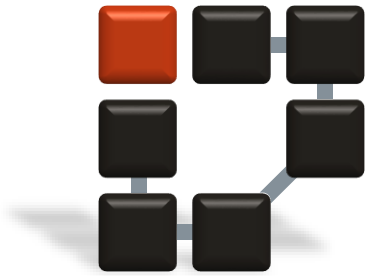
Socket Programming

Dr. -Ing. Abdalkarim Awad



Informatik 7
Rechnernetze und
Kommunikationssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Before we start

- Can you find the ip address of an interface?
- Can you find the mac address of an interface?

- What is the IP address of the linux machine?
- What is the IP address of the windows OS?

- What is the MAC address of the linux machine?
- What is the MAC address of the windows OS?

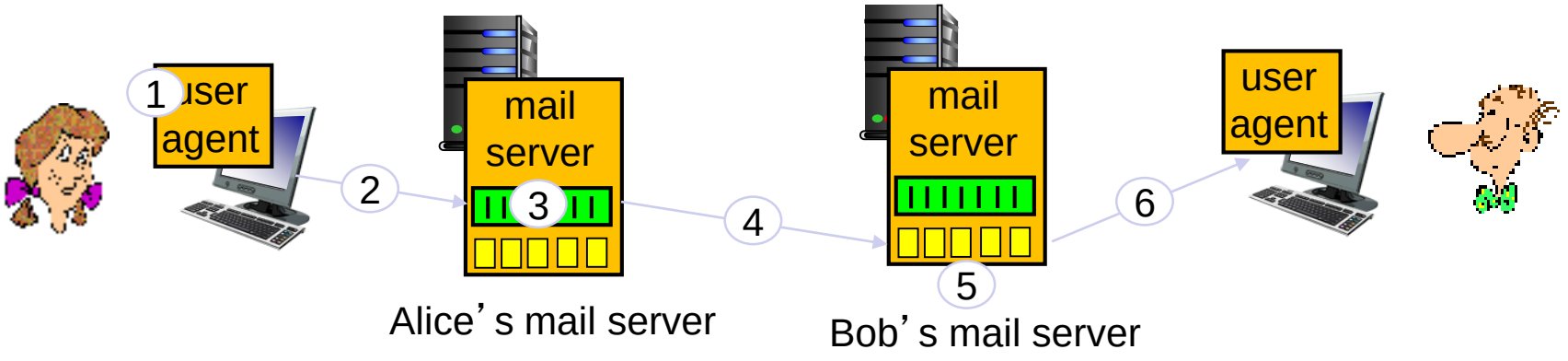
Email spoofing

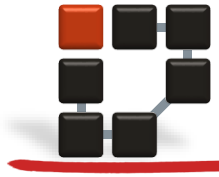
Electronic Mail: SMTP [RFC 2821]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCI

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "to" `bob@someschool.edu`
2) Alice's UA sends message to her mail server; message placed in message queue
3) client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message





Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

Try SMTP interaction for yourself:

- **`telnet servername 25`**
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

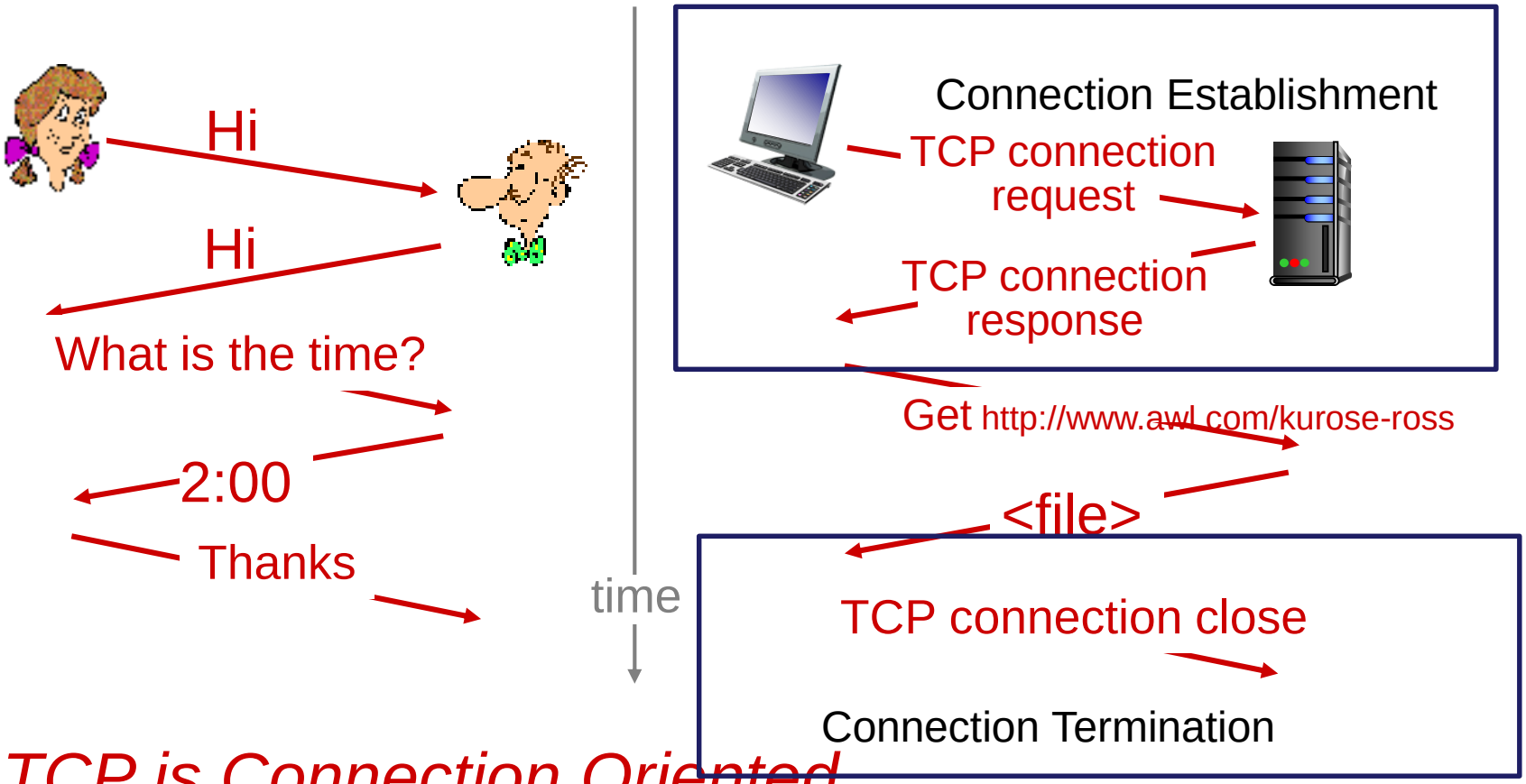
comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg

Socket Programming in Windows
TCP (Transmission Control Protocol)
UDP (User Datagram Protocol)

What's a protocol?
Ex. Transmission Control Protocol(TCP)?

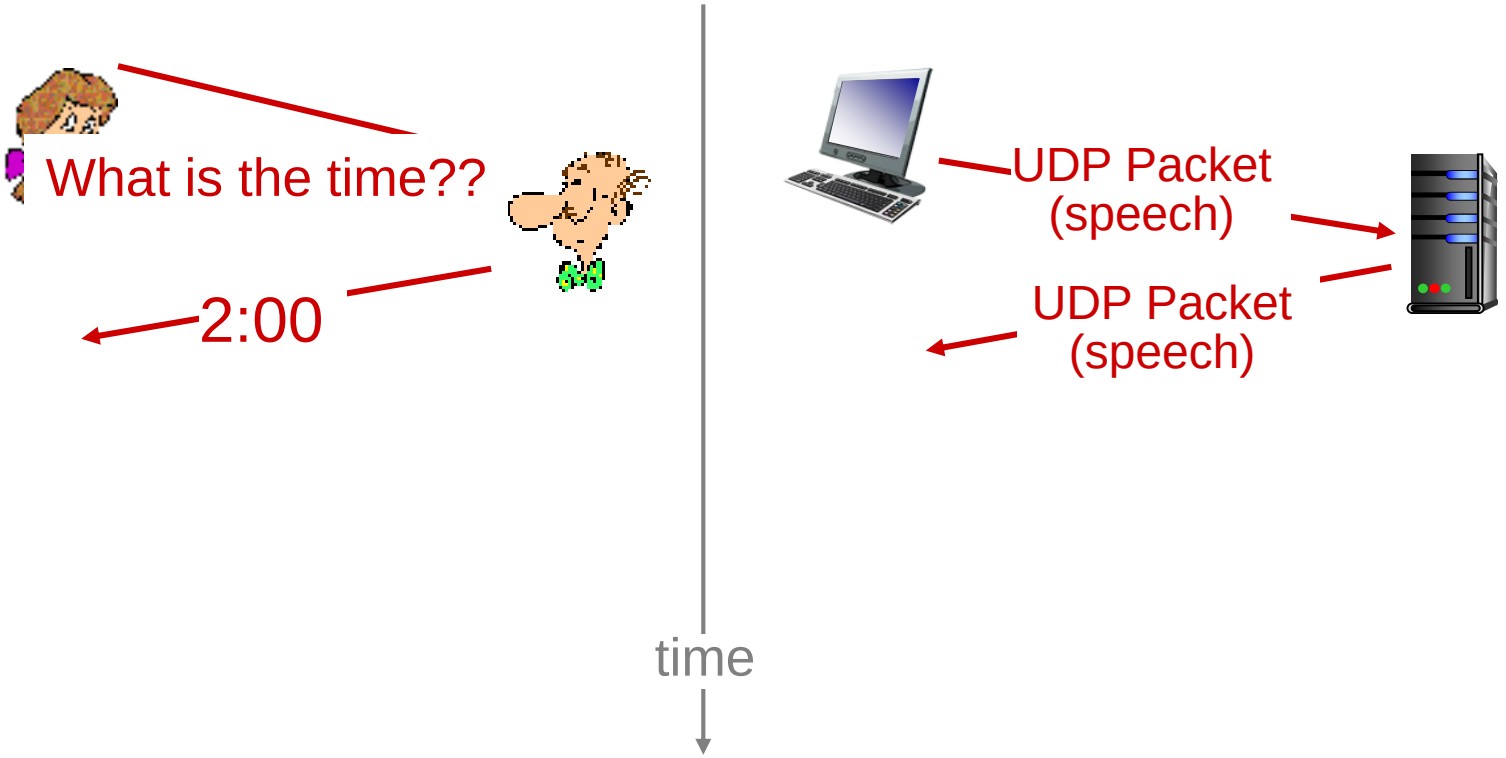
a human protocol and a computer network protocol:



TCP is Connection Oriented Protocol

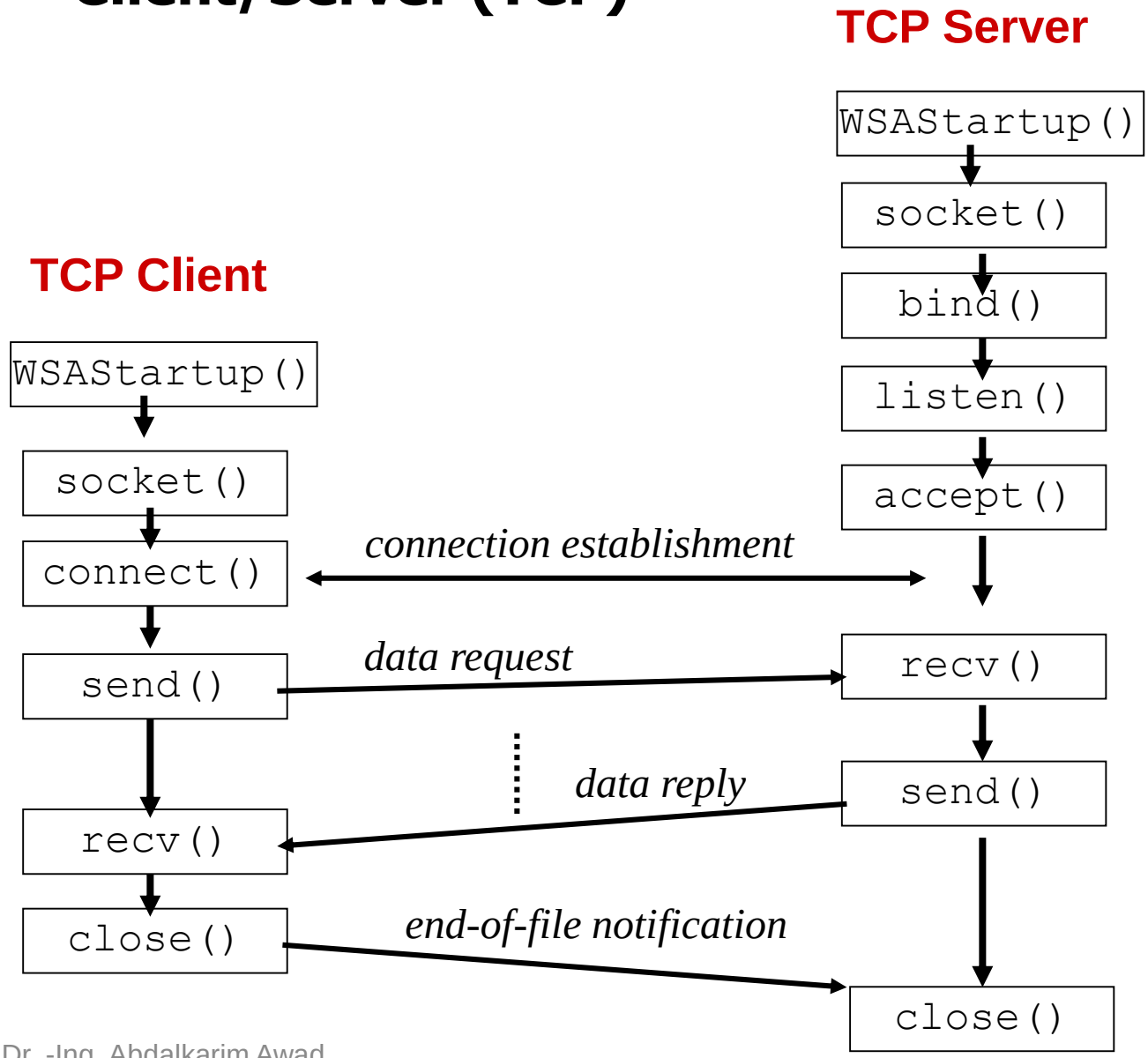
What's a protocol?
Ex. User Datagram Protocol (UDP)

a human protocol and a computer network protocol:

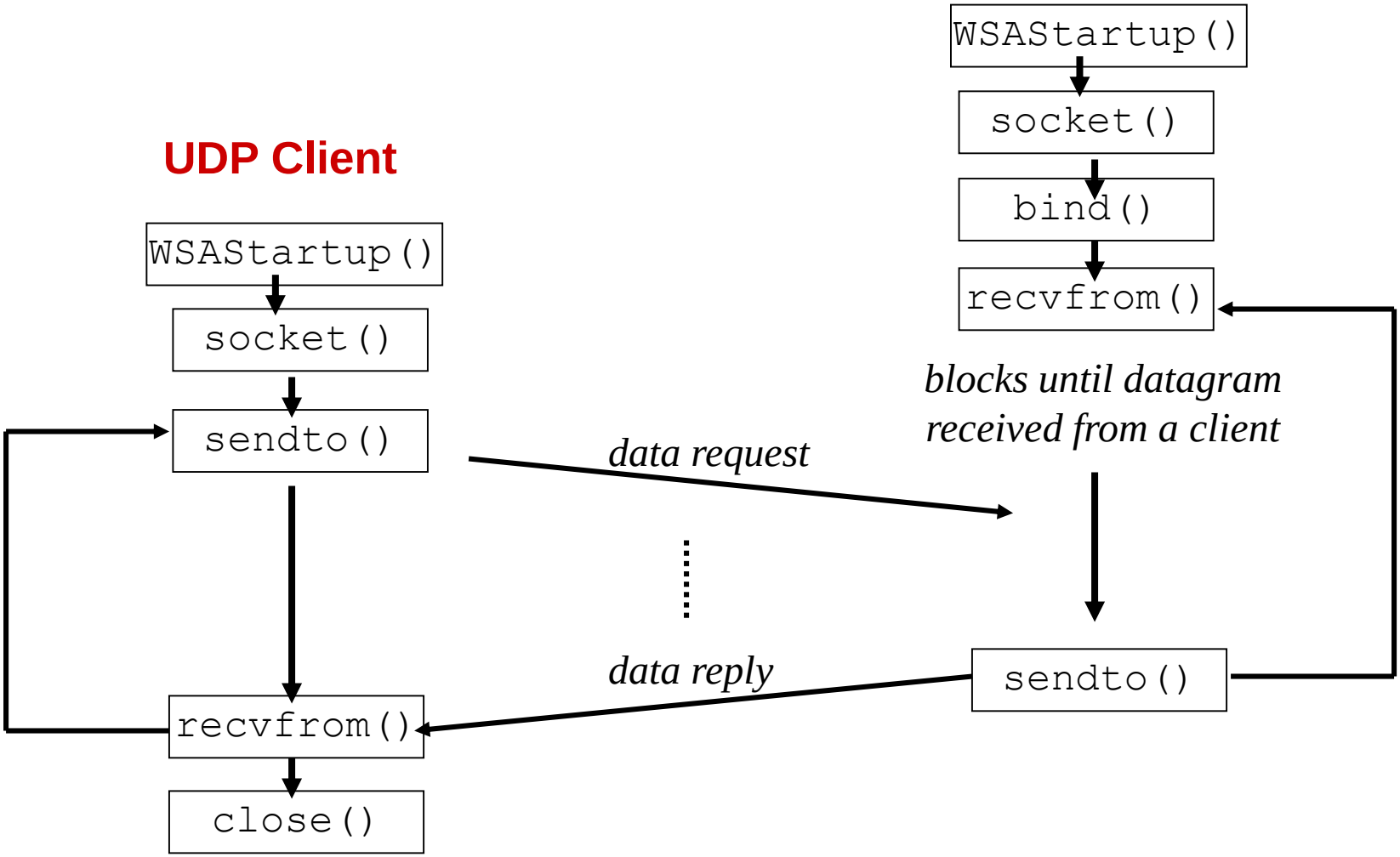


UDP is Connectionless Protocol

Client/Server (TCP)



Client/Server (UDP)

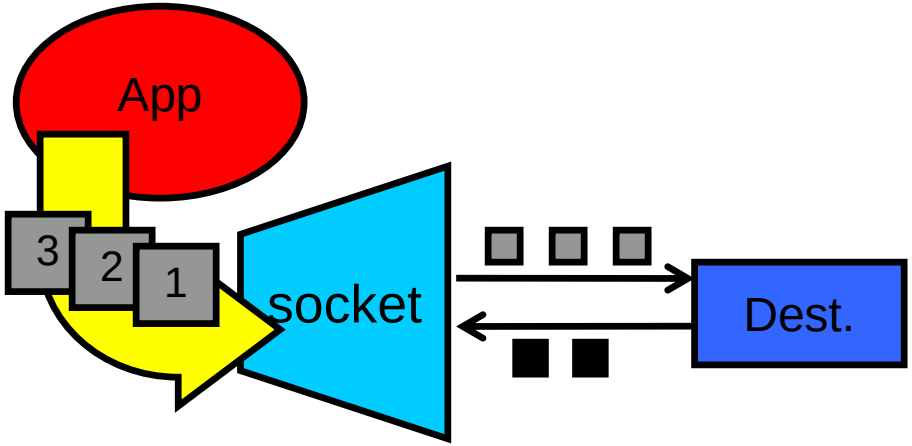


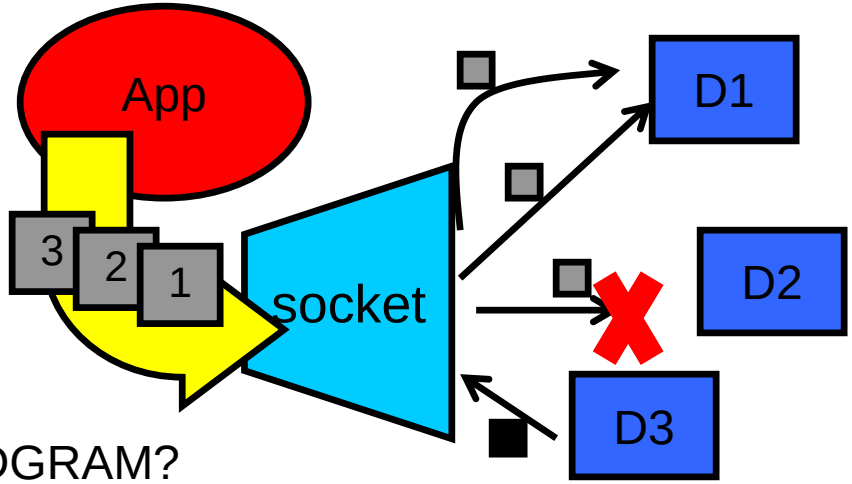
What is a socket?

- An interface between application and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional
- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" – app indicates dest. for each packet
  - can send or receive





Q: why have type SOCK_DGRAM?

Socket Creation in C: socket

- int s = socket(domain, type, protocol);
  - s: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) – typically used
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - protocol: specifies protocol (IPPROTO_UDP *or* IPPROTO_TCP) - usually set to 0
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

A Socket-eye view of the Internet

- Each host machine has an IP address
- When a packet arrives at a host

medellin.cs.columbia.edu

(128.59.21.14)

newworld.cs.umass.edu

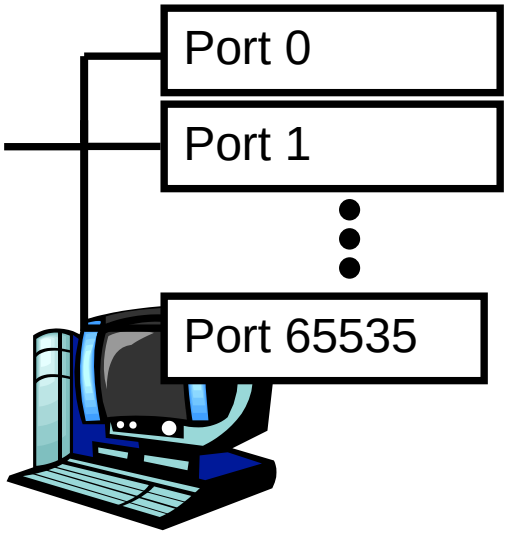
(128.119.245.93)

cluster.cs.columbia.edu

(128.59.21.14, 128.59.16.7, 128.59.16.5, 128.59.16.4)

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



- A socket provides an interface to send data to/from the network through a port

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the door that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)

The bind function

- associates and (can exclusively) reserves a port for use by the socket
- int status = bind(sockid, &addrport, size);
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
  - size: the size (in bytes) of the addrport structure

```
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
```

Skipping the bind

- SOCK_DGRAM:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
  - if receiving, need to bind
- SOCK_STREAM:
  - destination determined during conn. setup
  - don’t need to know port sending from (during connection setup, receiving end is informed of port)

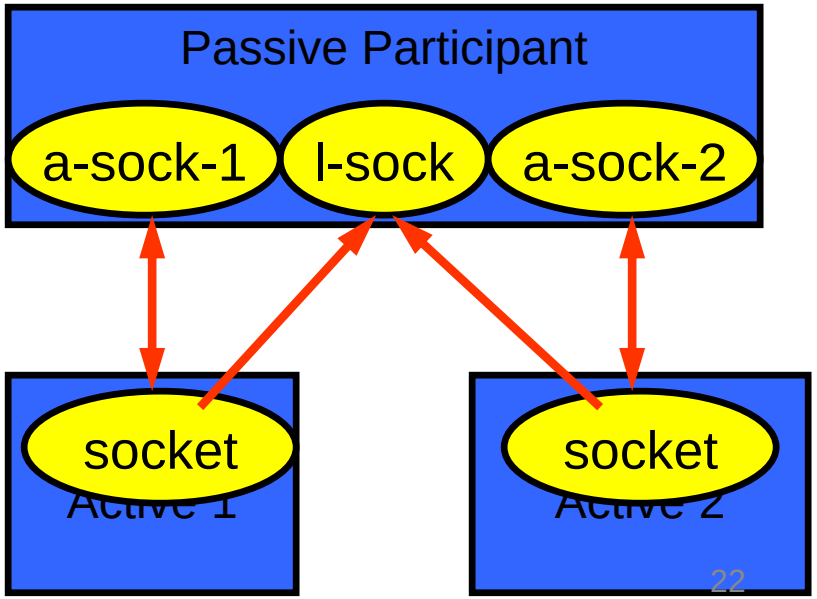
Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are "similar"
  - both can send & receive data
  - either can terminate the connection

Connection setup cont'd

- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants

- Active participant
  - step 2: request & establish connection
  - step 4: data transfer



Connection setup: listen & accept

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately
- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

connect call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- connect is **blocking**

Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

Sending / Receiving Data (cont'd)

- Without a connection (SOCK_DGRAM):
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]

close

- When finished using a socket, the socket should be closed:
- status = close(s);
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

The struct sockaddr

- The generic:

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

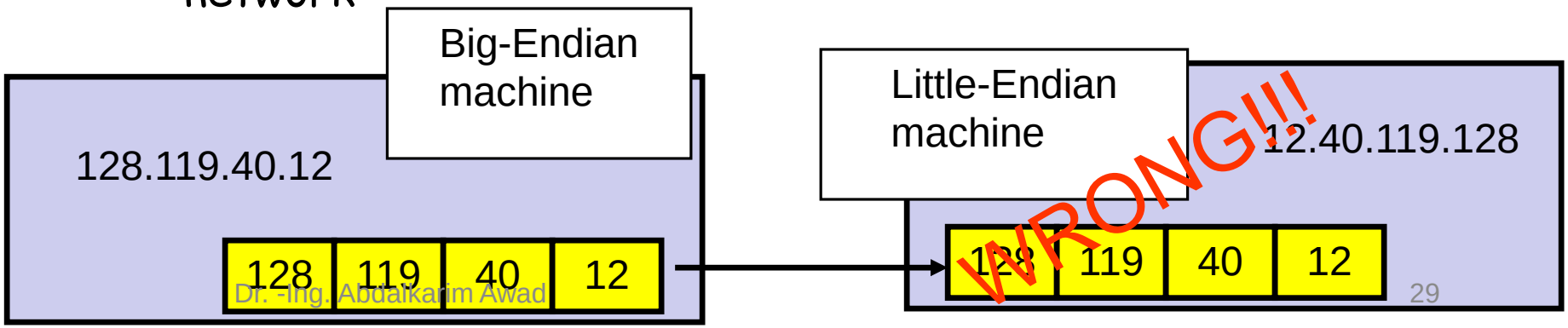
  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused

Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network



Solution: Network Byte-Ordering

- Defs:
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network – always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order