CS 3205 COMPUTER NETWORKS

JAN-MAY 2019

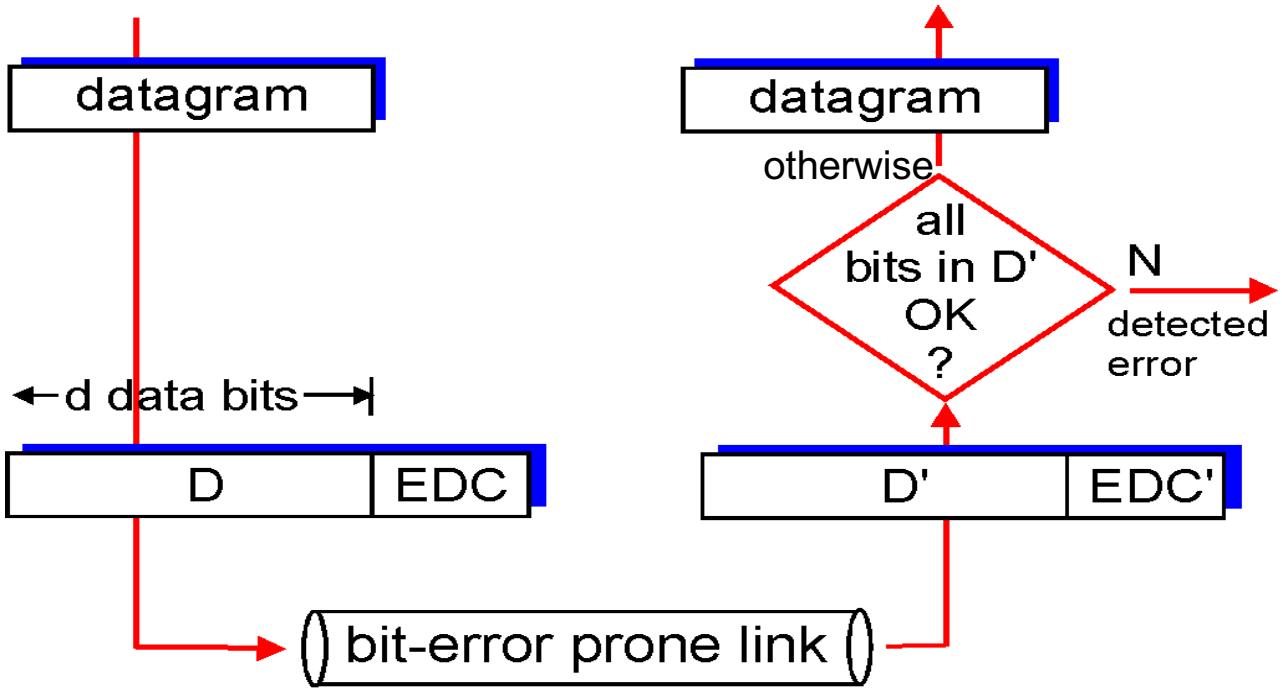
LECTURE 10: 6TH FEB 2019

Error Detection and Correction discussed in class is covered in **Section 5.2,** Computer Networking – A top-down approach, Kurose and Ross, 6th Edition.

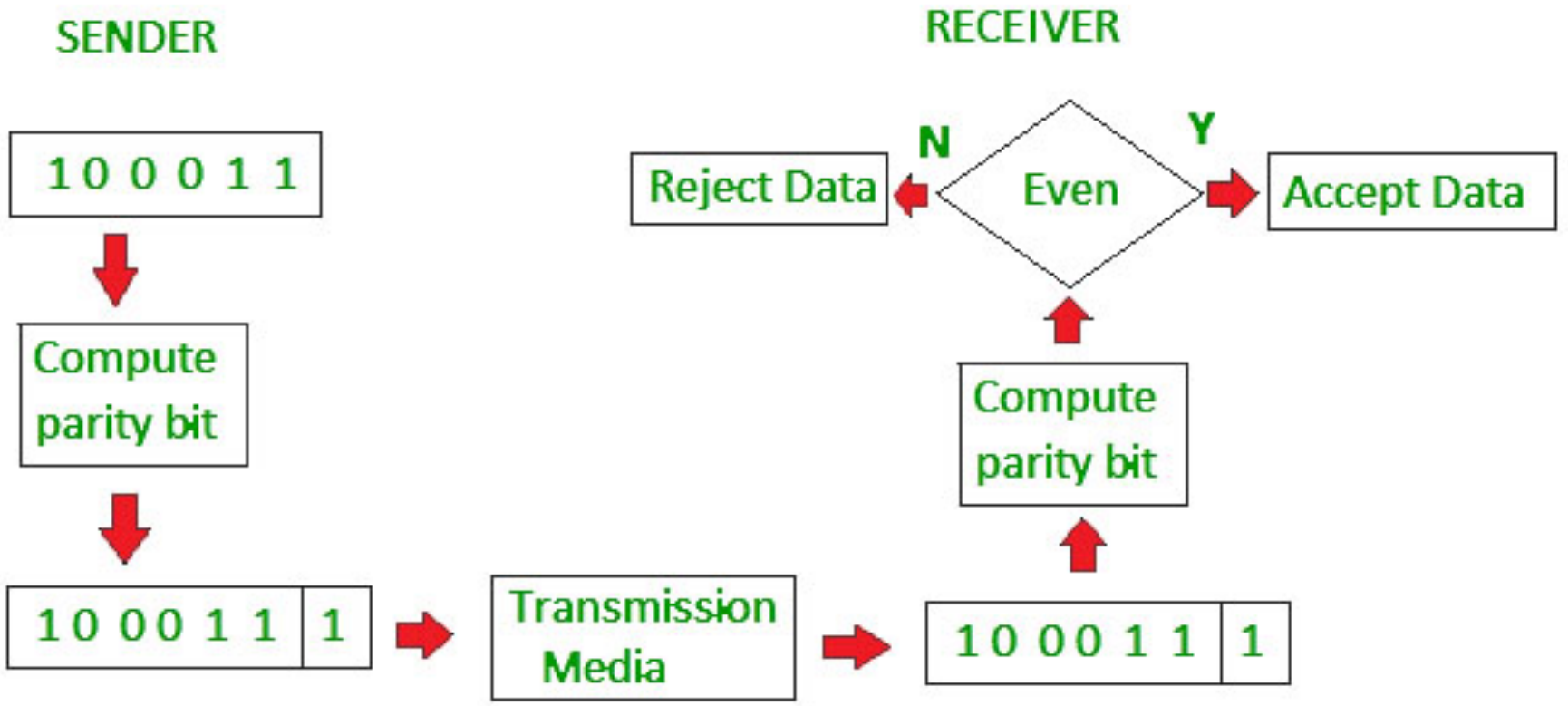
Error detection

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



Parity checking

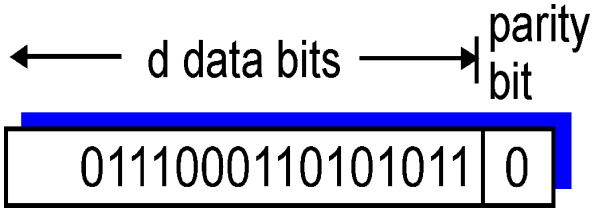


Source: https://www.geeksforgeeks.org/error-detection-computer-networks/

Parity checking

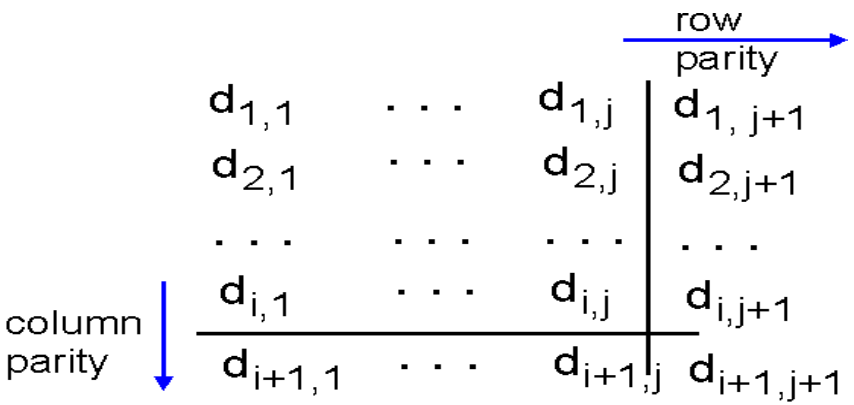
single bit parity:

- *d*etect single bit errors

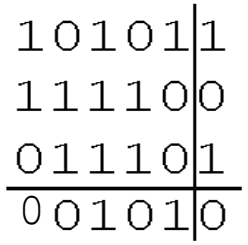


two-dimensional bit parity:

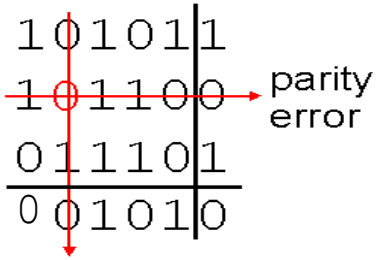
- detect and correct single bit errors



Two-dimensional bit parity helps to identify 2, 3 bit errors too, however cannot be corrected.



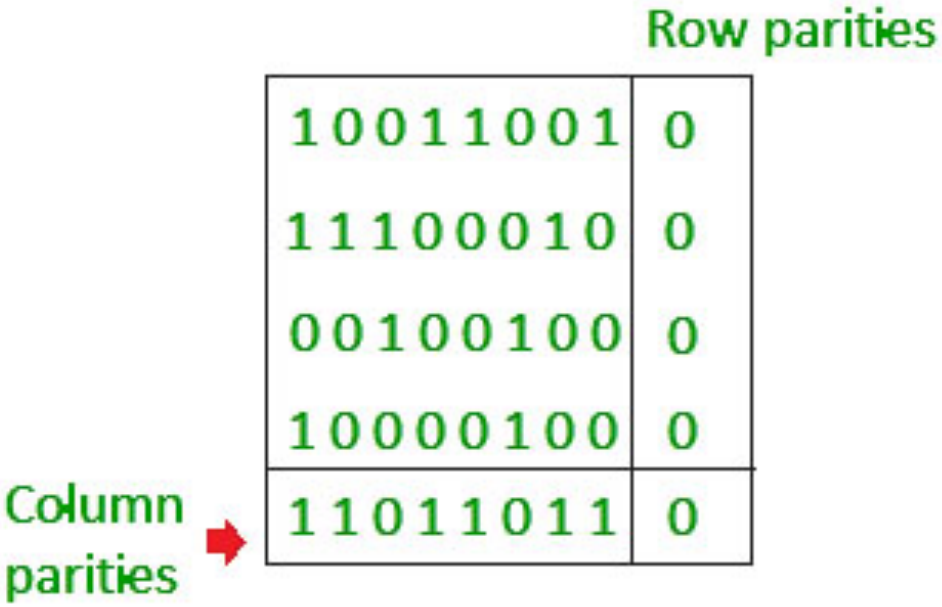
no errors



correctable single bit error

Original Data

10011001	11100010	00100100	10000100



100110010	111000100	001001000	100001000	110110110

Data to be sent

Source: https://www.geeksforgeeks.org/error-detection-computer-networks/

Cheksumming methods

- In checksum error detection scheme, the data is divided into k segments each of m bits.
- In the sender's end the segments are added using 1's complement arithmetic to get the sum. The sum is
- complemented to get the checksum.
- The checksum segment is sent along with the data
- segments.
- At the receiver's end, all received segments are added
- using 1's complement arithmetic to get the sum. The sum is complemented.
- If the result is zero, the received data is accepted; otherwise discarded.

Internet checksum (review)

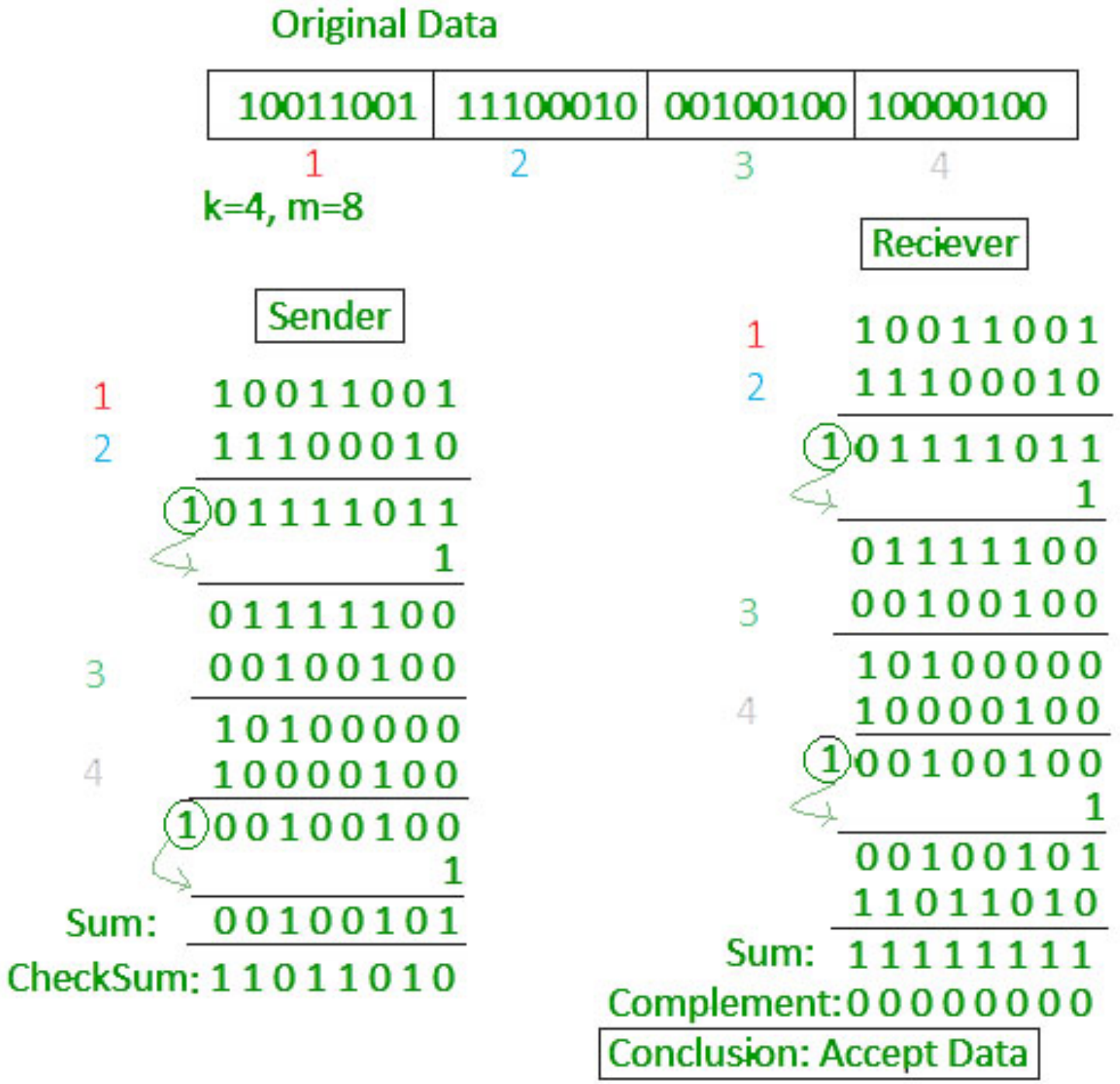
goal: detect "errors" (e.g., flipped bits) in transmitted packet (note: used at transport layer *only*)

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:

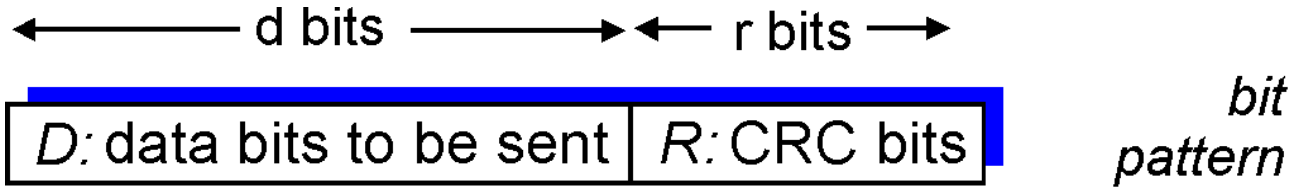
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?*



Source: https://www.geeksforgeeks.org/error-detection-computer-networks/

Cyclic redundancy check

- more powerful error-detection coding
- view data bits, D, as a binary number
- choose r+1 bit pattern (generator), G
- goal: choose r CRC bits, R, such that
  - <D,R> exactly divisible by G (modulo 2)
  - receiver knows G, divides <D,R> by G. If non-zero remainder: error detected!
  - can detect all burst errors less than r+1 bits
- widely used in practice (Ethernet, 802.11 WiFi, ATM)



$D * 2^r \quad XOR \quad R$ *mathematical formula*

CRC example

want:

$D \cdot 2^r$ XOR R = nG

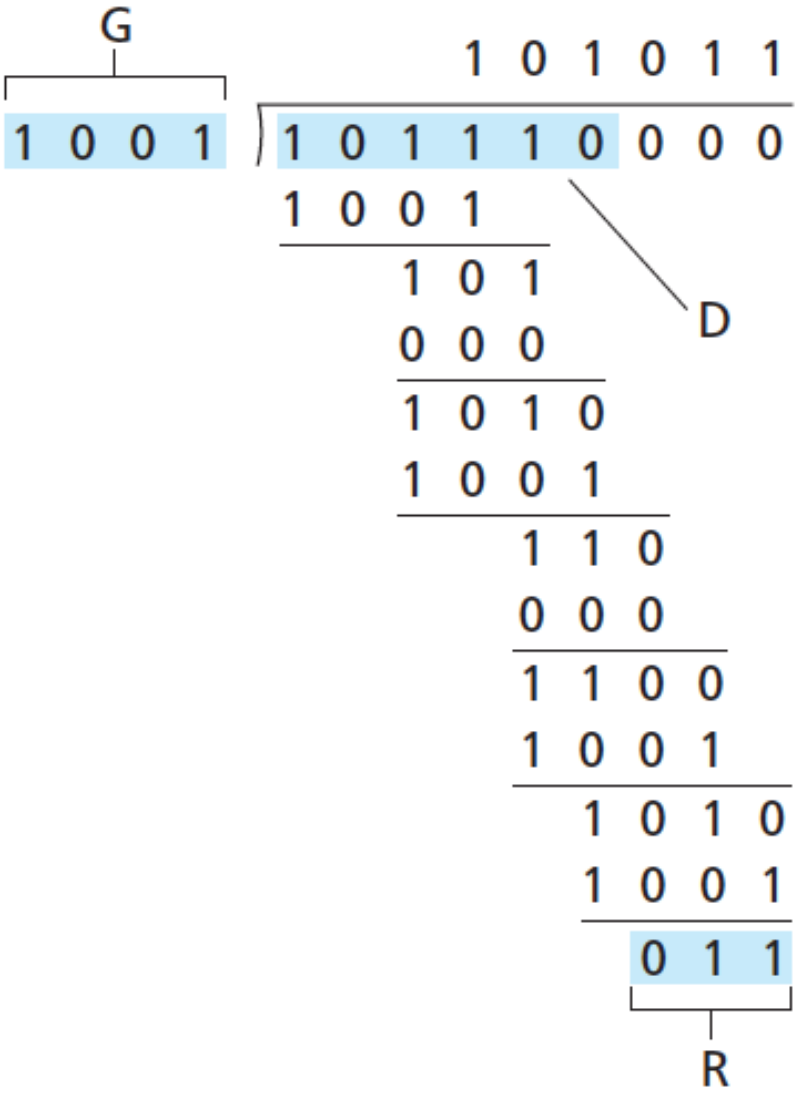
equivalently:

$D \cdot 2^r$ = nG XOR R

equivalently:

if we divide $D \cdot 2^r$ by G, want remainder R to satisfy:

$$R = remainder\left[\frac{D \cdot 2^r}{G}\right]$$



original message

1 0 1 0 0 0 0

@ means X-OR

Generator polynomial

x^3+1

$1.x^3+0.x^2+0.x^1+1.x^0$

CRC generator

1 0 0 1 4-bit

If CRC generator is of n bit then append (n-1) zeros in the end of original message

Sender

```
1001 | 1010000000
     @1001
      ----
      0011000000
       @1001
        ----
        0101000
         @1001
          ----
          0011000
           @1001
            ----
            01010
             @1001
              ----
              0011
```

Message to be transmitted

```
1010000000
      +0 1 1
----------
1010000011
----------
```

```
1001 | 1010000011
     @1001
      ----
      0011000011
       @1001
        ----
        0101 0011
         @1001
          ----
          0011011
           @1001
            ----
            01001
             @1001
              ----
              0000
```

Receiver

Zero means data is accepted

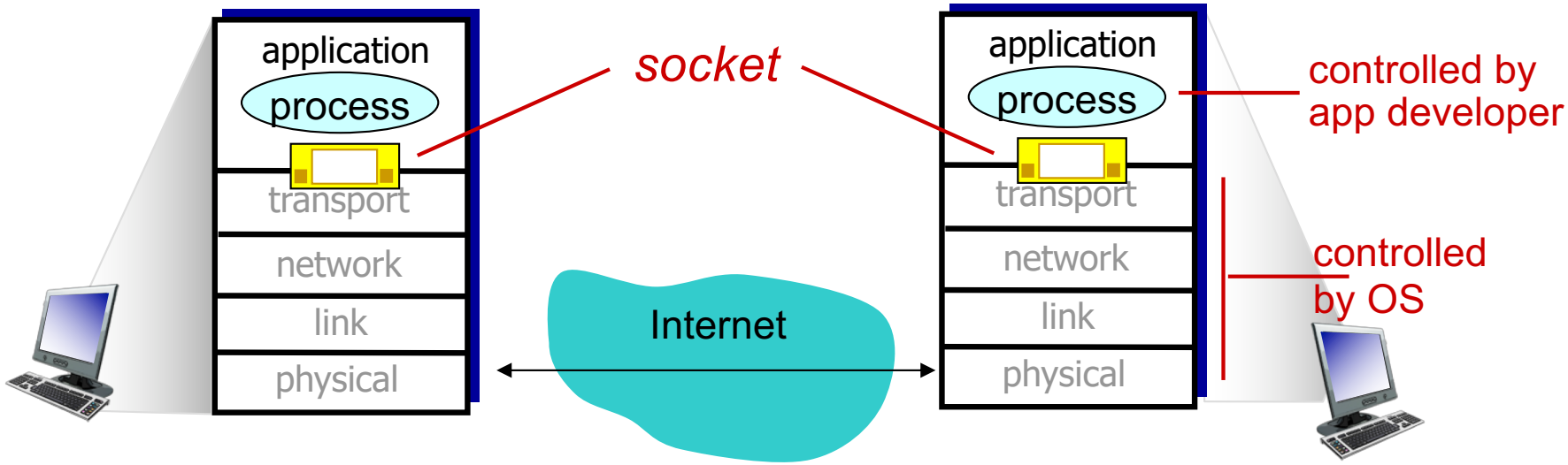
Network Application Programming

Sections 2.1.2, 2.7

Computer Networking – A top-down approach, Kurose and Ross, 6th Edition.

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

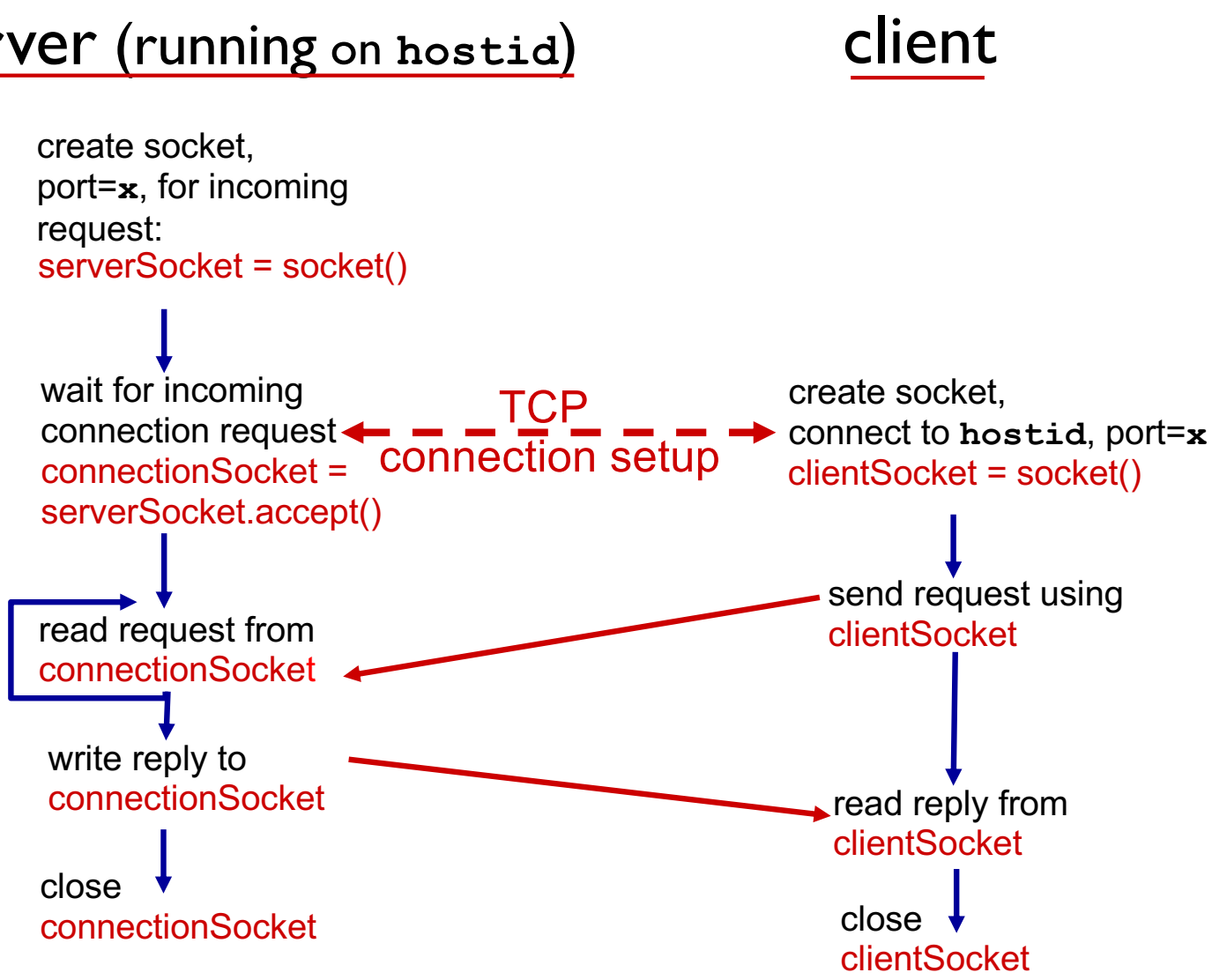
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
 from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)