Announcement

- Project 1 has been posted on the course website
  - Due: 4/28

- **Individual project!**
  - Students can discuss each other, but must write their own codes

- Submission
  - All commented source codes
  - Makefile
  - Report (1-3 pages)

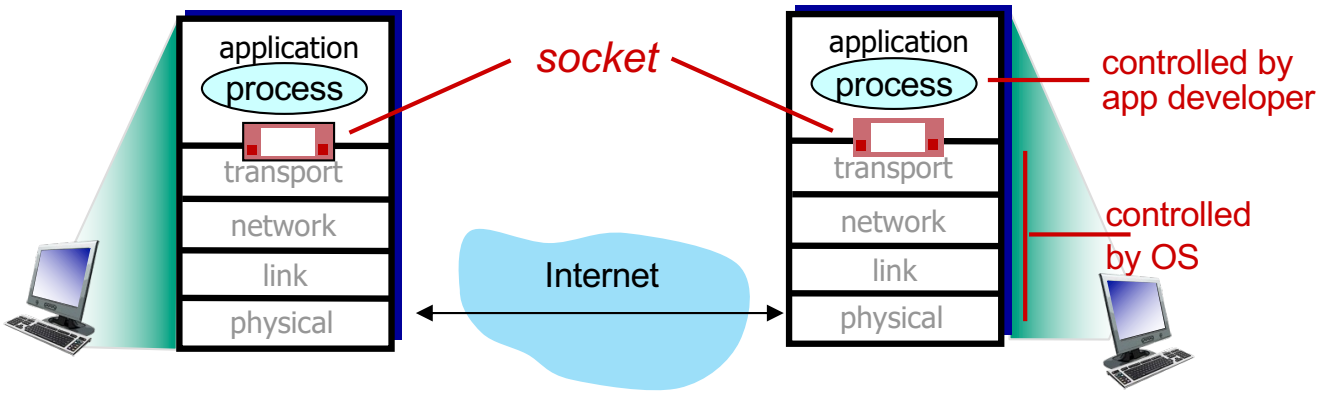
Socket Programming

- What is a socket?
- Using sockets
  - Types (Protocols)
  - Associated functions
  - Styles
- Socket programming tutorial video:
  - https://youtu.be/LtXEMwSG5-8

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

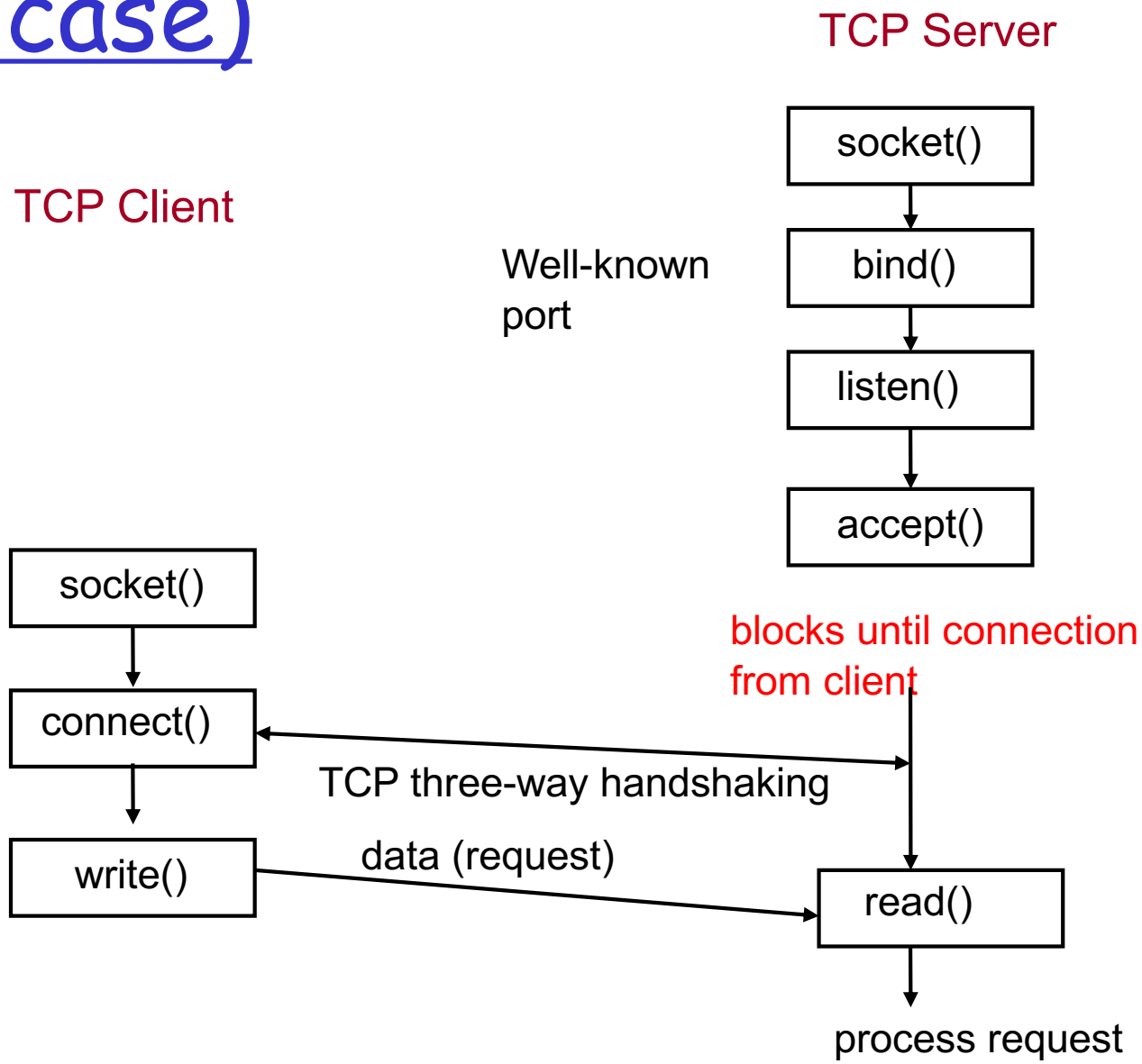
Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

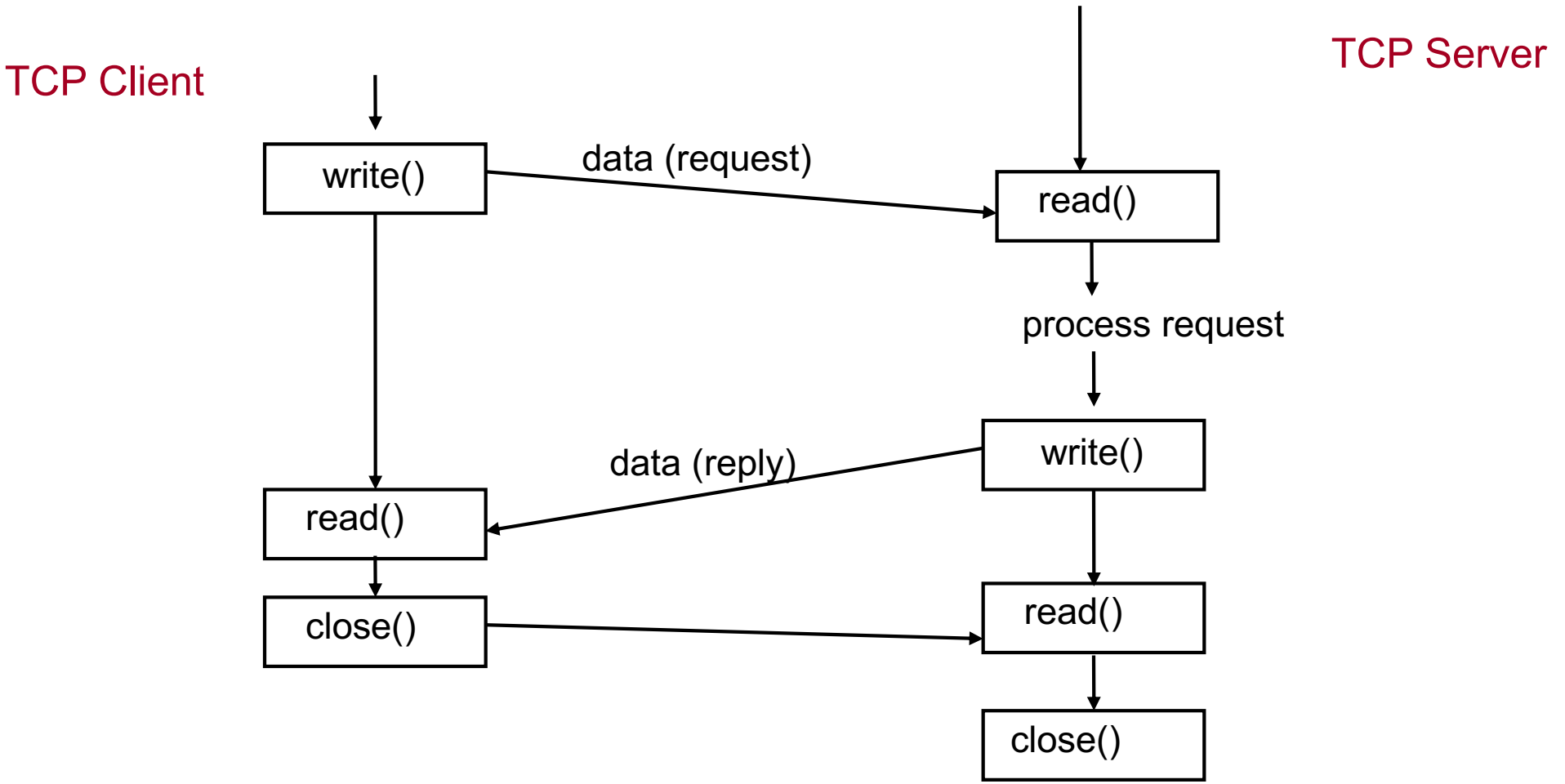
Sockets API

- Creation and Setup
- Establishing a Connection (TCP)
- Sending and Receiving Data
- Tearing Down a Connection (TCP)

Big picture: Socket Functions (TCP case)



Big picture: Socket Functions (TCP case) cont.



7

Sockets API

- Creation and Setup
- Establishing a Connection (TCP)
- Sending and Receiving Data
- Tearing Down a Connection (TCP)

Socket Creation and Setup

- Include file **<sys/socket.h>**
- Create a socket
  - **int socket (int domain, int type, int protocol);**
    - Returns file descriptor or -1.
- Bind a socket to a local IP address and port number
  - **int bind (int sockfd, struct sockaddr* myaddr, int addrlen);**
- Put socket into passive state (wait for connections rather than initiate a connection).
  - **int listen (int sockfd, int backlog);**
- Accept connections
  - **int accept (int sockfd, struct sockaddr* cliaddr, int* addrlen);**
    - Returns file descriptor or -1.

9

Function: socket

```
int socket (int domain, int type, int
  protocol);
```

- Create a socket.
  - Returns file descriptor or -1. Also sets **errno** on failure.
  - **domain**: protocol family (same as address family)
    - **PF_INET** for IPv4 (typicall used)
    - other possibilities: **PF_INET6** (IPv6), **PF_UNIX** or **PF_LOCAL** (Unix socket), **PF_ROUTE** (routing)
  - **type**: style of communication
    - **SOCK_STREAM** for TCP (with **PF_INET**)
    - **SOCK_DGRAM** for UDP (with **PF_INET**)
  - **protocol**: protocol within family
    - Typically set to 0
    - getprotobyname(), /etc/protocols for list of protocols

10

Function: bind

```
int bind (int sockfd, struct sockaddr*
  myaddr, int addrlen);
```

- Bind a socket to a local IP address and port number.
  - Returns 0 on success, -1 and sets **errno** on failure.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **myaddr**: includes IP address and port number
    - IP address: set by kernel if value passed is **INADDR_ANY**, else set by caller
    - port number: set by kernel if value passed is 0, else set by caller
  - **addrlen**: length of address structure
    - **= sizeof (struct sockaddr_in)**

11

Function: listen

```
int listen (int sockfd, int backlog);
```

- Put socket into passive state (wait for connections rather than initiate a connection).
  - Returns 0 on success, -1 and sets **errno** on failure.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **backlog**: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high
- **Listen is non-blocking**: returns immediately

12

Function: accept

```
int accept (int sockfd, struct sockaddr*
  cliaddr, int* addrlen);
```

- Accept a new connection.
  - Returns file descriptor or -1. Also sets **errno** on failure.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **cliaddr**: IP address and port number of client (returned from call)
  - **addrlen**: length of address structure = pointer to **int** set to **sizeof (struct sockaddr_in)**
- **Accept is blocking**
  - Waits for connection before returning

13

Sockets API

- Creation and Setup
- Establishing a Connection (TCP)
- Sending and Receiving Data
- Tearing Down a Connection (TCP)

14

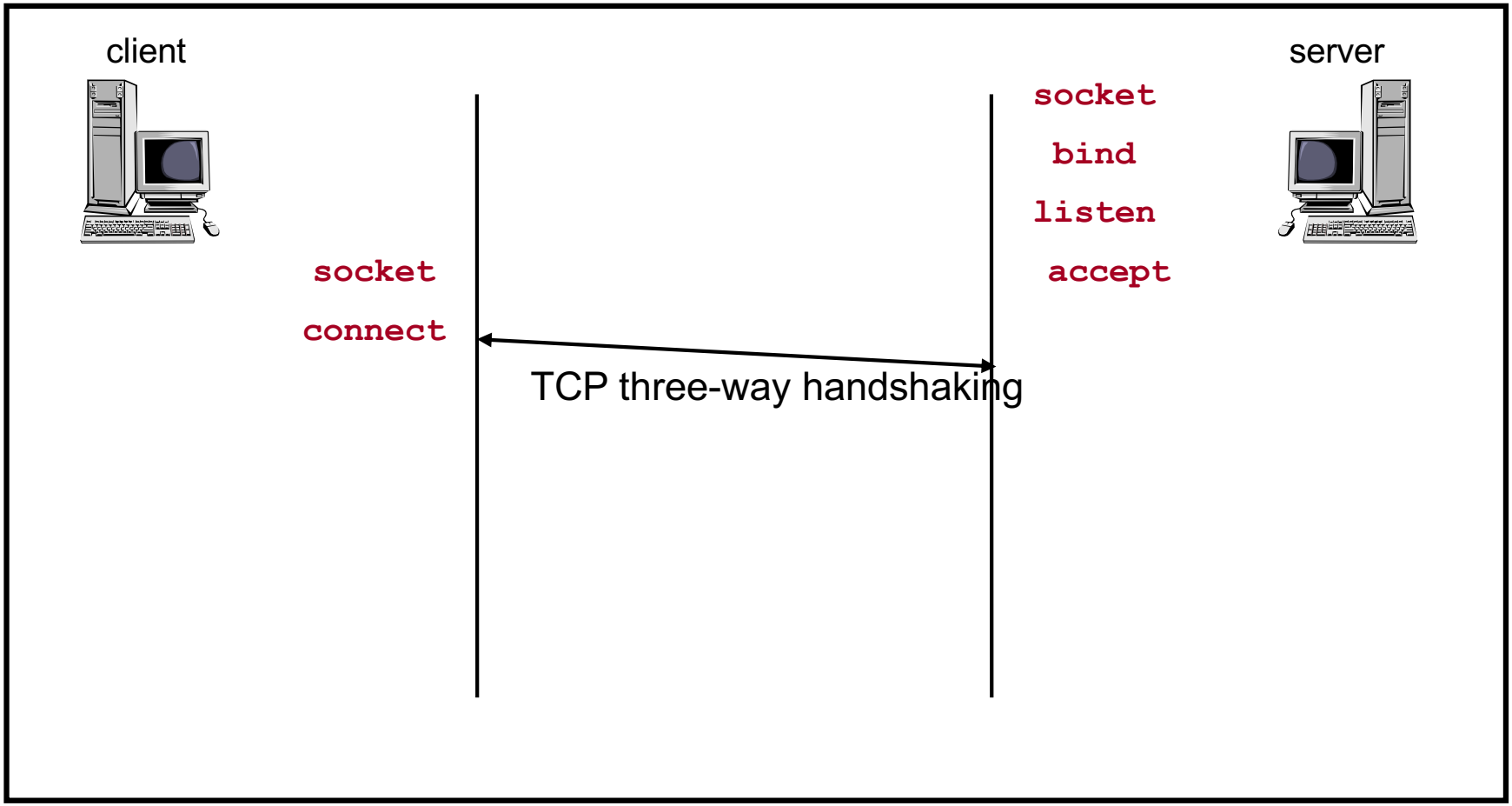
Function: connect

```
int connect (int sockfd, struct sockaddr*
  servaddr, int addrlen);
```

- Connect to another socket.
  - Returns 0 on success, -1 and sets `errno` on failure.
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `servaddr`: IP address and port number of ***server***
  - `addrlen`: length of address structure
    - `= sizeof (struct sockaddr_in)`
- Connect is **blocking**

15

Recap: TCP socket connection setup



16

Sample code: server

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#define PORT 3490
#define BACKLOG 10          /* how many pending
                               connections queue
                               will hold */
```


17

server

```
main()
{
   int sockfd, new_fd;      /* listen on sock_fd, new
                                  connection on new_fd
   */
   struct sockaddr_in my_addr;    /* my address */
   struct sockaddr_in their_addr; /* connector addr */
   int sin_size;

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))==-1){
       perror("socket");
       exit(1);
   }
```


18

server

- The Internet-specific:

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        };
```

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

```
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT); /* short, network
                                     byte order */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* INADDR_ANY allows clients to connect to any one of
the host's IP address */


if (bind(sockfd, (struct sockaddr *)&my_addr,
         sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```


19

server

```
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}
while(1) { /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr*)
                    &their_addr,&sin_size)) == -1) {
        perror("accept");
        continue;
    }
    printf("server: got connection from %s\n",
                        inet_ntoa(their_addr.sin_addr));
}
```


20

client

```
if ((sockfd = socket (PF_INET, SOCK_STREAM, 0)) == -1) {
     perror ("socket");
     exit (1);
}

their_addr.sin_family = AF_INET;
their_addr.sin_port = htons (Server_Portnumber);
their_addr.sin_addr = htonl(Server_IP_address);


if (connect (sockfd, (struct sockaddr*)&their_addr,
                sizeof (struct sockaddr)) == -1) {
     perror ("connect");
     exit (1);
}
```


21

Sockets API

- Creation and Setup
- Establishing a Connection (TCP)
- Sending and Receiving Data
- Tearing Down a Connection (TCP)

22

Functions: write

```
int write (int sockfd, char* buf, size_t
  nbytes);
```

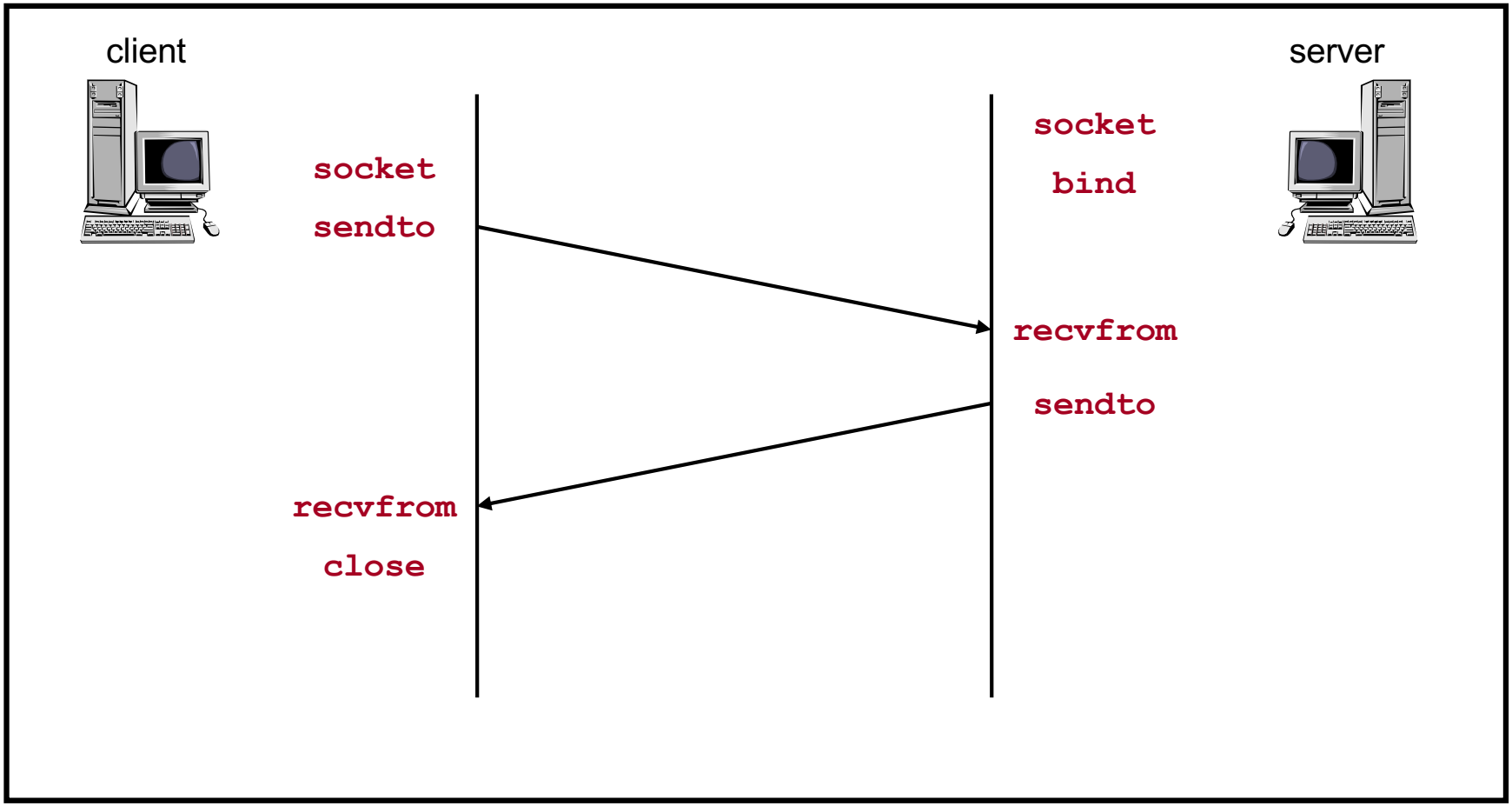
- Write data to a stream (TCP).
  - Returns number of bytes written or -1. Also sets **errno** on failure.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to write
- write is **blocking**; returns only after data is sent

Functions: read

```
int read (int sockfd, char* buf, size_t
  nbytes);
```

- Read data from a stream (TCP).
  - Returns number of bytes read or -1. Also sets **errno** on failure.
  - Returns 0 if socket closed.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
- read is **blocking**; returns only after data is received

Big picture: UDP Socket Functions



25

Functions: sendto

```
int sendto (int sockfd, char* buf, size_t nbytes,
   int flags, struct sockaddr* destaddr, int addrlen);
```

- Send a datagram to UDP socket.
  - Returns number of bytes written or -1. Also sets **errno** on failure.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
  - **flags**: see man page for details; typically use 0
  - **destaddr**: IP address and port number of destination socket
  - **addrlen**: length of address structure
    - = **sizeof (struct sockaddr_in)**
- sendto is **blocking**; returns only after data is sent

26

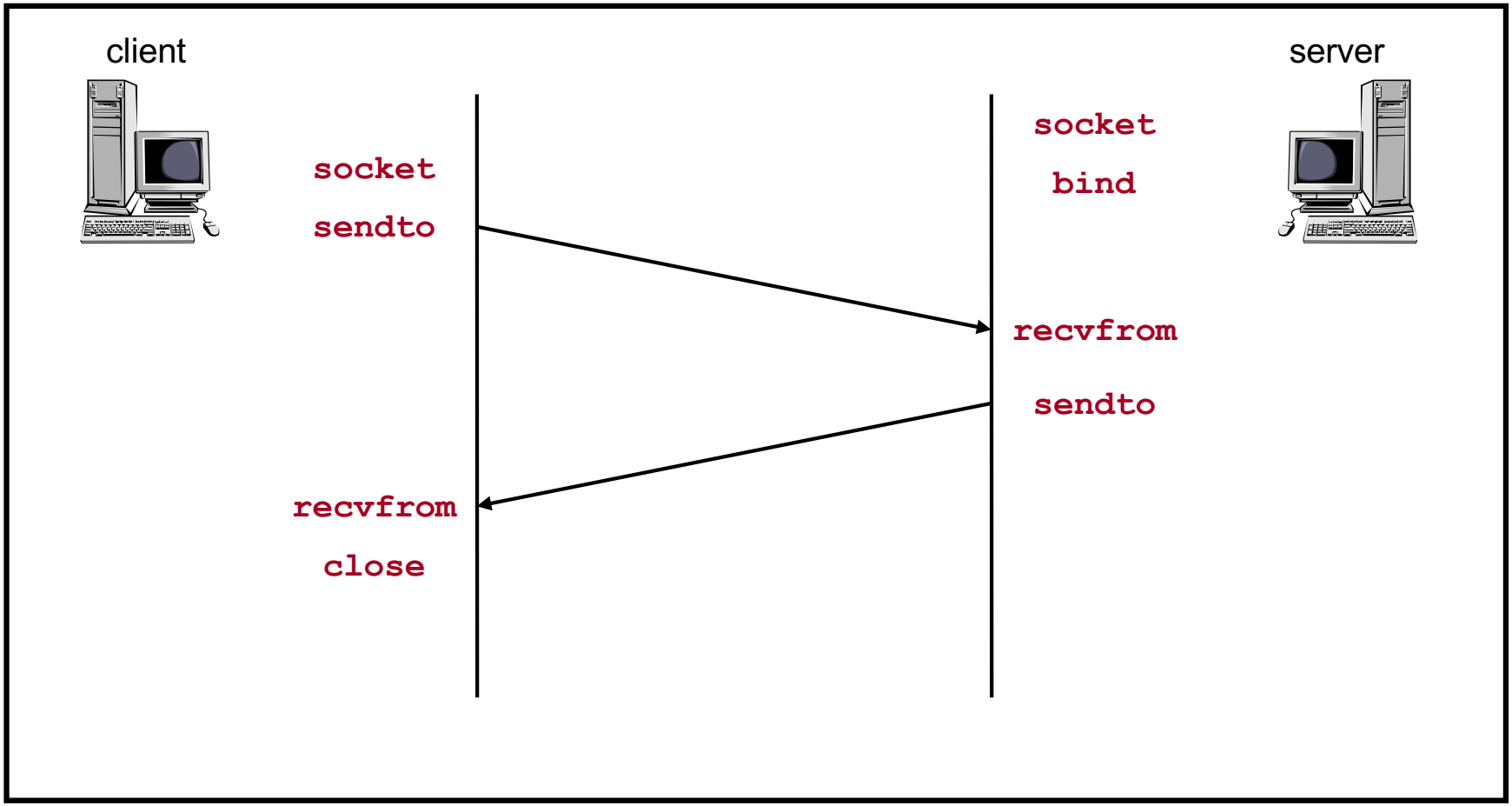
Function: recvfrom

```
int recvfrom (int sockfd, char* buf, size_t nbytes,
   int flags, struct sockaddr* srcaddr, int* addrlen);
```

- Read a datagram from a UDP socket.
  - Returns number of bytes read (0 is valid) or -1. Also sets **errno** on failure.
  - **sockfd**: socket file descriptor (returned from **socket**)
  - **buf**: data buffer
  - **nbytes**: number of bytes to try to read
  - **flags**: see man page for details; typically use 0
  - **srcaddr**: IP address and port number of sending socket (returned from call)
  - **addrlen**: length of address structure = pointer to **int** set to **sizeof (struct sockaddr_in)**
- recvfrom is **blocking**; returns only after data is received

27

Recap: UDP socket functions



Sockets API

- Creation and Setup
- Establishing a Connection (TCP)
- Sending and Receiving Data
- Tearing Down a Connection (TCP)

29

Function: close

```
int close (int sockfd);
```

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

Tip: Release of ports

- Sometimes, a "rough" exit from a program (e.g., ctrl-c) does not properly free up a port
- Eventually (after a few minutes), the port will be freed
- To reduce the likelihood of this problem, include the following code:

```
#include <signal.h>
void cleanExit(){exit(0);}
```

  - in socket code:

```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

Project 1: Web server

- Part A:
  - Web server simply dumps HTTP request messages to the console.

- Part B:
  - Based on Part A, the Web server:
    1. parses the HTTP request from the browser
    2. Creates an HTTP response message containing the requested file preceded by header lines
    3. Sends the response directly to the client (i.e., browser)

Some more useful information for socket programming…

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];
};
```

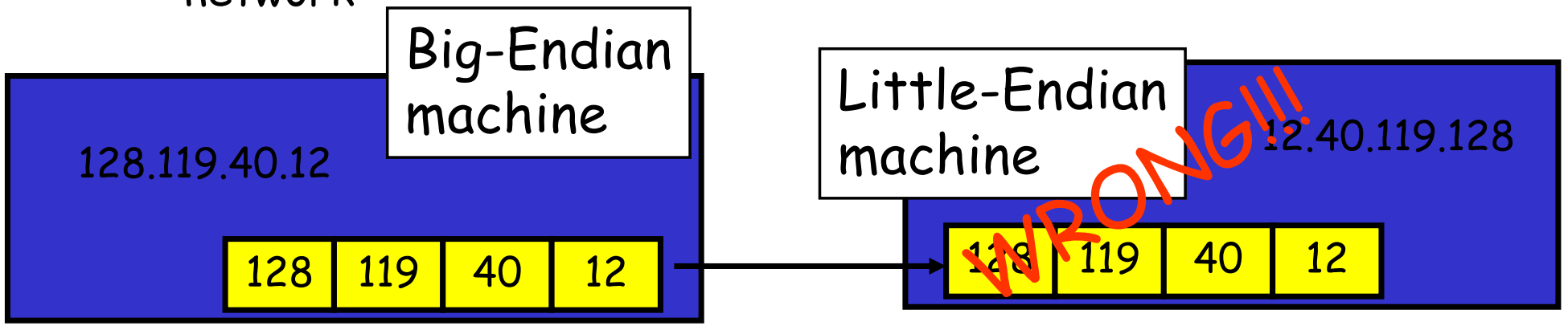
  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused

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

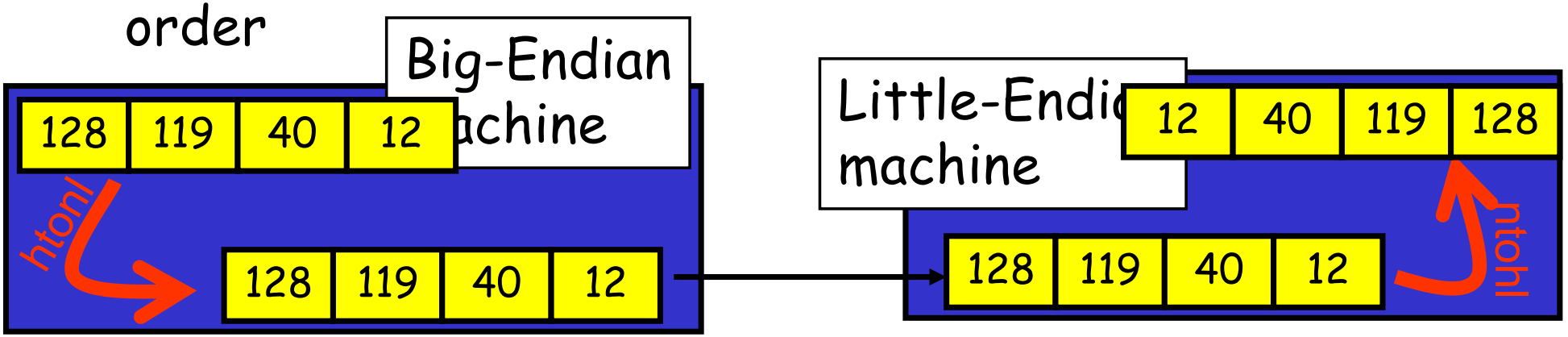
- Define:
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - **Network Byte-Ordering**: the byte ordering used by the network – **always big-endian**
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order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

- Q: Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?

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



- Same code would have worked regardless of endian-ness of the two machines

Other useful functions

- bzero(char* c, int n): 0's n bytes starting at c
- gethostname(char *name, int len): gets the name of the current host
- gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer
- inet_addr(const char *cp): converts dotted-decimal char-string to long integer
- inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order

Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Dealing w/ blocking (cont'd)

- Options:
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select function call.
- What does select do?
  - can be permanent blocking, time-limited blocking or non-blocking
  - input: a set of file-descriptors
  - output: info on the file-descriptors' status
  - i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately

Function: select

- int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
  - status: # of ready objects, -1 if error
  - nfds: 1 + largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - timeout: time after which select returns, even if nothing ready - can be 0 or ∞
    (point timeout parameter to NULL for ∞)

To be used with select:

- Recall select uses a structure, struct fd_set
  - it is just a bit-vector
  - if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select:
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to check file desc. *i*
- After calling select:
  - int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is "ready"