

# Part 1: Networking Review

## Goals:

- review key topics from intro networks course
  - equalize backgrounds
  - identify remedial work
  - ease into course

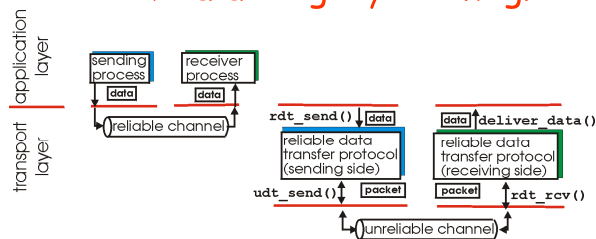
## Overview:

- overview
- **error control**
- flow control
- congestion control
- routing
- LANs
- addressing
- synthesis:
  - "a day in the life"
  - control timescales

1-23

# Error control

- reliable point-point communication
  - A generic problem: application-to-application, over path, over link
- what's the error model:
  - bits flipped in packet?
  - packets "lost"?
  - packets delayed or reordered?
- **Error control ≠ data integrity checking!**



(a) provided service

(b) service implementation

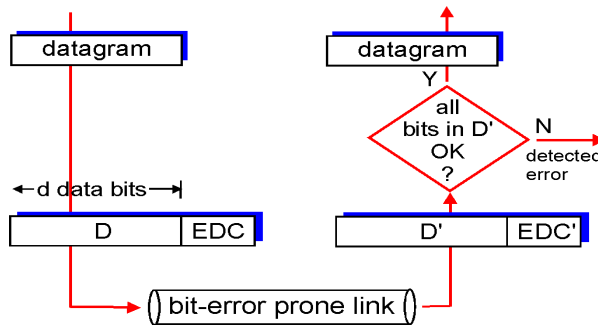
1-24

# Bit level error detection

EDC= Error Detection and Correction bits (redundancy)

D = Data protected by error checking, may include header fields

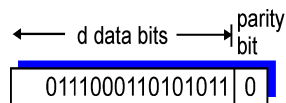
- Error detection not 100% reliable!
  - protocol may miss some errors, but rarely
  - larger EDC field yields better detection and correction



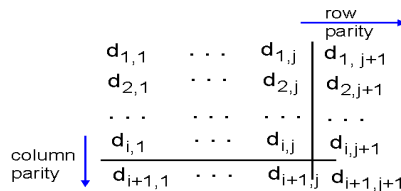
1-25

# Parity Checking

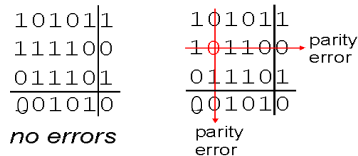
**Single Bit Parity:**  
Detect single bit errors



**Two Dimensional Bit Parity:**  
Detect and correct single bit errors



Much more powerful error detection/correction schemes:  
Cyclic Redundancy Check (CRC)  
- polynomial division modulo 2



Simple form of forward error correction (FEC)

1-26

## Internet checksum

**Goal:** detect "errors" (e.g., flipped bits) in transmitted segment (note: used at transport layer *only*)

### 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?*

Benefits: easy to compute; can do incremental update; endian-independent

1-27

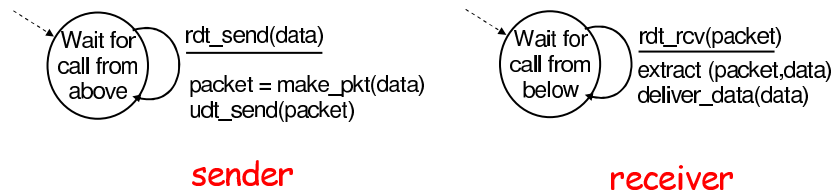
## Recovering from lost packets

- Why are packets lost: at end system, "within" network
  - Limited storage, discarded in congestion
  - outages: eventually reroute around failure (~sec recovery ties hopefully)
  - dropped at end system e.g., on NIC
- ARQ: automatic request repeat
  - sender puts sequence numbers on packets (why)
  - receiver positively or negatively acknowledges correct receipt of packet
  - sender starts (logical) timer for each packet, timeout and retransmits

1-28

## 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



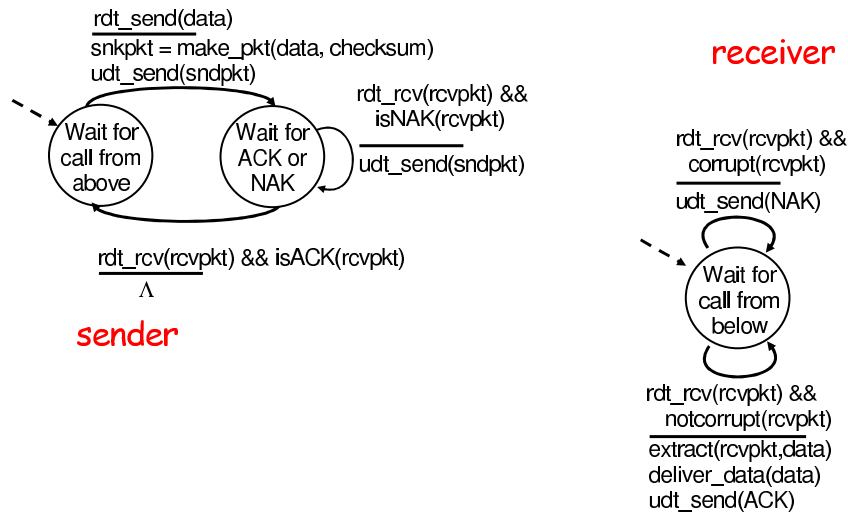
1-29

## Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender

1-30

## rdt2.0: FSM specification



1-31

## 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

### What to do?

- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

### Handling duplicates:

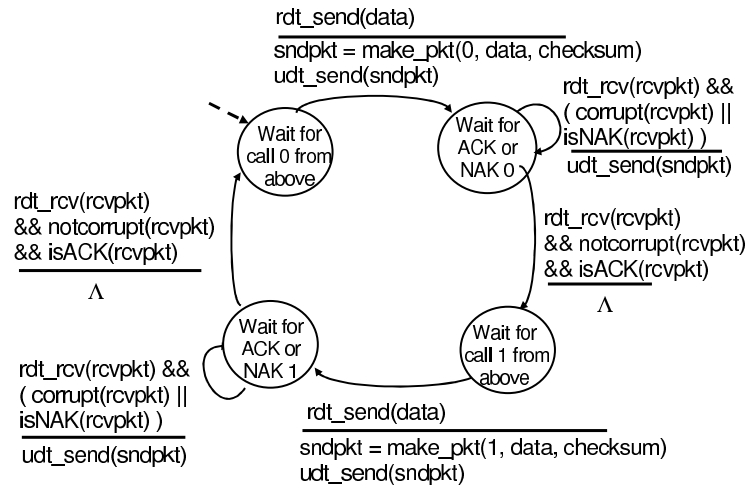
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

### stop and wait

Sender sends one packet, then waits for receiver response

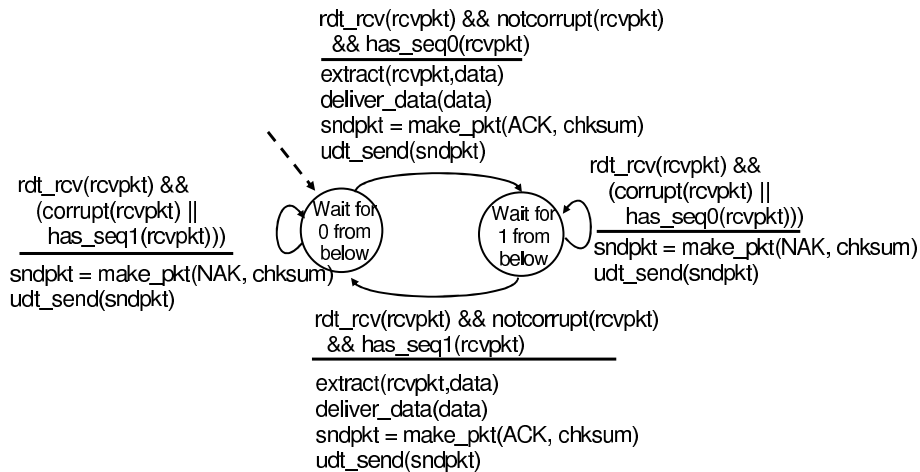
1-32

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



1-33

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



1-34

## rdt2.1: discussion

### Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- FSM has 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 #

1-35

## 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*

1-36

## rdt3.0: channels with errors AND loss

**Assumption:** underlying channel can also lose packets (data or ACKs)

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

□ Why seq #s

- detect reordering
- ACK, NAKing
- Detect missing packet
- Duplicate detection due to retransmissions

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

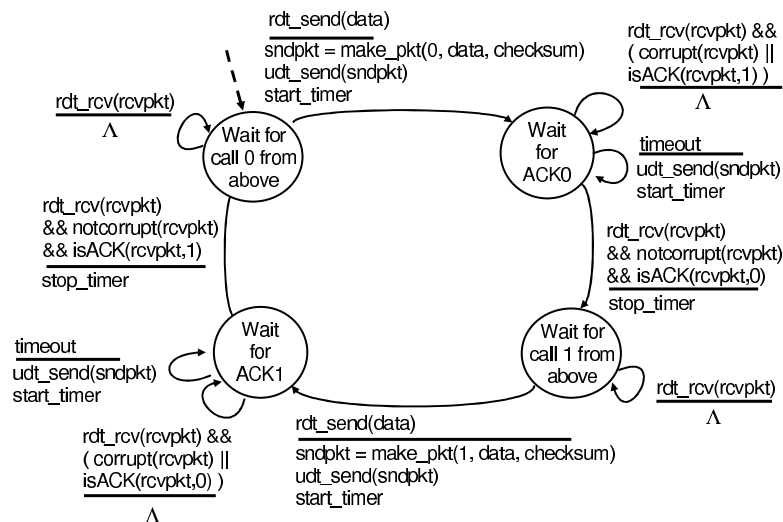
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):

- retransmission will be *duplicate*, but use of 0,1 seq. #'s already handles this
- receiver must specify seq # of pkt being ACKed

- requires countdown timer

1-37

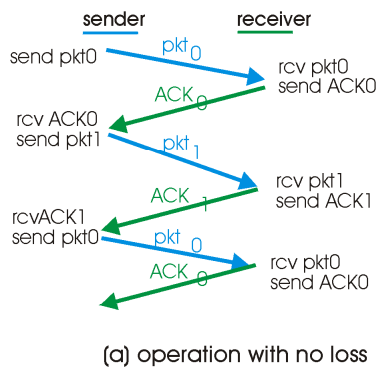
## rdt3.0 sender



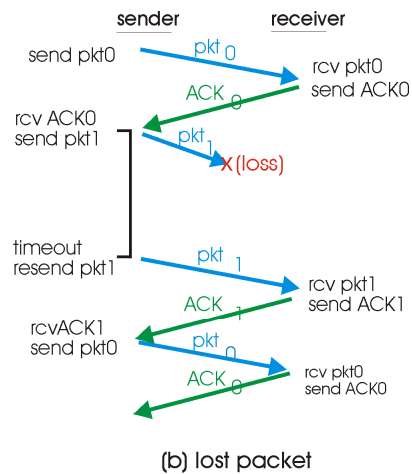
FSM specification of sender (details not important)

1-38

## rdt3.0 in action



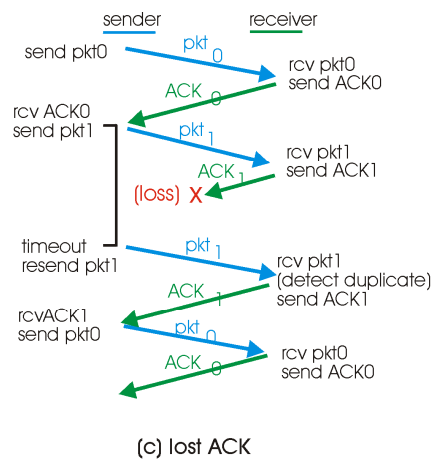
(a) operation with no loss



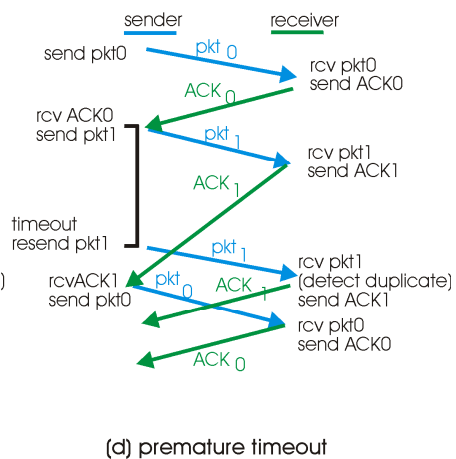
(b) lost packet

1-39

## rdt3.0 in action



(c) lost ACK

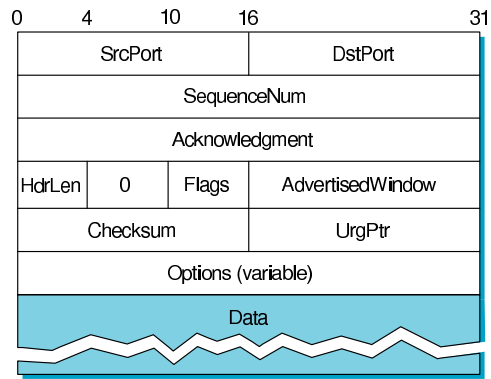


(d) premature timeout

1-40

## Case Study: TCP

TCP Segment Format:



Flags: SYN, FIN, RESET, PUSH, URG, ACK