

## Chapter 3: Transport Layer

our goals:

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

Transport Layer 3-2

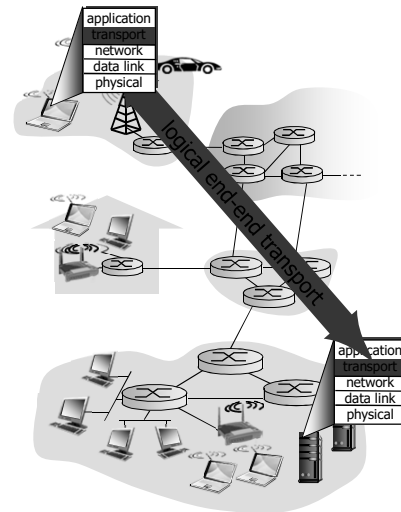
## Chapter 3 outline

- 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

Transport Layer 3-3

## 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 Layer 3-4

## Transport vs. network layer

- *network layer*: logical communication between hosts
- *transport layer*: logical communication between processes
  - relies on, enhances, network layer services

### household analogy:

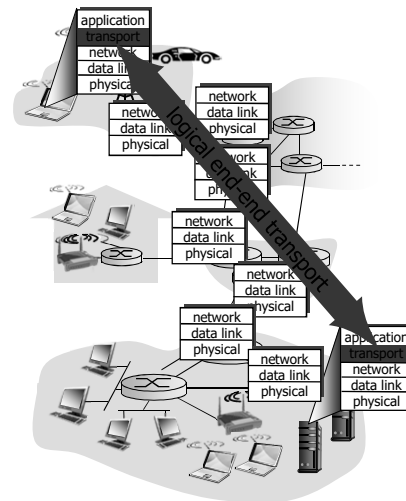
12 kids in Ann's house sending letters to 12 kids in Bill's house:

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

Transport Layer 3-5

# 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



Transport Layer 3-6

## Chapter 3 outline

- |  |  |
|--|--|
| 3.1 transport-layer services             | 3.5 connection-oriented transport: TCP <ul style="list-style-type: none"><li>• segment structure</li><li>• reliable data transfer</li><li>• flow control</li><li>• connection management</li></ul> |
| 3.2 multiplexing and demultiplexing      |  |
| 3.3 connectionless transport: UDP        | 3.6 principles of congestion control   |
| 3.4 principles of reliable data transfer | 3.7 TCP congestion control   |

Transport Layer 3-7

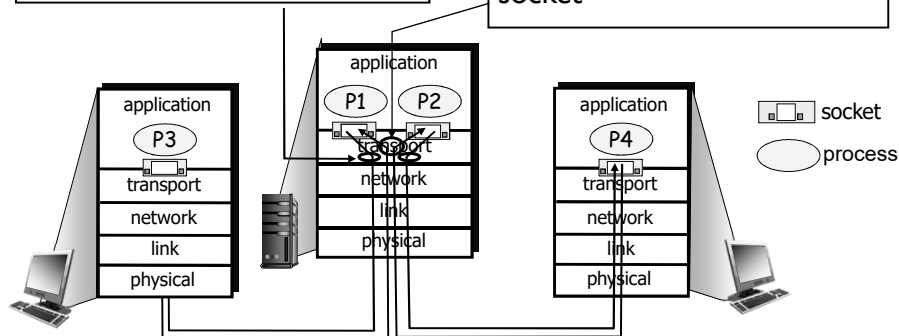
# Multiplexing/demultiplexing

*multiplexing at sender:*

handle data from multiple sockets, add transport header (later used for demultiplexing)

*demultiplexing at receiver:*

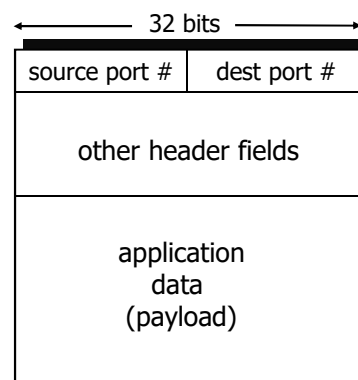
use header info to deliver received segments to correct socket



Transport Layer 3-8

## How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Transport Layer 3-9

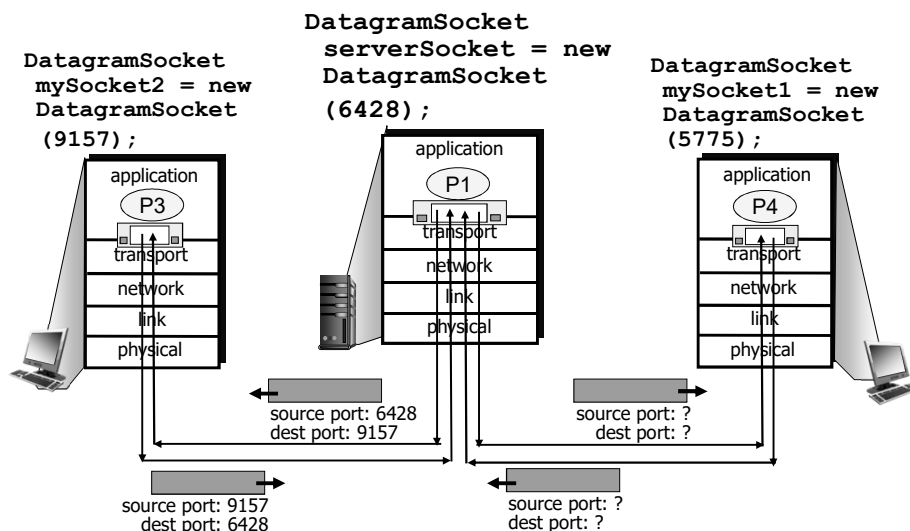
## Connectionless demultiplexing

- *recall*: created socket has host-local port #:
 

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```
  - *recall*: when creating datagram to send into UDP socket, must specify
    - destination IP address
    - destination port #
- 
- when host receives UDP segment:
    - checks destination port # in segment
    - directs UDP segment to socket with that port #
- ➔
- IP datagrams with same *dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Transport Layer 3-10

## Connectionless demux: example



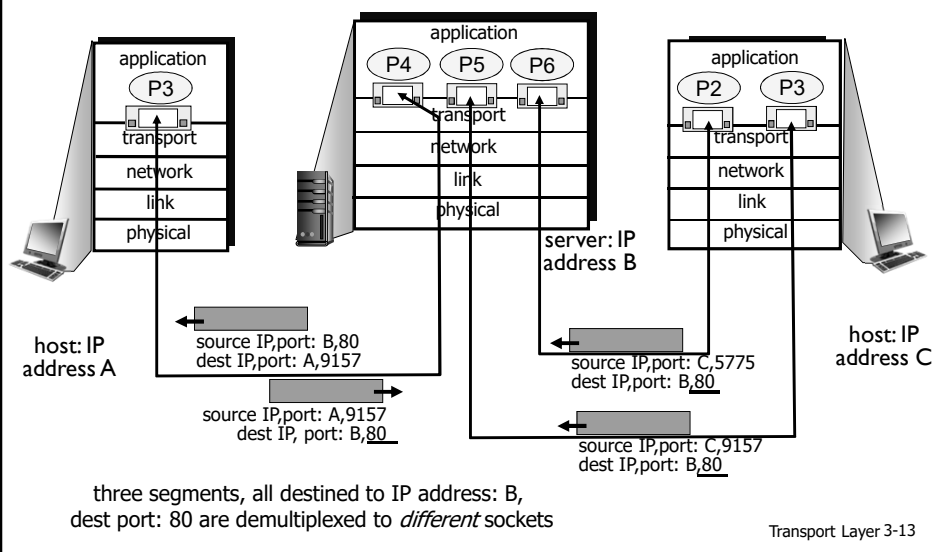
Transport Layer 3-11

## Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver 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

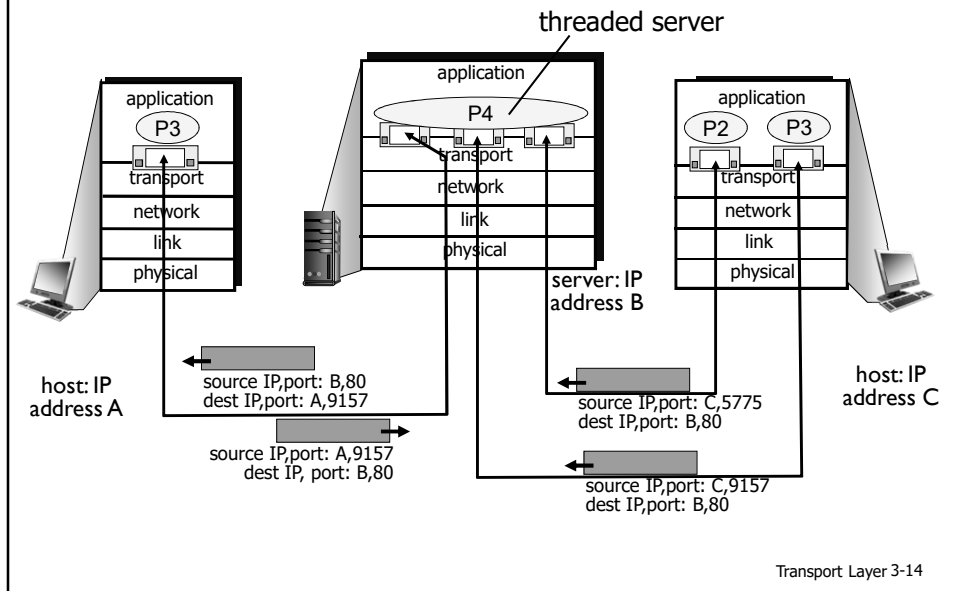
Transport Layer 3-12

## Connection-oriented demux: example



Transport Layer 3-13

## Connection-oriented demux: example



## 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

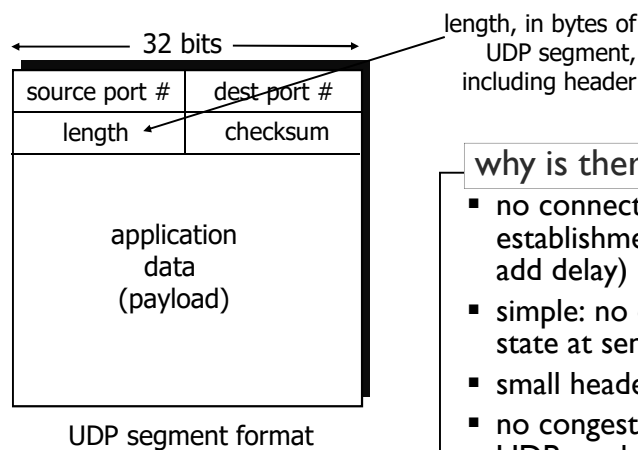
Transport Layer 3-15

## 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
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

Transport Layer 3-16

## UDP: segment header



### why is there a UDP?

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

Transport Layer 3-17



## UDP checksum

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

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one'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*  
....

Transport Layer 3-18

## Internet checksum: example

example: add two 16-bit integers

	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	<hr/>																
wraparound	①	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
	<hr/>																
sum	1	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	1	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

*Note:* when adding numbers, a carryout from the most significant bit needs to be added to the result

\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

Transport Layer 3-19

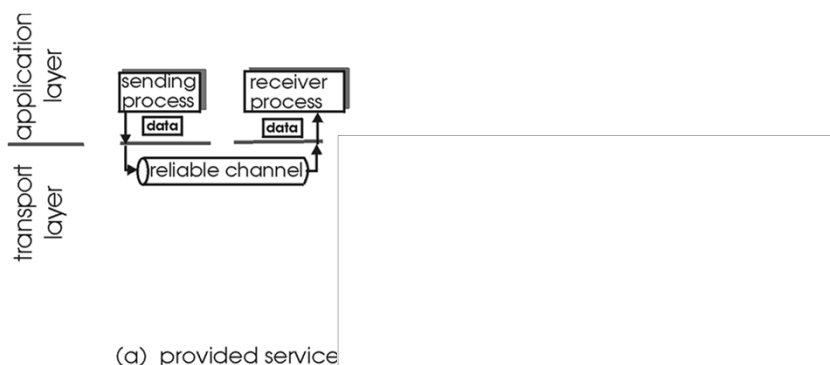
## Chapter 3 outline

- 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

Transport Layer 3-20

## Principles of reliable data transfer

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

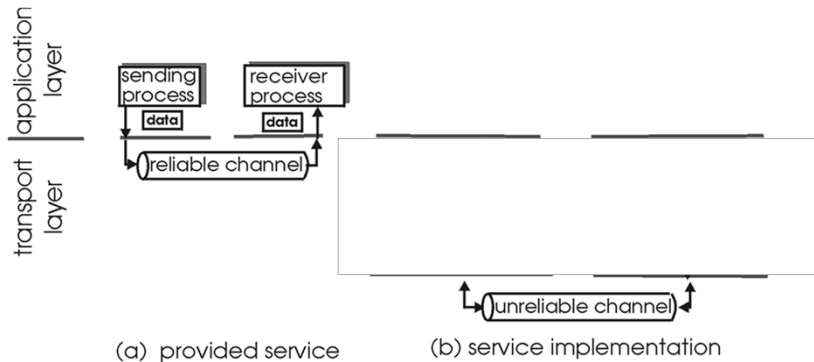


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Transport Layer 3-21

## Principles of reliable data transfer

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

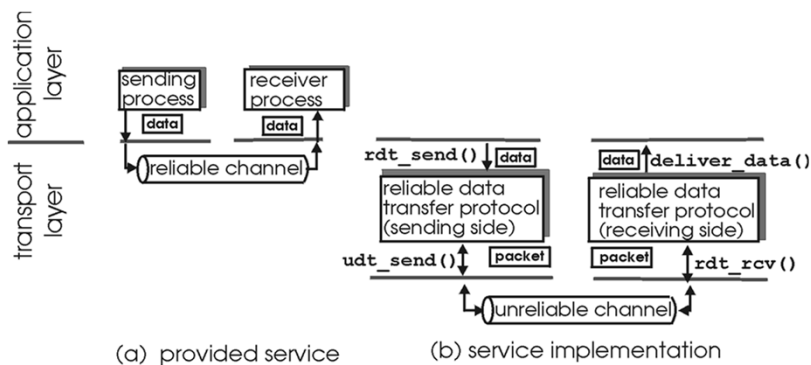


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Transport Layer 3-22

## Principles of reliable data transfer

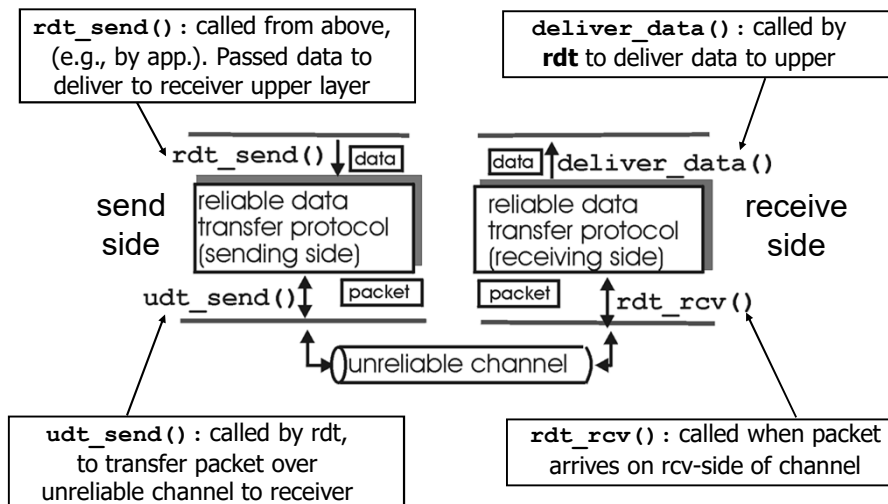
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Transport Layer 3-23

## Reliable data transfer: getting started

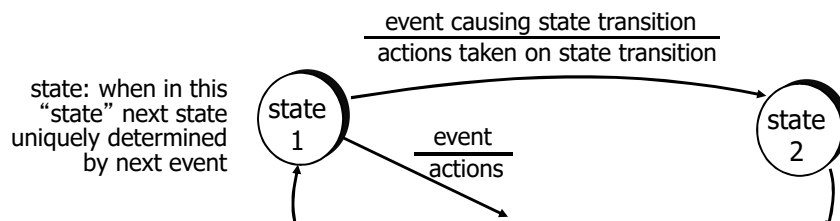


Transport Layer 3-24

## Reliable data transfer: getting started

we'll:

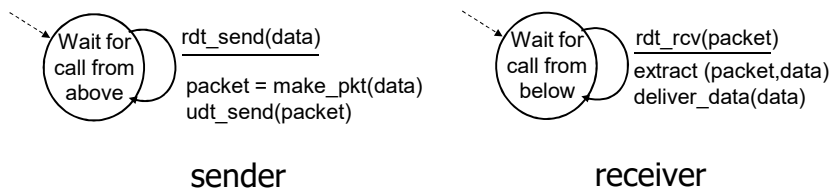
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



Transport Layer 3-25

## 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



Transport Layer 3-26

## 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?*

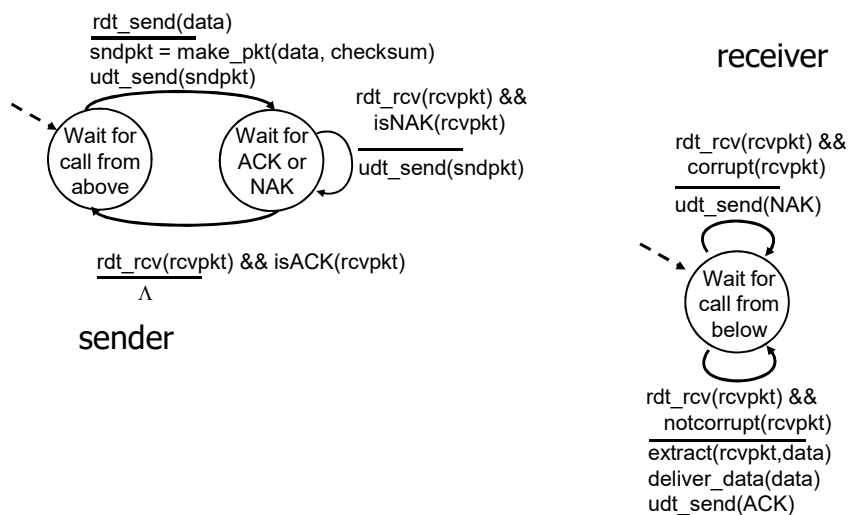
Transport Layer 3-27

## 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

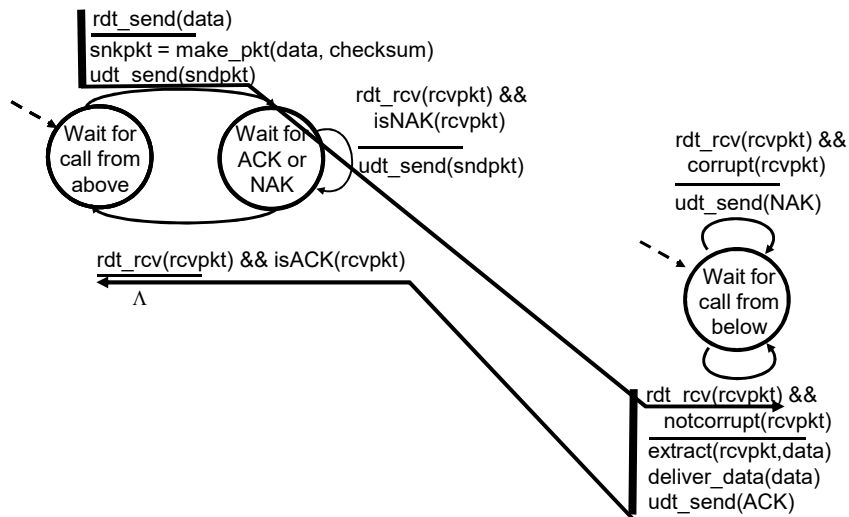
Transport Layer 3-28

## rdt2.0: FSM specification



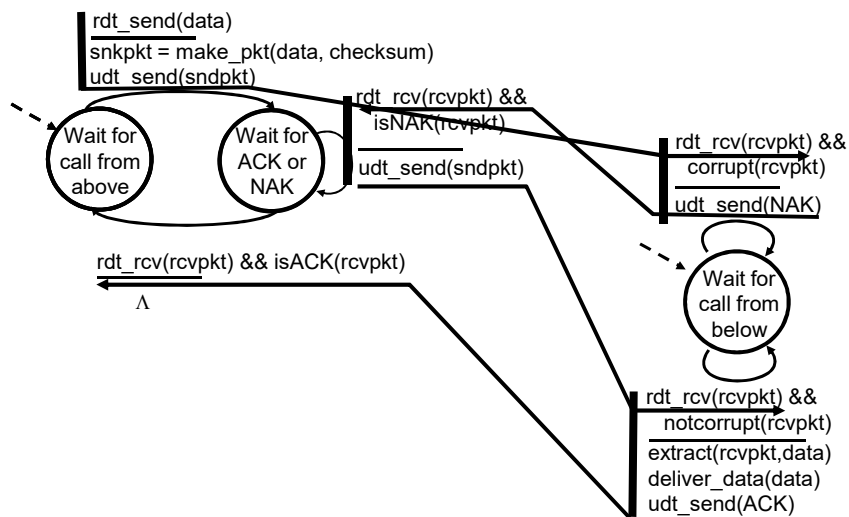
Transport Layer 3-29

## rdt2.0: operation with no errors



Transport Layer 3-30

## rdt2.0: error scenario



Transport Layer 3-31

## 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

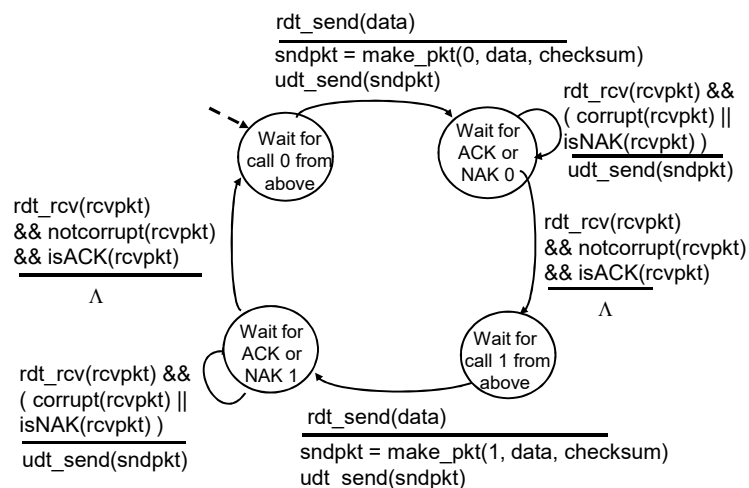
handling duplicates:

- 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

Transport Layer 3-32

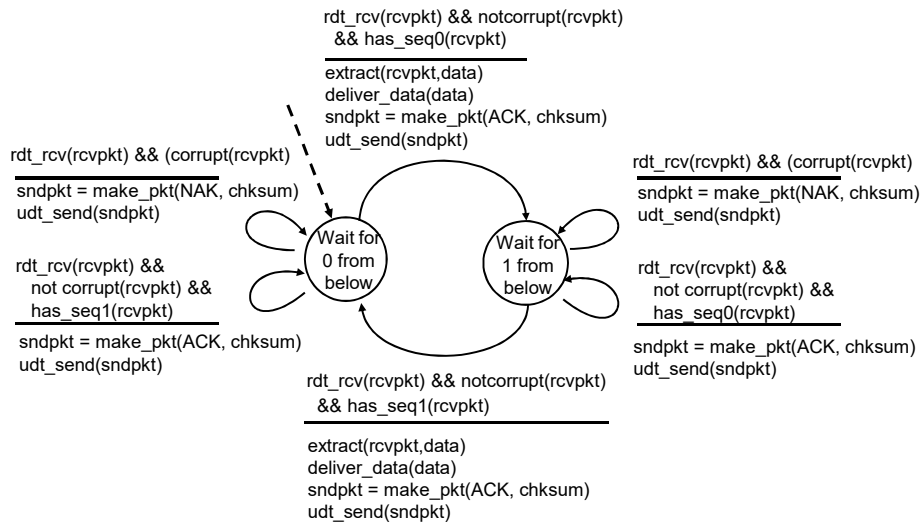
## rdt2.1: sender, handles garbled ACK/NAKs



Transport Layer 3-33



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



Transport Layer 3-34

## 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

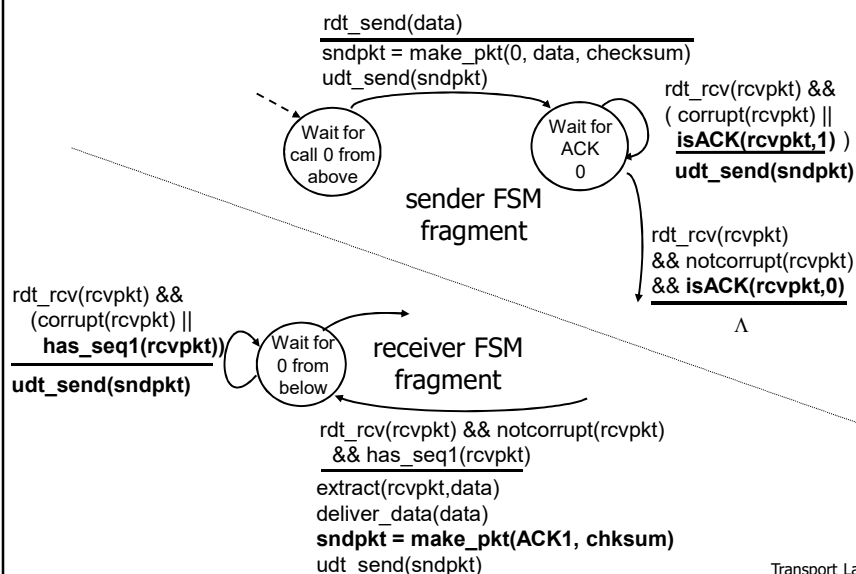
Transport Layer 3-35

## 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*

Transport Layer 3-36

## rdt2.2: sender, receiver fragments



Transport Layer 3-37

## 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

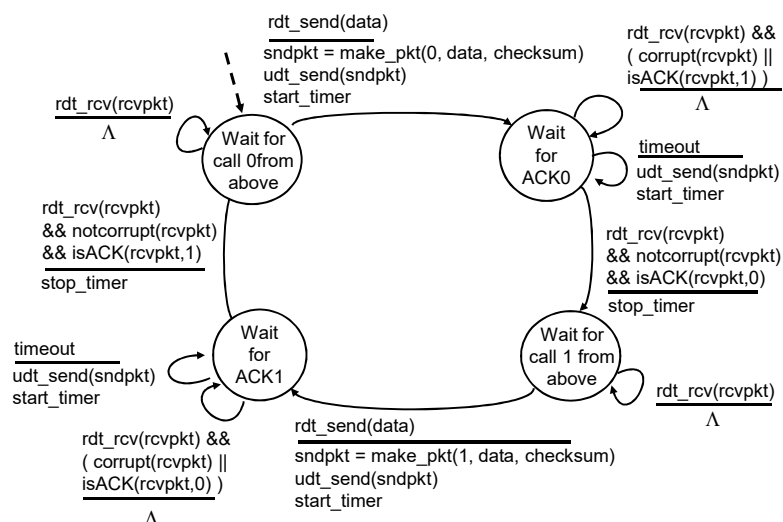
### approach: sender waits

“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

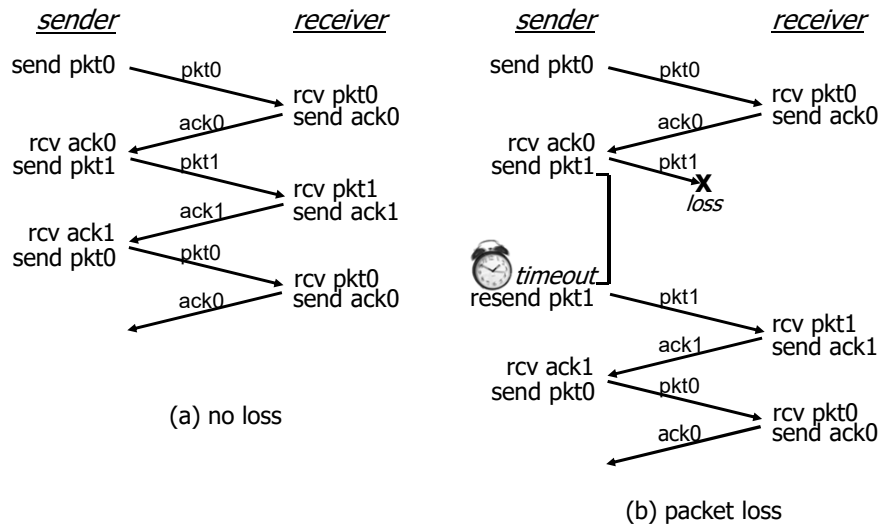
Transport Layer 3-38

## rdt3.0 sender



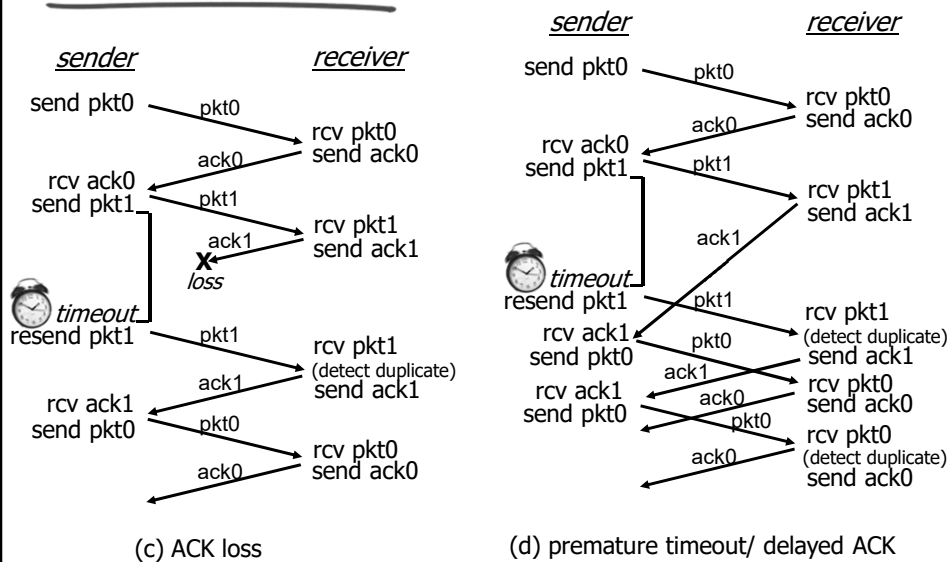
Transport Layer 3-39

## rdt3.0 in action



Transport Layer 3-40

## rdt3.0 in action



Transport Layer 3-41

## Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

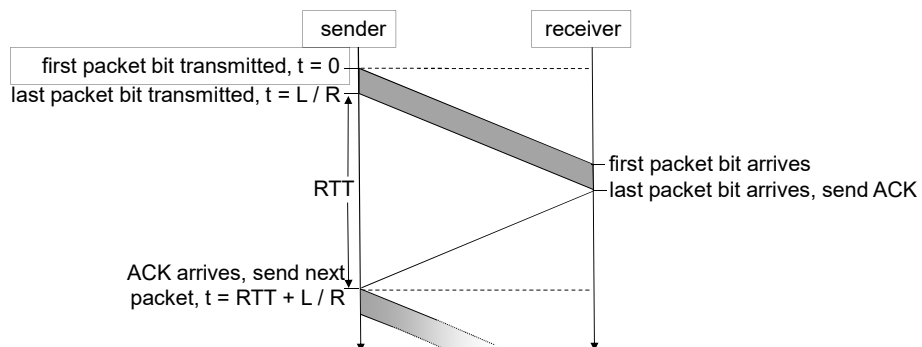
- $U_{sender}$ : utilization – fraction of time sender busy sending

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

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thrupt over 1 Gbps link
- network protocol limits use of physical resources!

Transport Layer 3-42

## rdt3.0: stop-and-wait operation



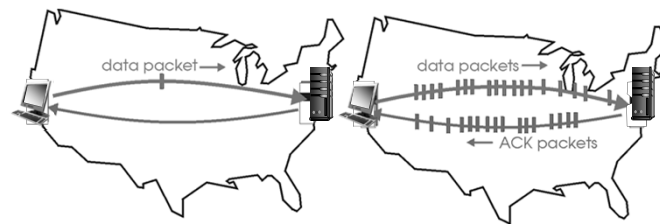
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Transport Layer 3-43

## 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



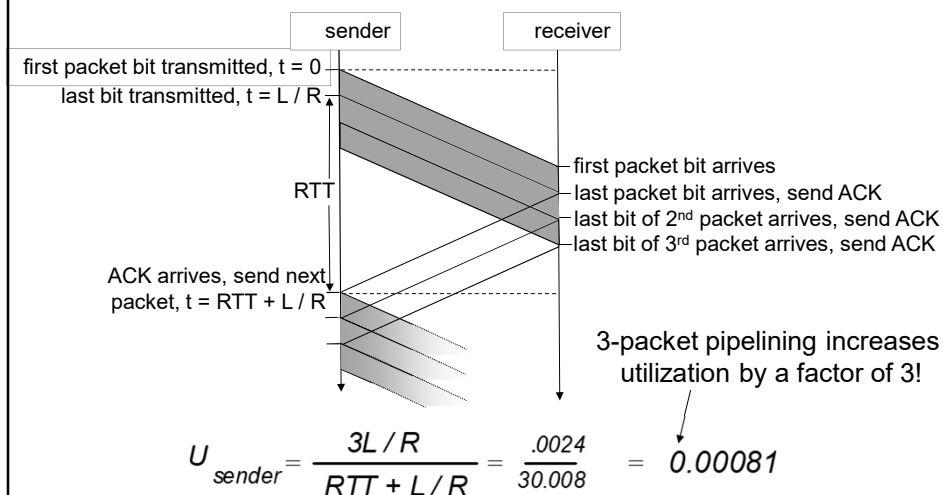
(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Transport Layer 3-44

## Pipelining: increased utilization



Transport Layer 3-45

## 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

Transport Layer 3-46