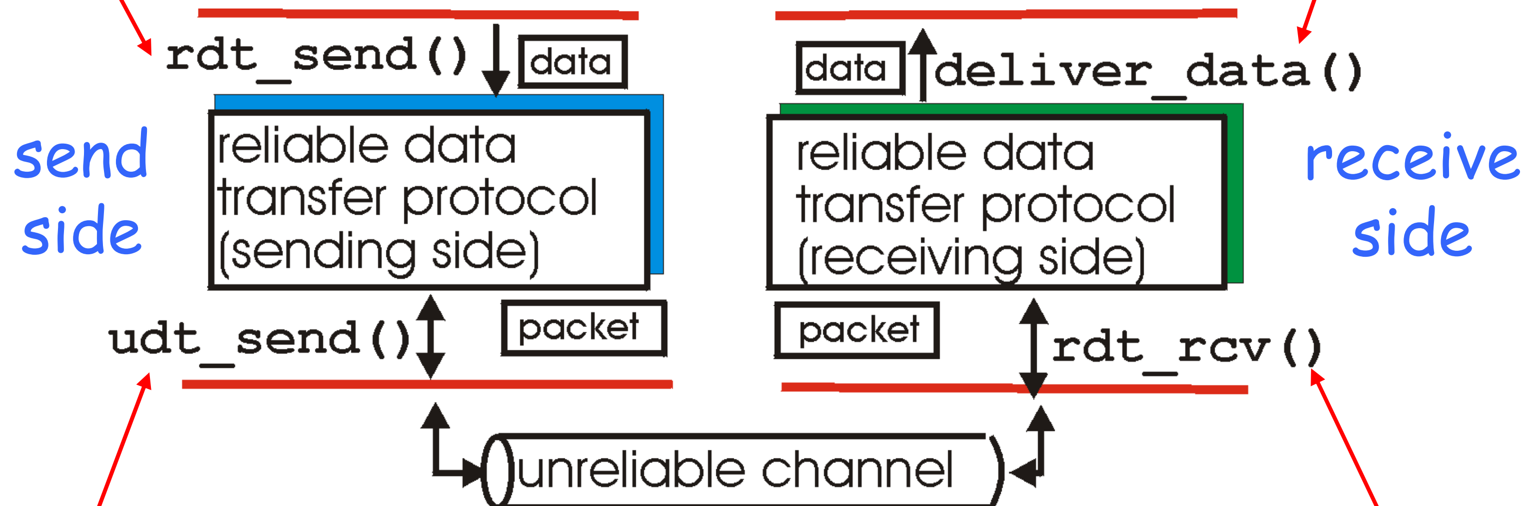


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



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

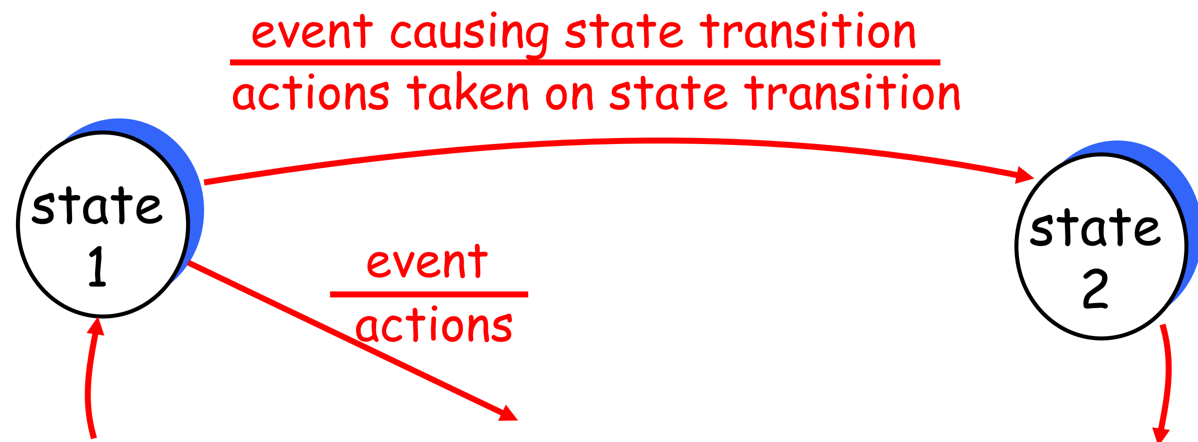
`rdt_rcv()`: called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

We'll:

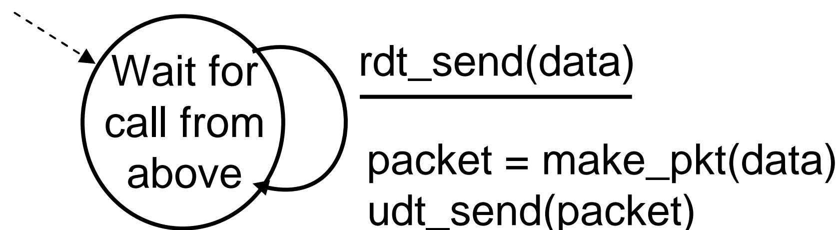
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

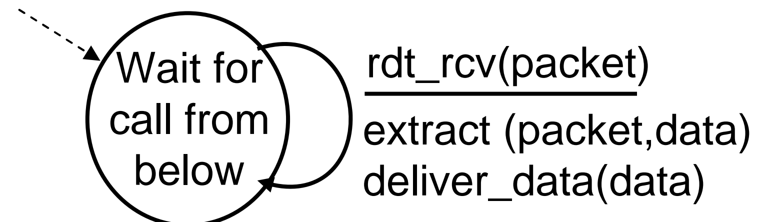
state: when in this "state" next state uniquely determined by next event



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 read data from underlying channel





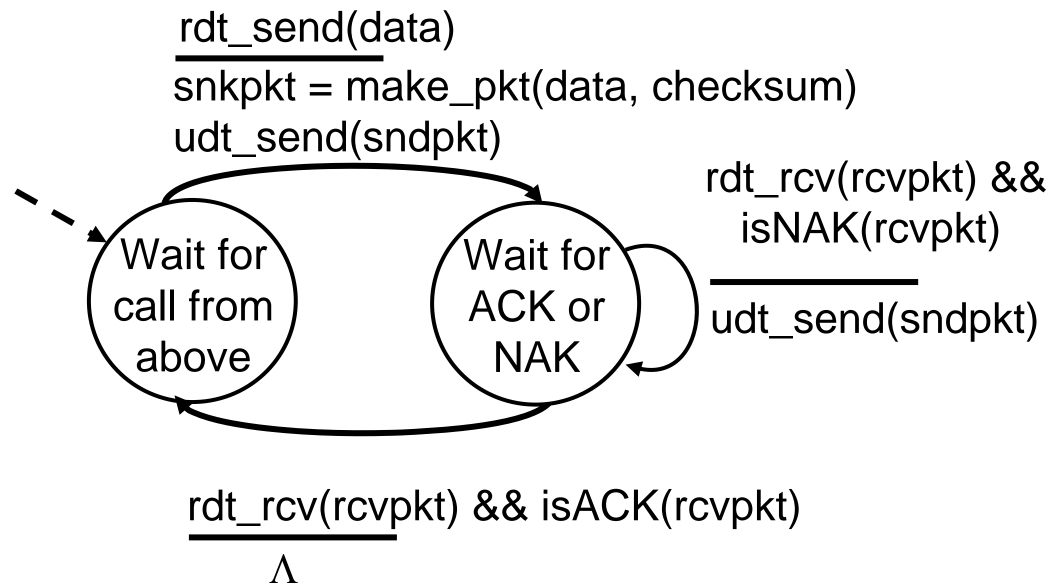
sender

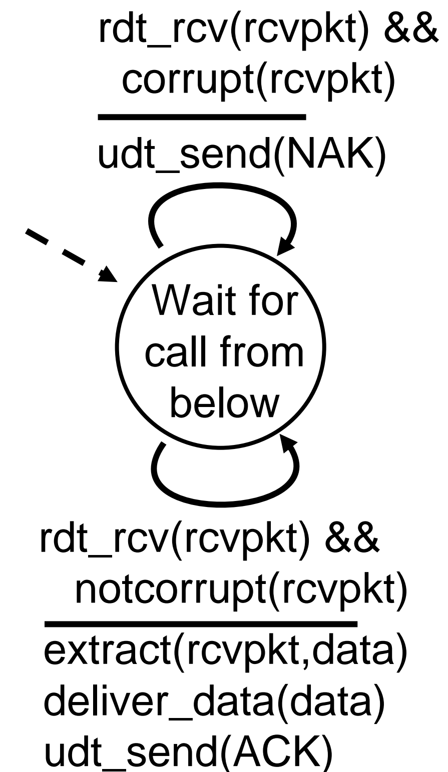
receiver

Rdt2.0: channel with bit errors

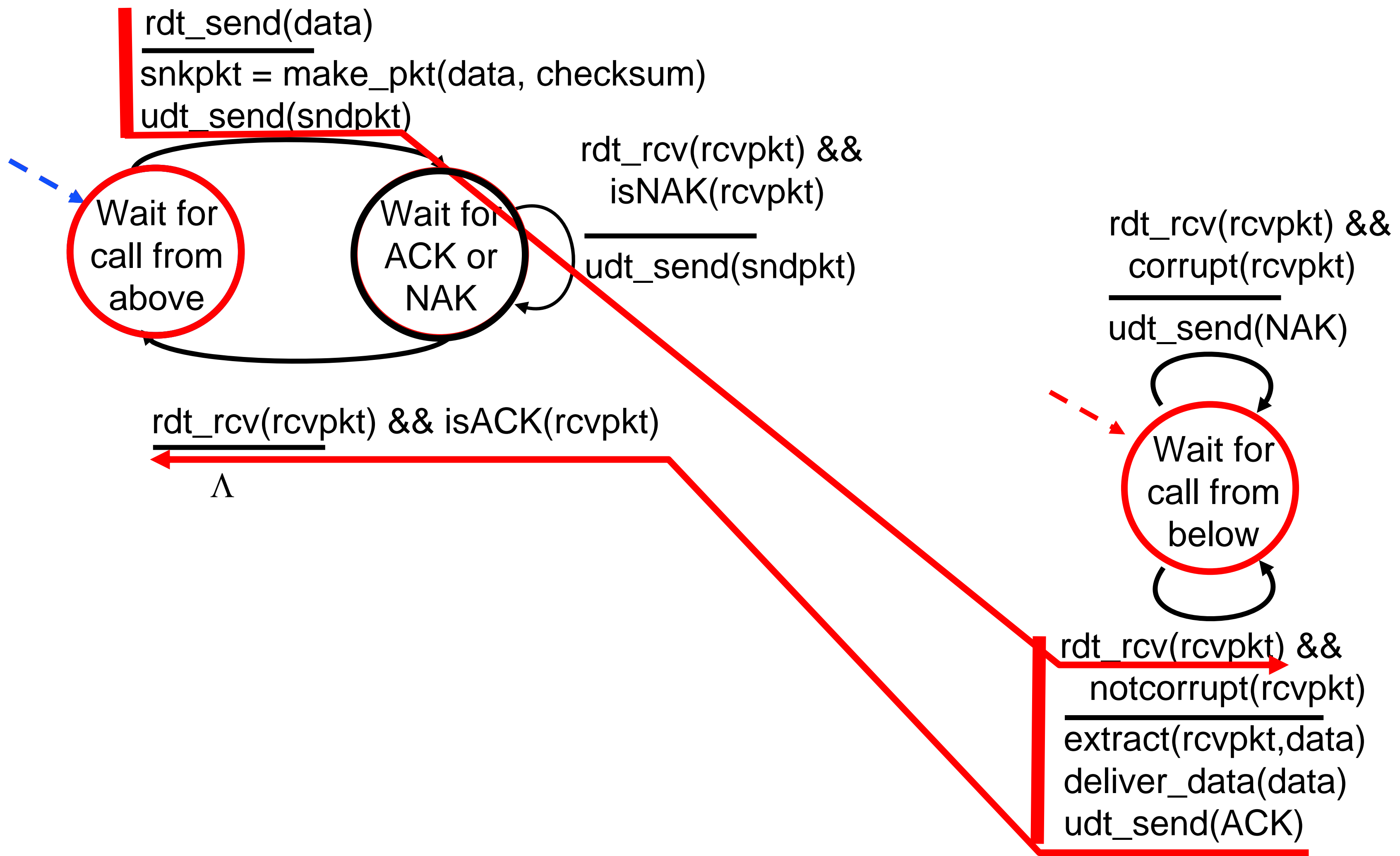
- Underlying channel may flip bits in packet
  - checksum to detect bit errors
- *The* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - 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

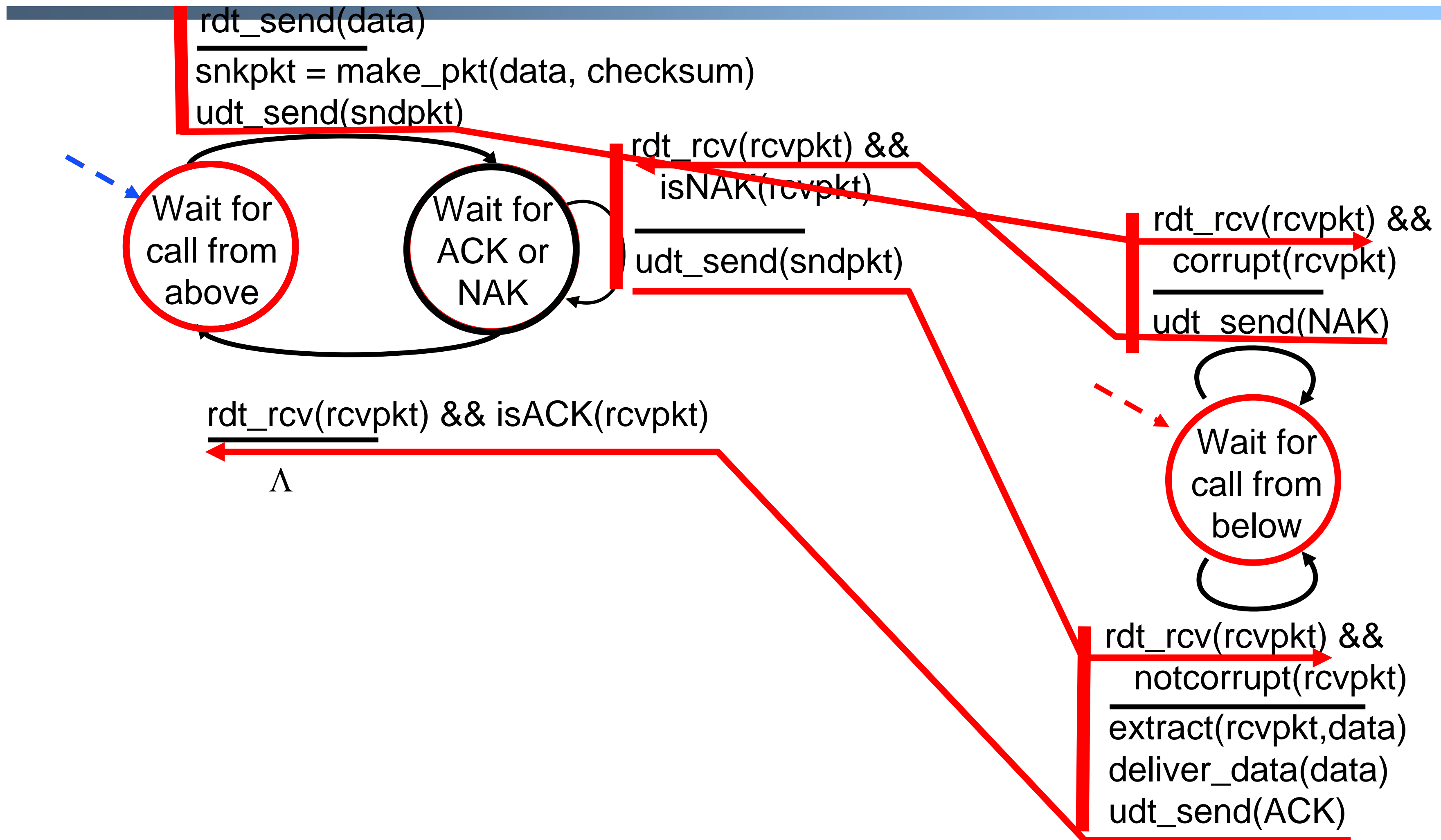




rdt2.0: operation with no errors



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

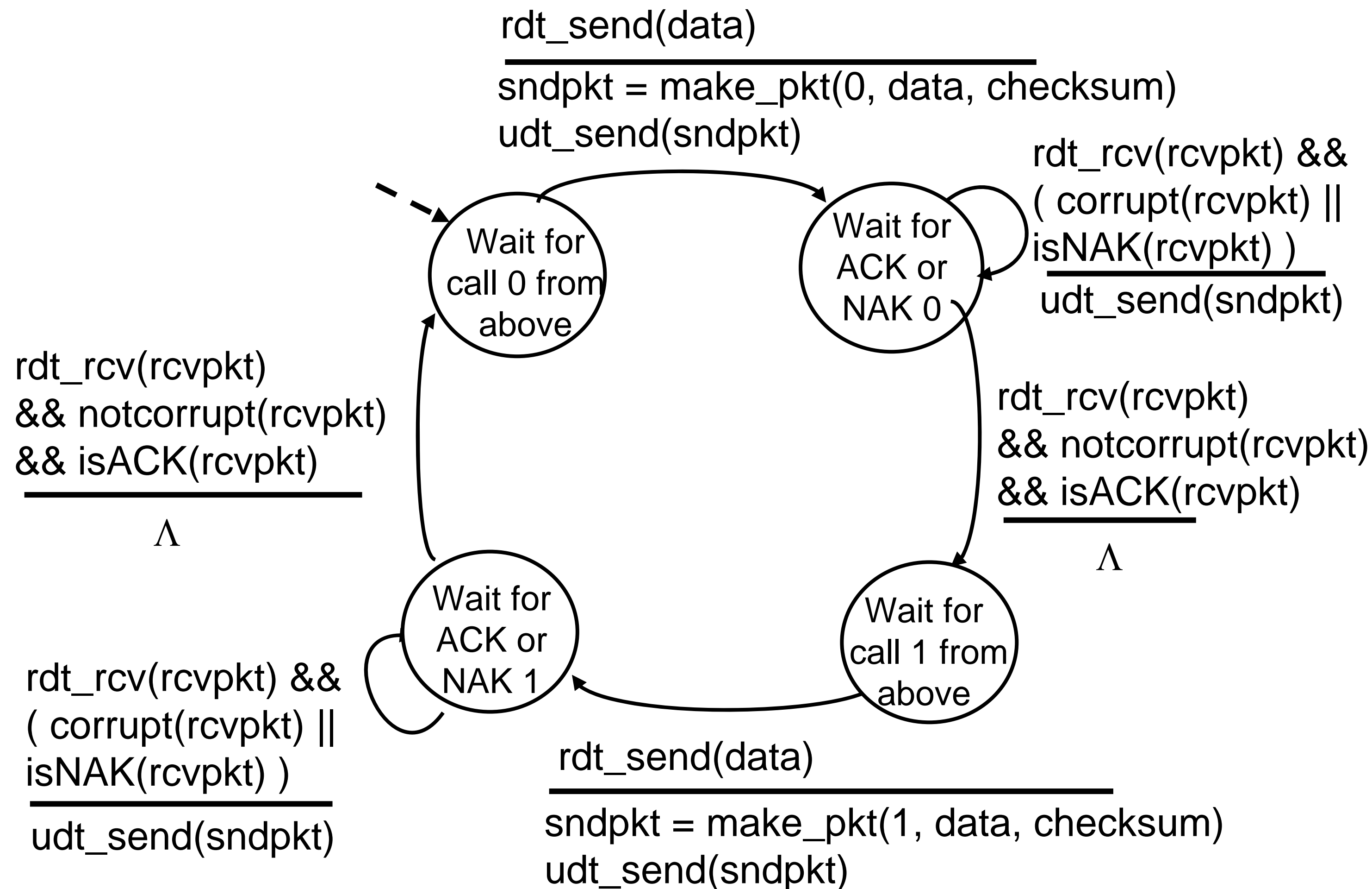
Handling duplicates:

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

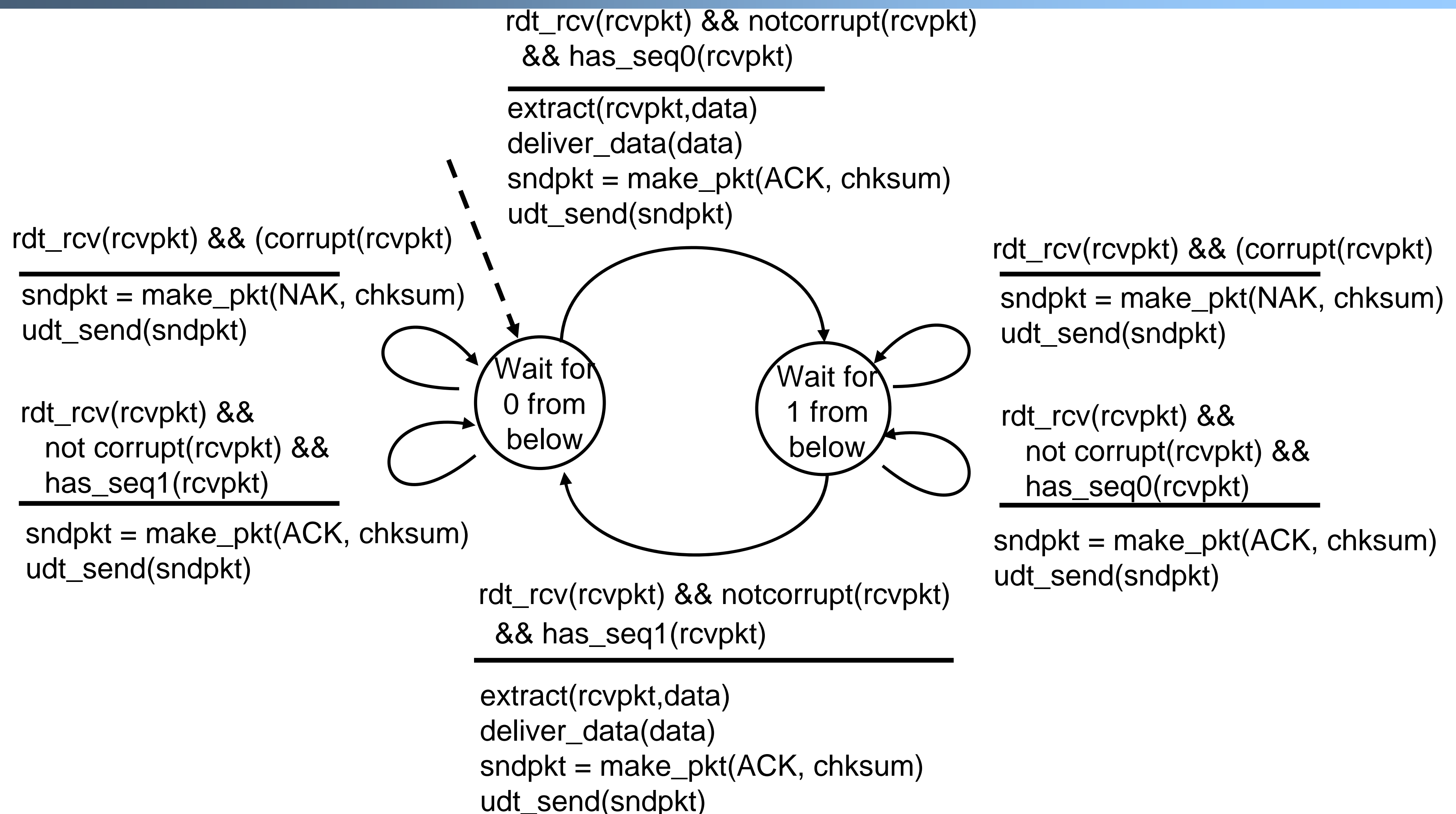
stop and wait

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. #

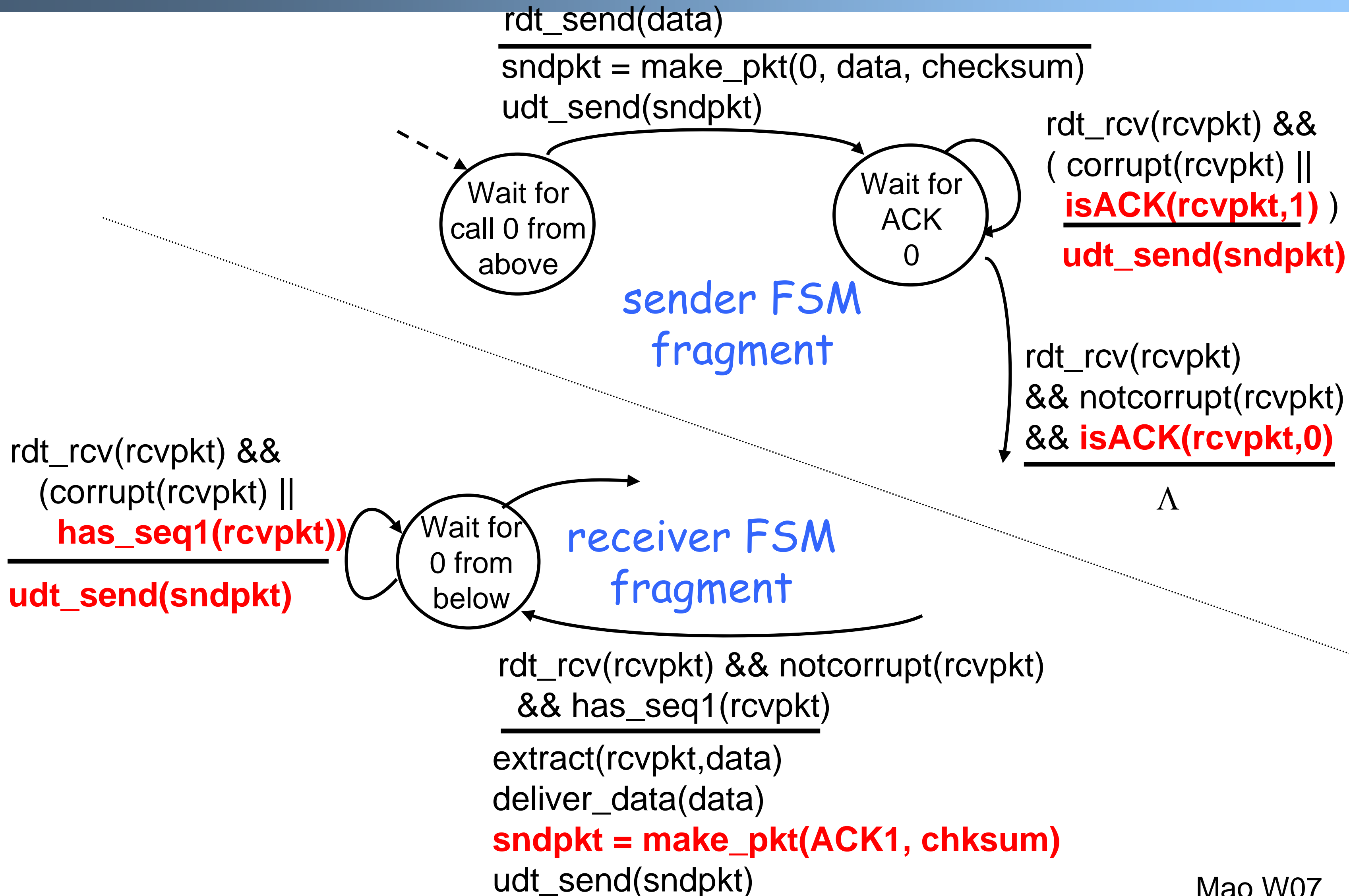
Receiver:

- 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

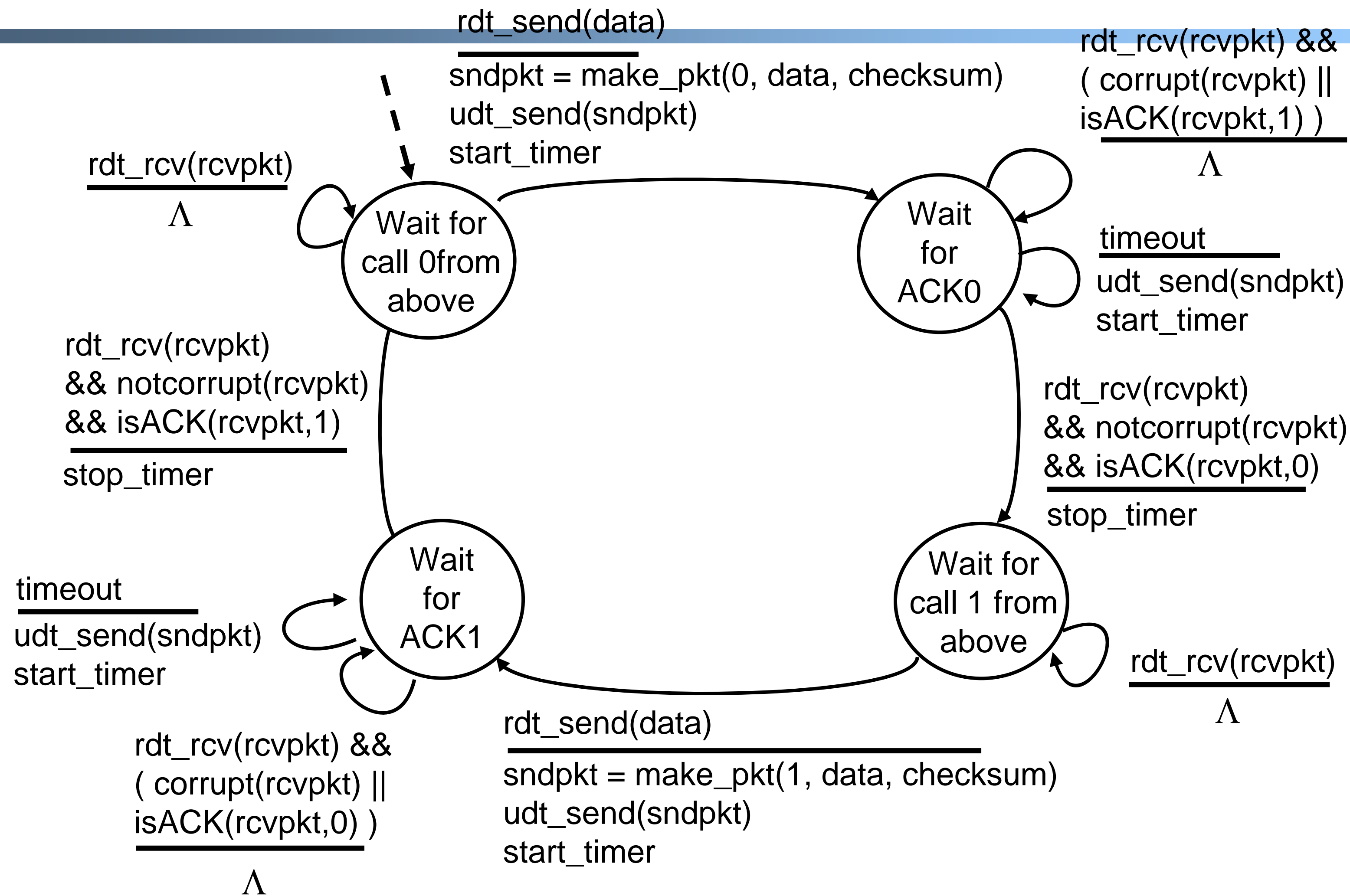
New assumption: underlying channel can also lose packets (data or ACKs)

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

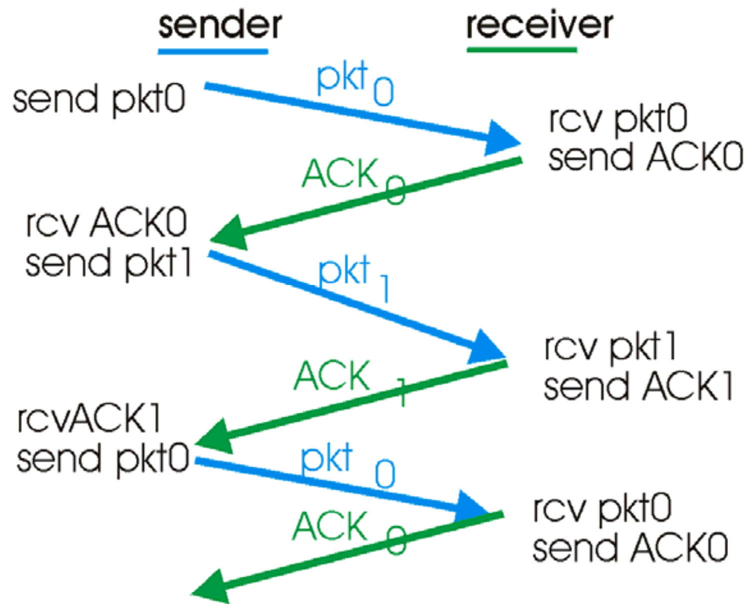
Approach: sender waits "reasonable" amount of time for ACK

- 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

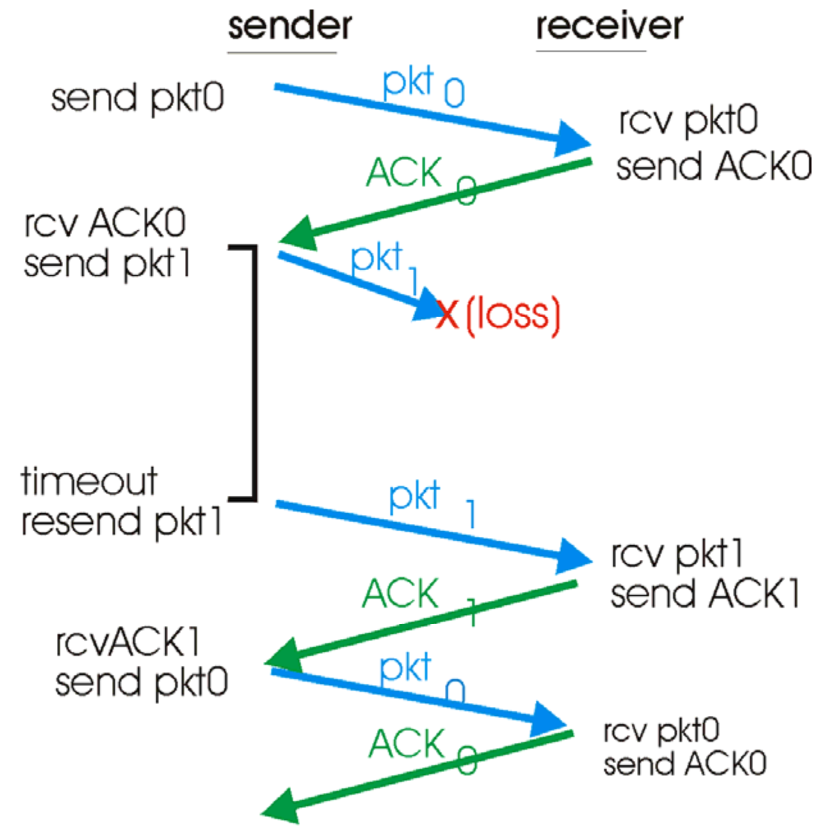
rdt3.0 sender



rdt3.0 in action

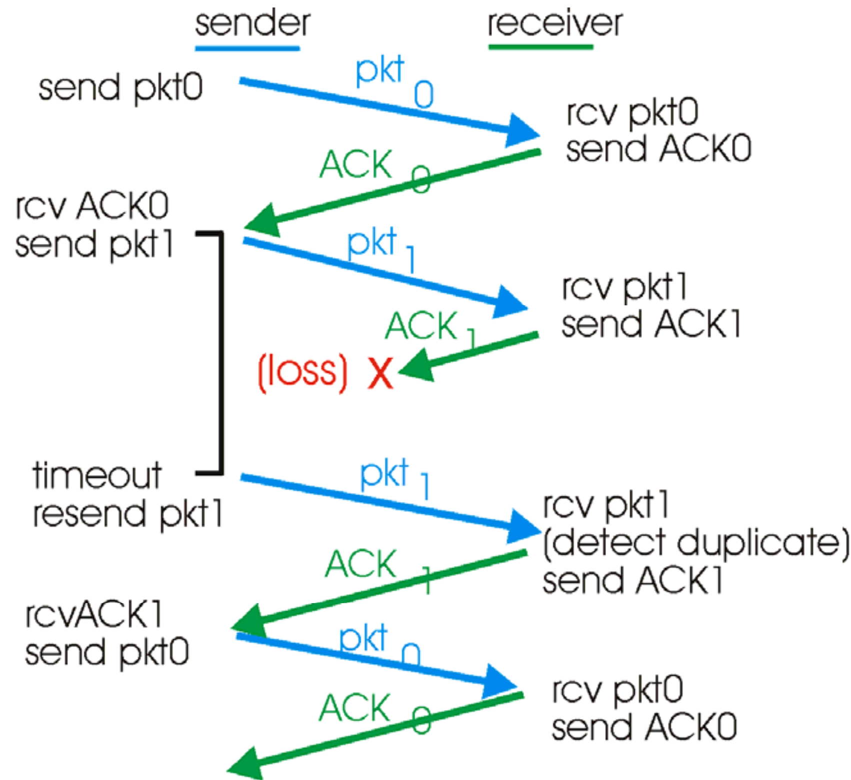


(a) operation with no loss

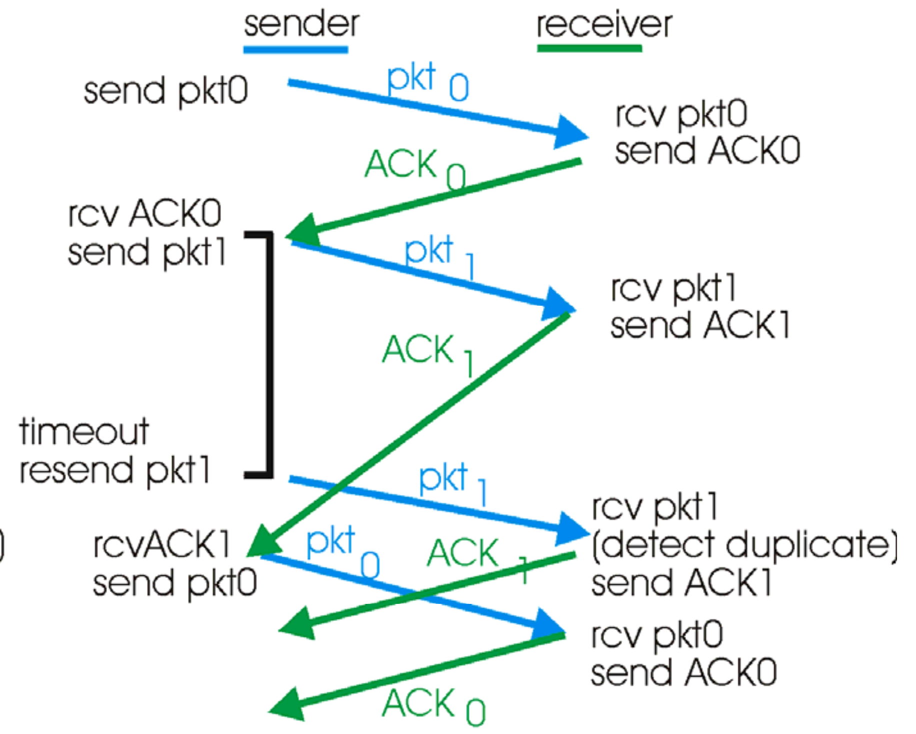


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

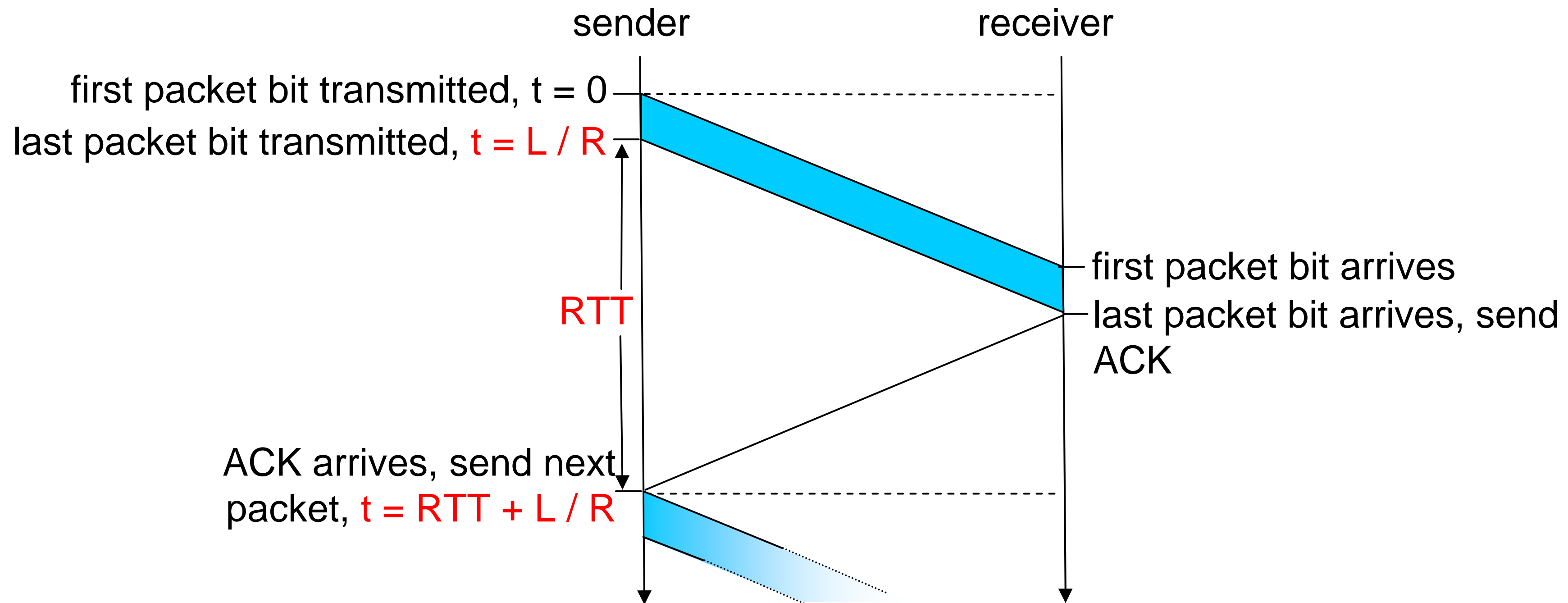
- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

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

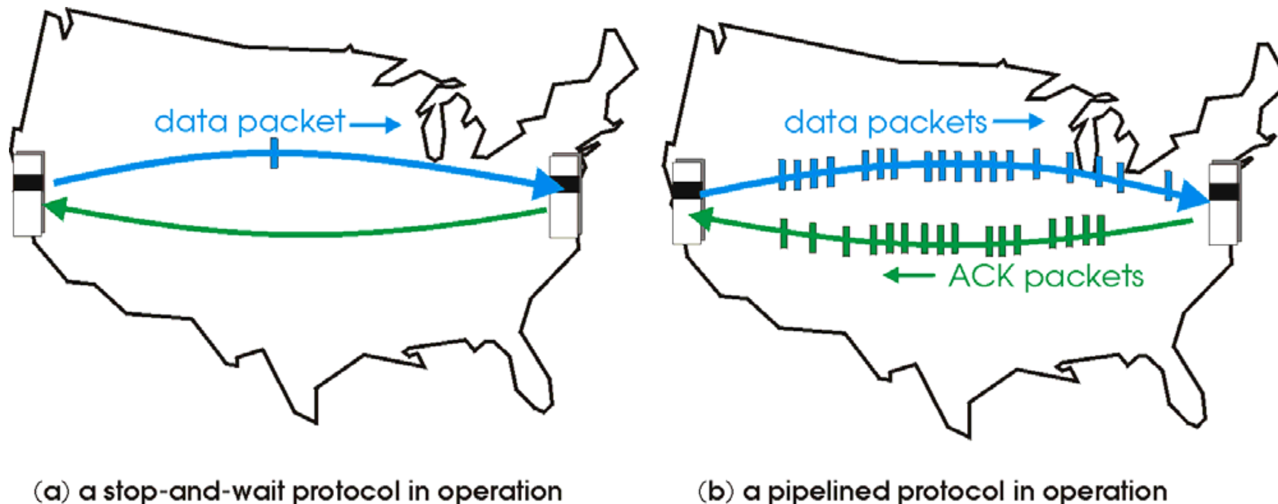


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

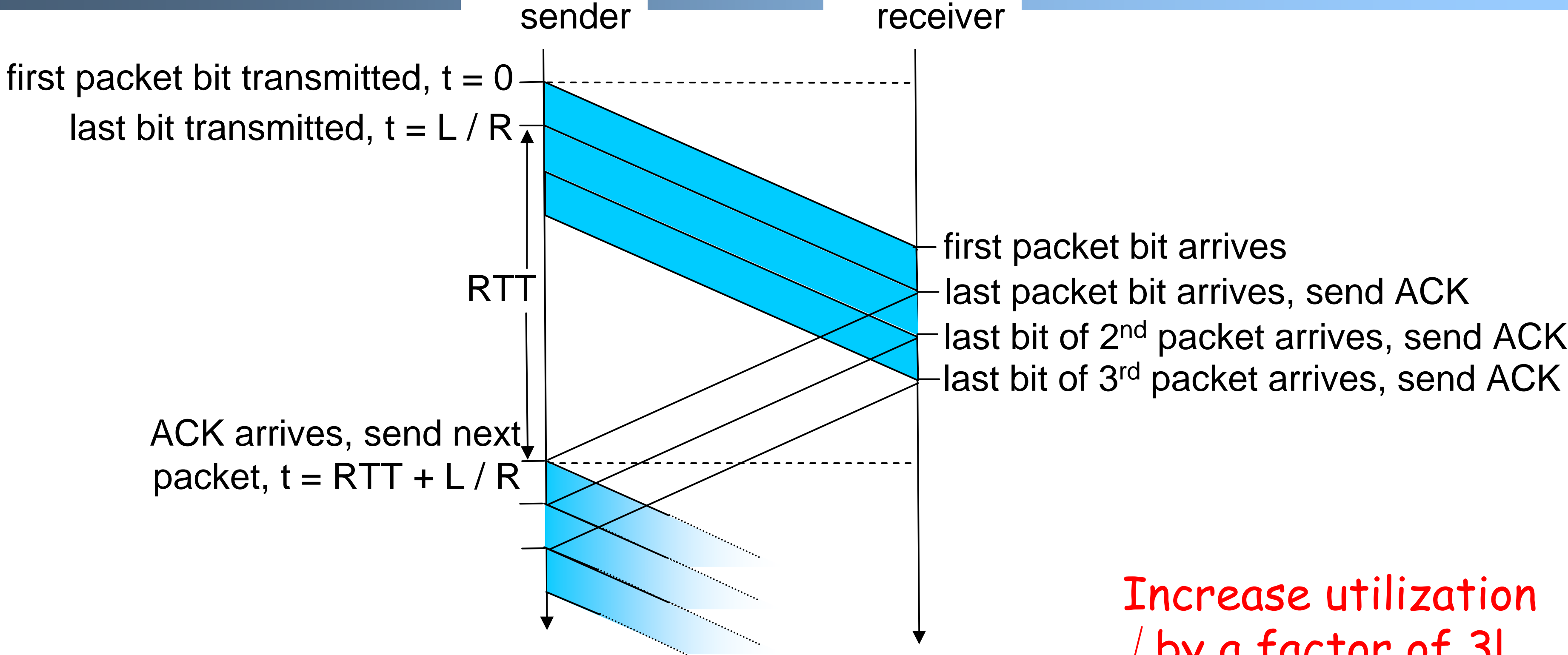


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization

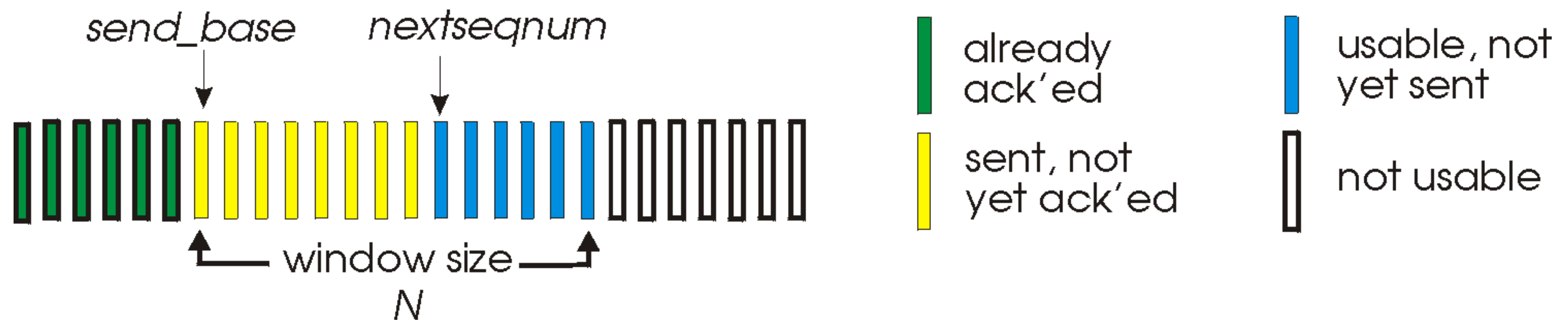


$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

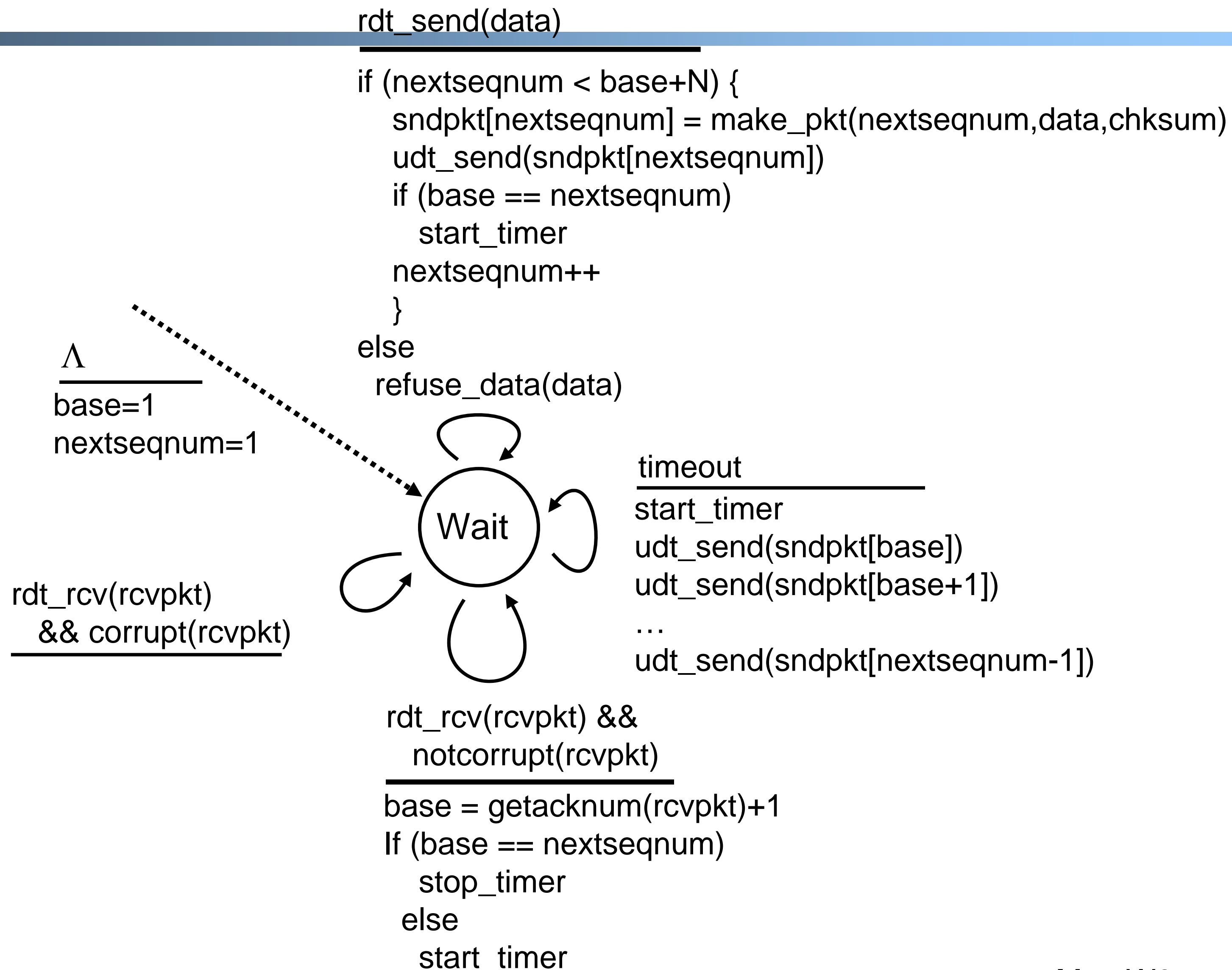
Sender:

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

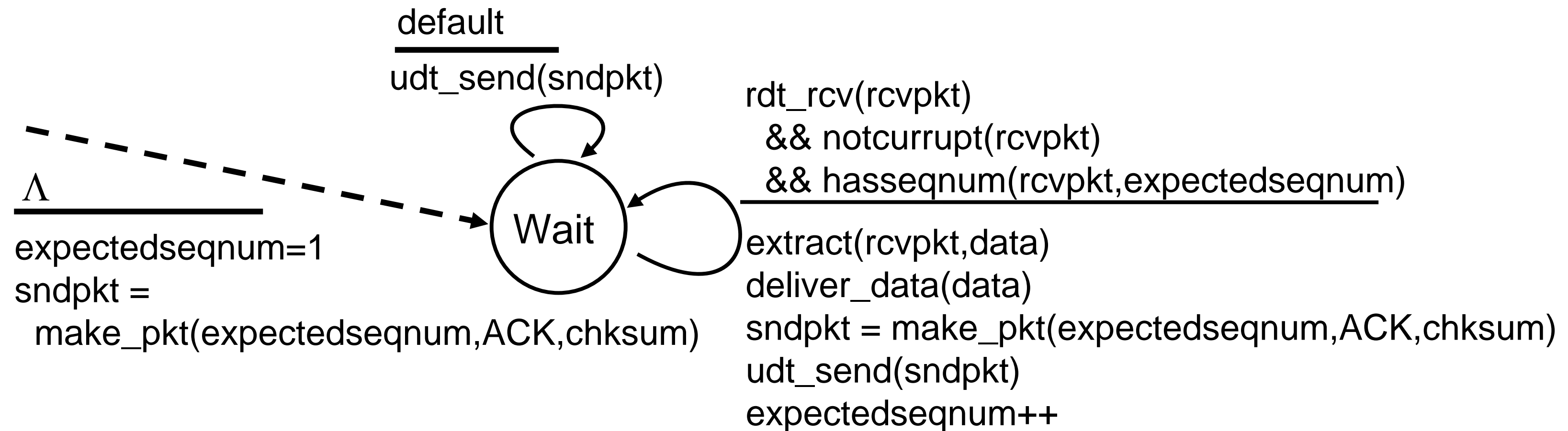


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

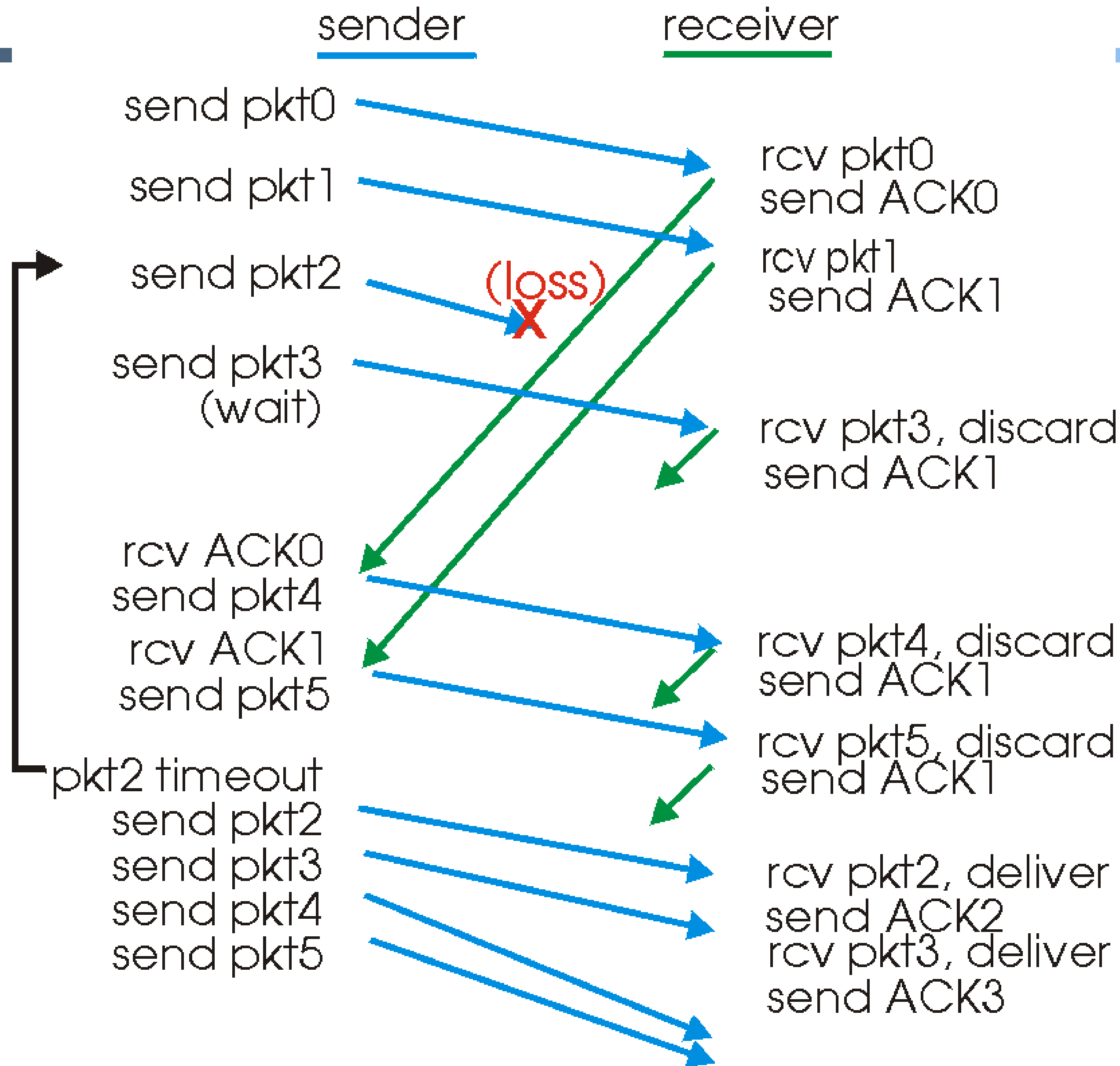


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #

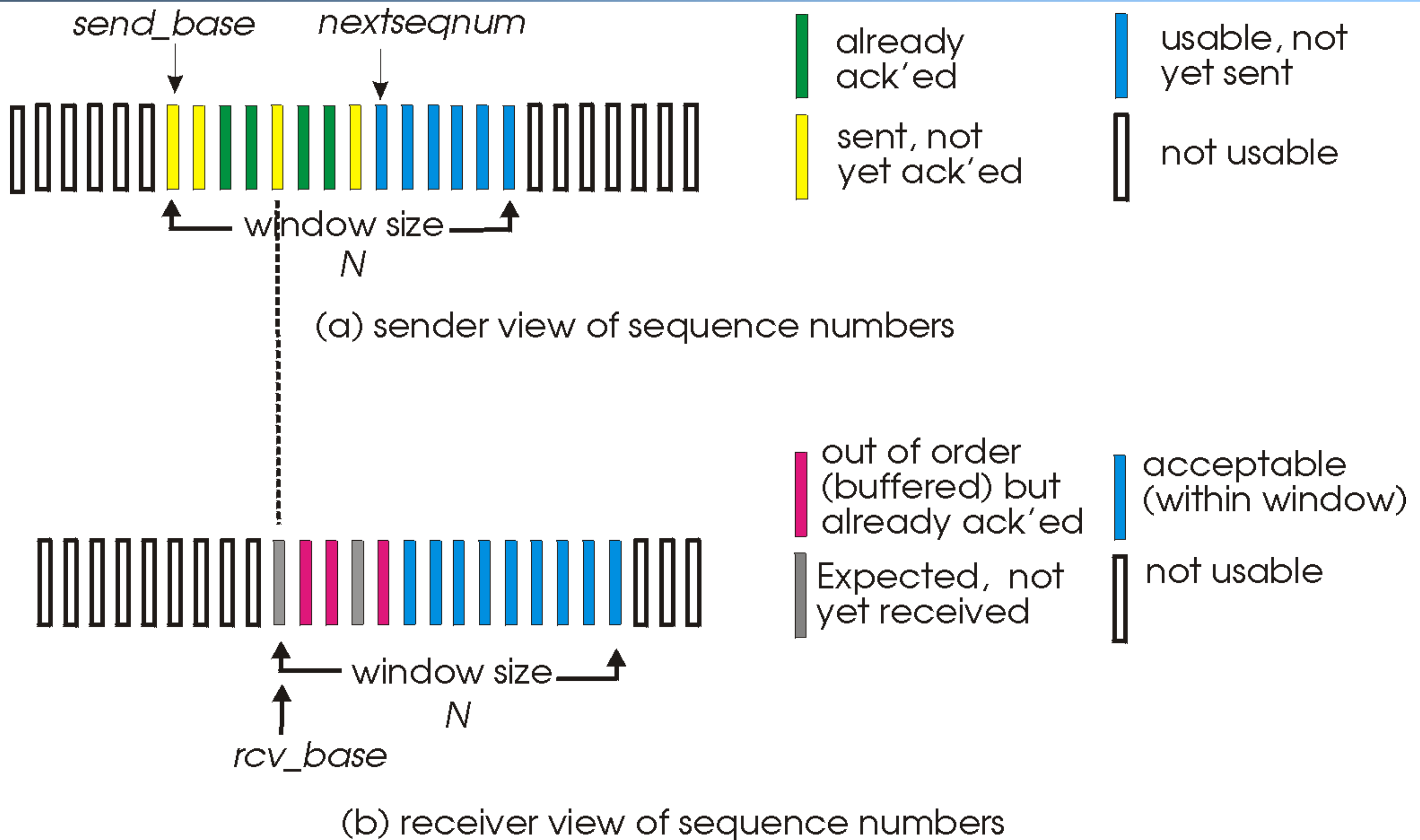
GBN in action



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

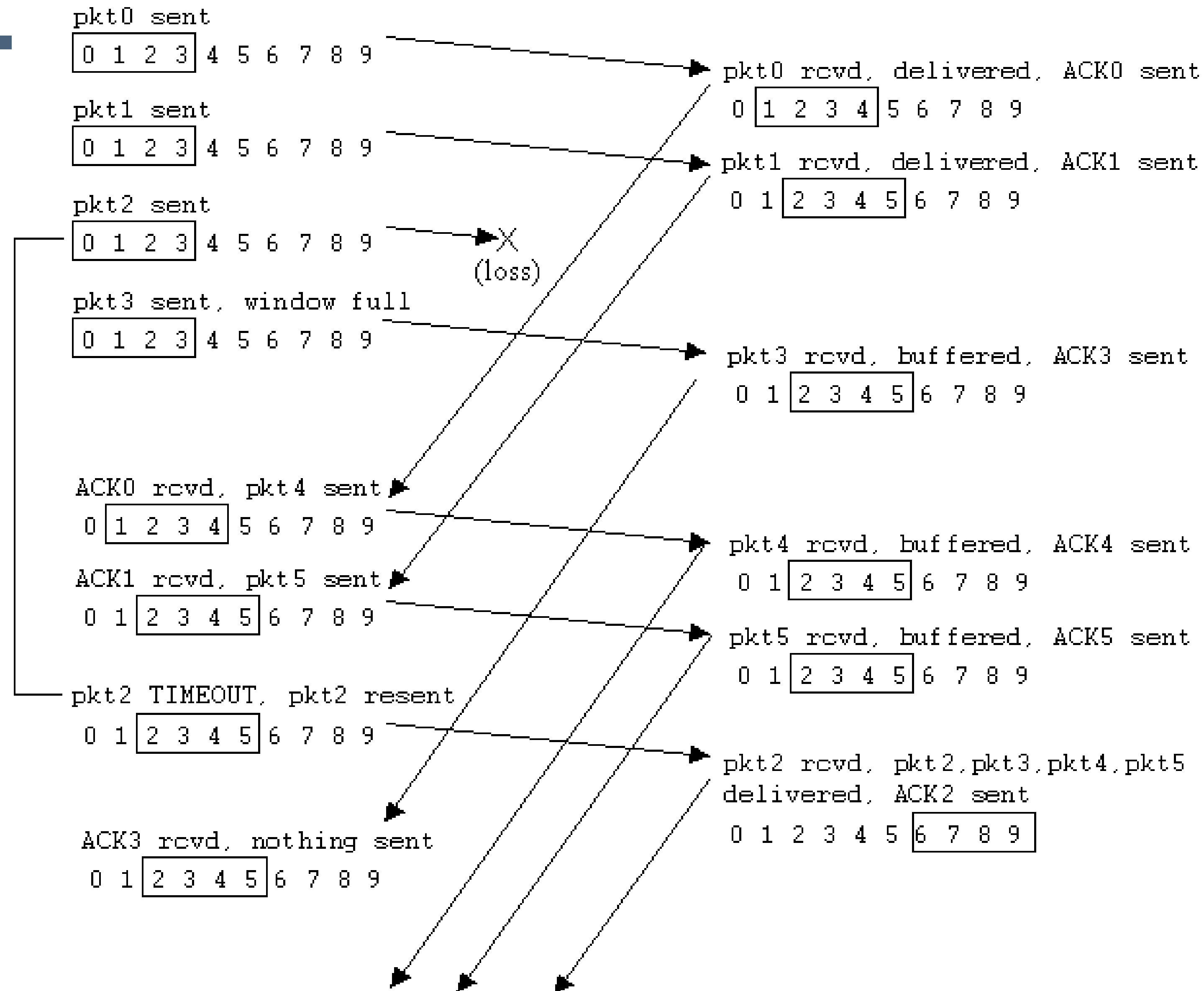
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

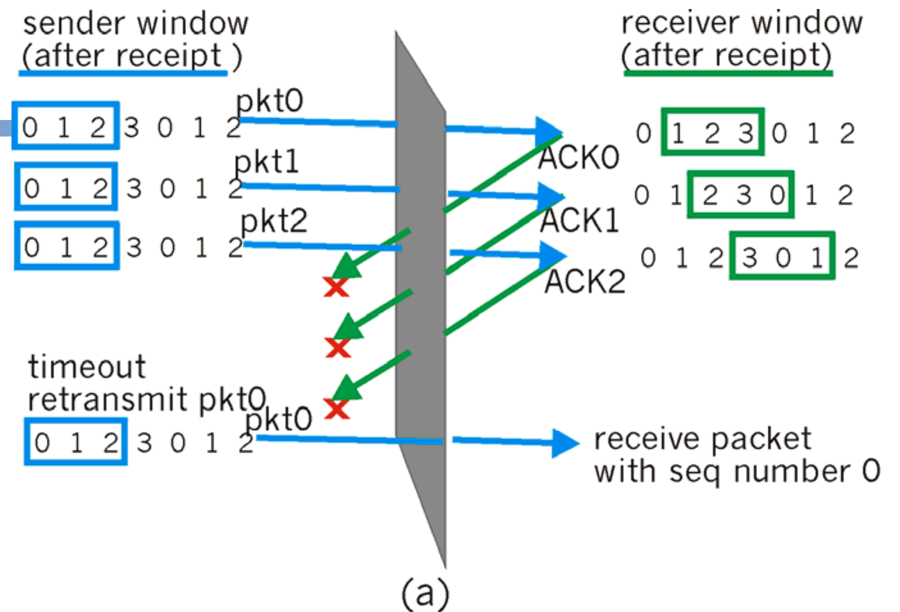
Selective repeat in action

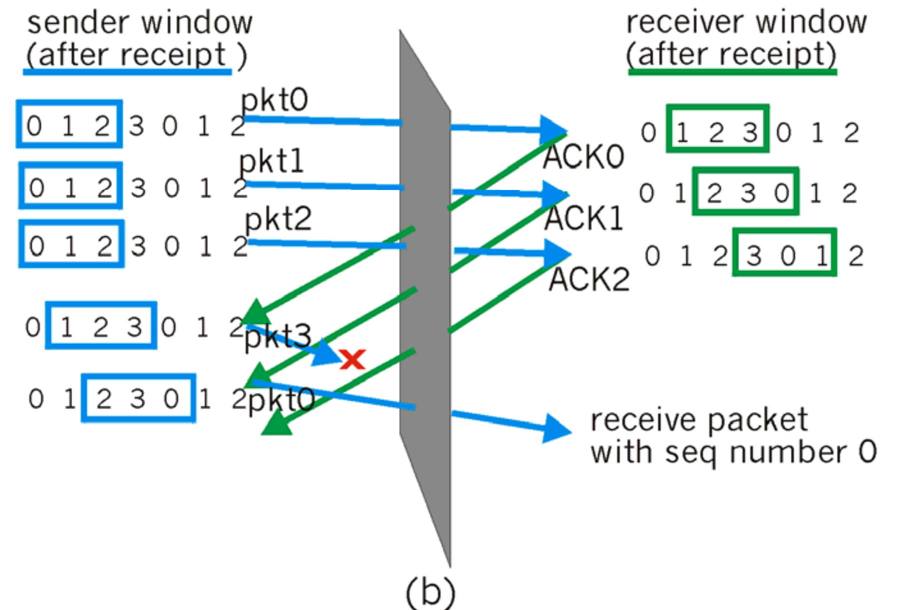


Selective repeat: dilemma

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?