

Data Communication: Socket Programming

Dr. Yingwu Zhu

Socket Programming

- ❑ What is a socket?
- ❑ Using sockets
 - Types (Protocols)
 - Associated functions
 - Styles
 - We will look at using sockets in C

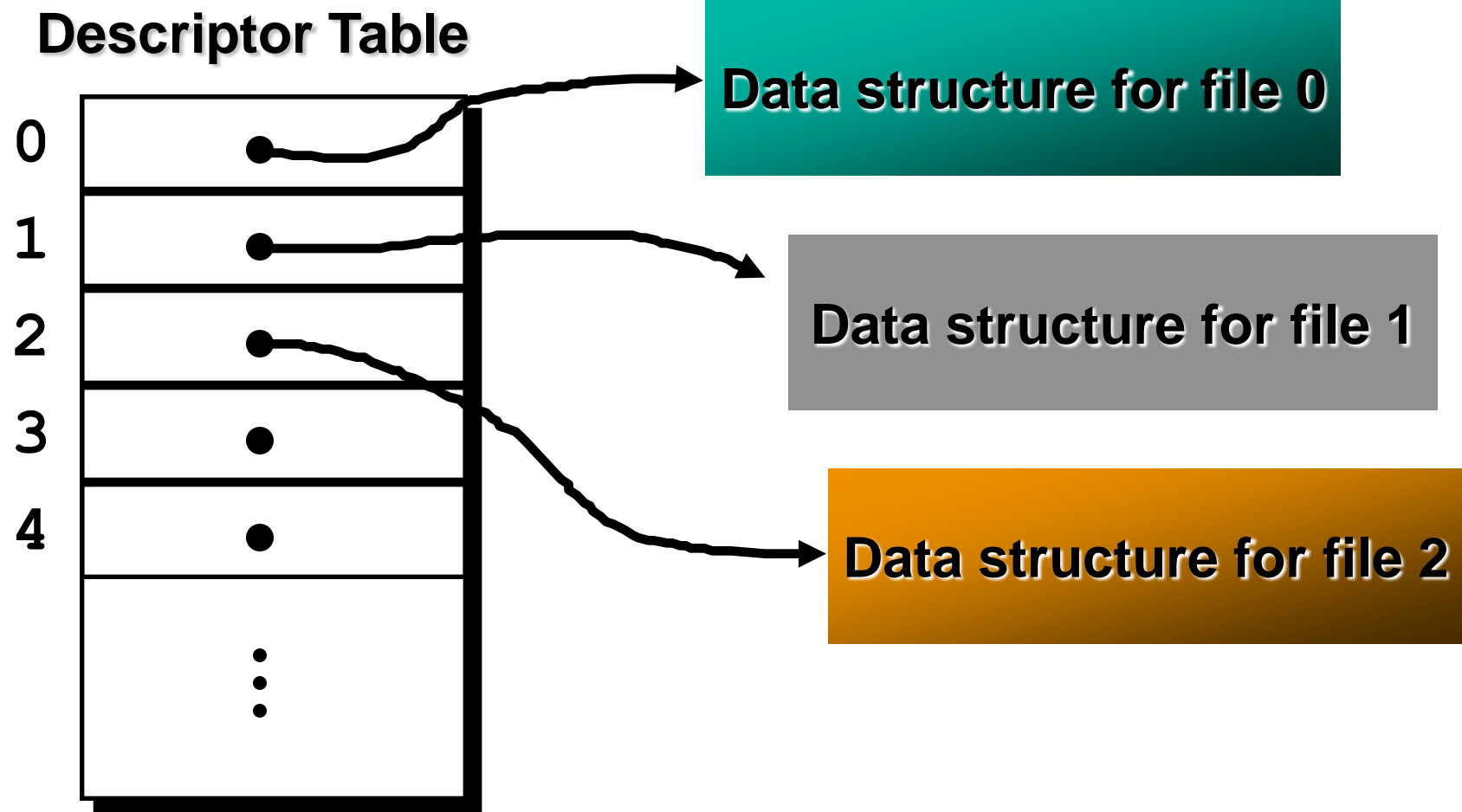
What is a socket?

- ❑ 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)
 - Use it like a file descriptor for reads/writes

Socket

- ❑ A socket is an abstract representation of a communication endpoint.
- ❑ Sockets work with Unix I/O services just like files, pipes & FIFOs.
 - Treat me as a file, please!
- ❑ Sockets (obviously) have special needs:
 - establishing a connection
 - specifying communication endpoint addresses

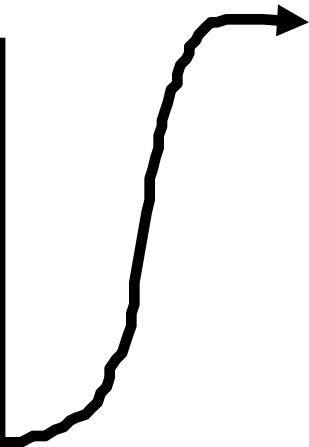
Unix Descriptor Table



Socket Descriptor Data Structure

Descriptor Table

0	•
1	•
2	•
3	•
4	•
	⋮

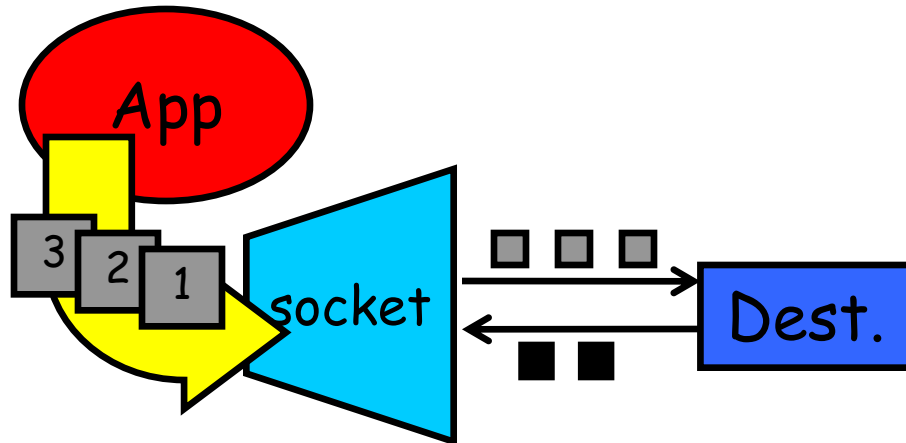


Family: PF_INET
Service: SOCK_STREAM
Local IP: 111.22.3.4
Remote IP: 123.45.6.78
Local Port: 2249
Remote Port: 3726

Two essential types of sockets

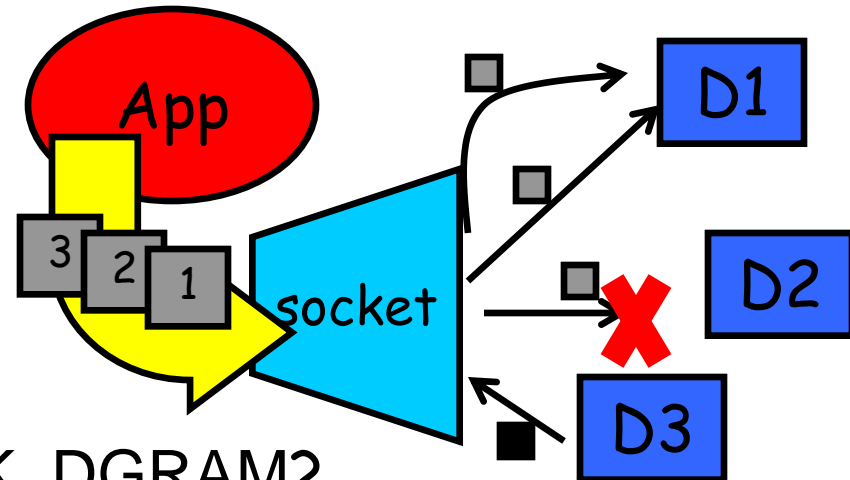
❑ SOCK_STREAM

- 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

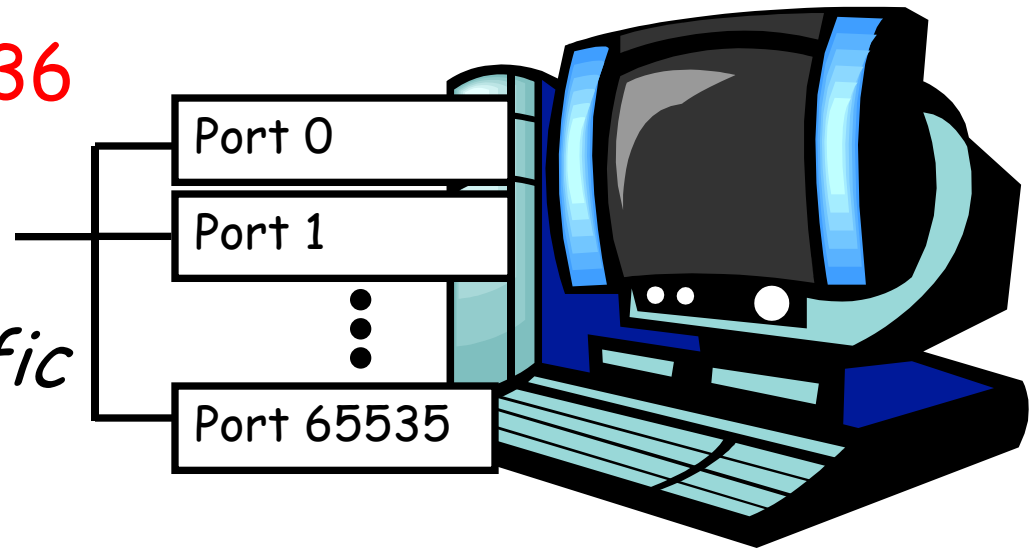
- ❑ `int s = socket(domain, type, protocol);`
 - `s`: socket descriptor, an integer (like a file-handle)
 - `domain`: integer, communication domain
 - e.g., `PF_INET` (IPv4 protocol) - typically used
 - Now in Linux: `#define PF_INET AF_INET` (value of 2)
 - `type`: communication type
 - `SOCK_STREAM`: reliable, 2-way, connection-based service
 - `SOCK_DGRAM`: unreliable, connectionless,
 - other values: need root permission, rarely used, or obsolete
 - `protocol`: specifies protocol (see file `/etc/protocols` for a list of options) - 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!

socket ()

- ❑ The `socket ()` system call returns a socket descriptor (small integer) or `-1` on error.
- ❑ `socket ()` allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.

Ports

- ❑ Each host has 65,536 ports (limited!)
- ❑ 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 key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)


- ❑ Q: How do you choose which port a socket connects to?

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 can be skipped for both types of sockets.
When and why?

Assigning an address to a socket

- ❑ The `bind()` system call is used to assign an address to an existing socket.

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

- ❑ `bind` returns 0 if successful or -1 on error.

bind()

- ❑ calling `bind()` assigns the address specified by the `sockaddr` structure to the socket descriptor.
- ❑ You can give `bind()` a `sockaddr_in` structure:

```
bind( mysock,  
      (struct sockaddr*) &myaddr,  
      sizeof(myaddr) );
```

bind() Example

```
int mysock,err;  
struct sockaddr_in myaddr;  
  
mysock = socket(PF_INET,SOCK_STREAM,0);  
myaddr.sin_family = AF_INET;  
myaddr.sin_port = htons( portnum );  
myaddr.sin_addr = htonl( ipaddress);  
  
err=bind(mysock, (sockaddr *) &myaddr,  
        sizeof(myaddr));
```

Uses for bind()

- ❑ There are a number of uses for `bind()`:
 - Server would like to bind to a well known address (port number).
 - Client can bind to a specific port.
 - Client can ask the O.S. to assign *any available* port number.

Port schmort - who cares ?

- ❑ Clients typically don't care what port they are assigned.
- ❑ When you call bind you can tell it to assign you any available port:

```
myaddr.port = htons(0);
```

What is my IP address ?

- ❑ How can you find out what your IP address is so you can tell `bind()` ?
- ❑ There is no realistic way for you to know the right IP address to give `bind()` - what if the computer has multiple network interfaces?
- ❑ specify the IP address as: `INADDR_ANY`, this tells the OS to take care of things.

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 connection setup
- don't need to know port sending from (during connection setup, receiving end is informed of port)

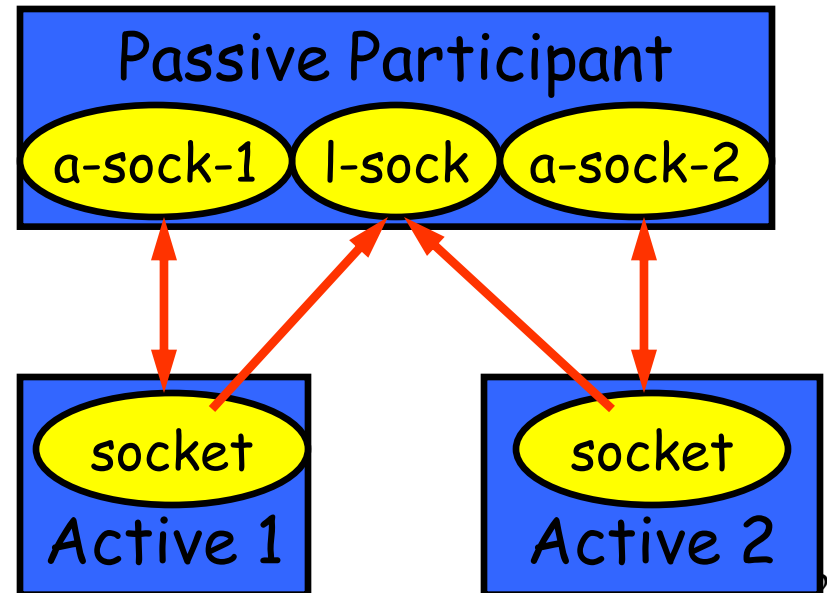
Connection Setup (SOCK_STREAM)

- ❑ 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
- ❑ Why?

- ❑ 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 = write(sock, &buf, len);`

- `count`: # bytes transmitted

- 0: The connection was closed by the remote host.
 - -1: The read system call was interrupted, or failed for some reason.
 - n: The write system call wrote 'n' bytes into the socket..

- `buf`: `char*`, buffer to be transmitted

- `len`: integer, length of buffer (in bytes) to transmit

- `int count = read(sock, &buf, len);`

- `count`: # bytes received (-1 if error)

- 0: The connection was closed by the remote host.
 - -1: The read system call was interrupted, or failed for some reason.
 - n: The read system call put 'n' bytes into the buffer we supplied it with.

- `buf`: `char*`, stores received bytes

- `len`: integer, length of buffer (in bytes) to receive

- Calls are **blocking** [returns only after data is sent (to socket buffer) / received]

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`: integer, length of buffer (in bytes) to receive
 - `flags`: integer, special options, usually just 0
 - Calls are **blocking** [returns only after data is sent (to socket buffer) / 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 buffer) / 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

TCP/IP Addresses

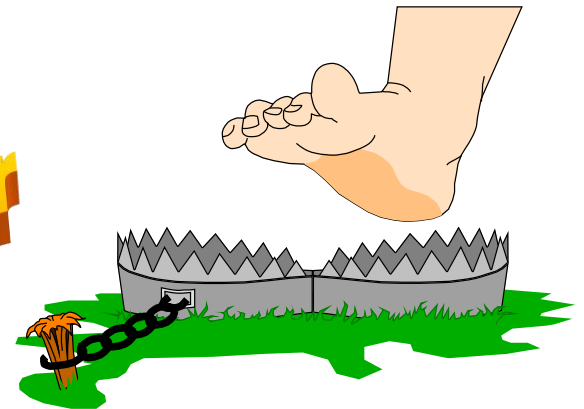
- ❑ We don't need to deal with **sockaddr** structures since we will only deal with a real protocol family.
- ❑ We can use **sockaddr_in** structures.

BUT: The C functions that make up the sockets API expect structures of type **sockaddr**.

Network Byte Order

- ❑ All values stored in a `sockaddr_in` must be in network byte order.
 - `sin_port` a TCP/IP port number.
 - `sin_addr` an IP address.

**Common Mistake:
Ignoring Network Byte Order**



Address and port byte-ordering

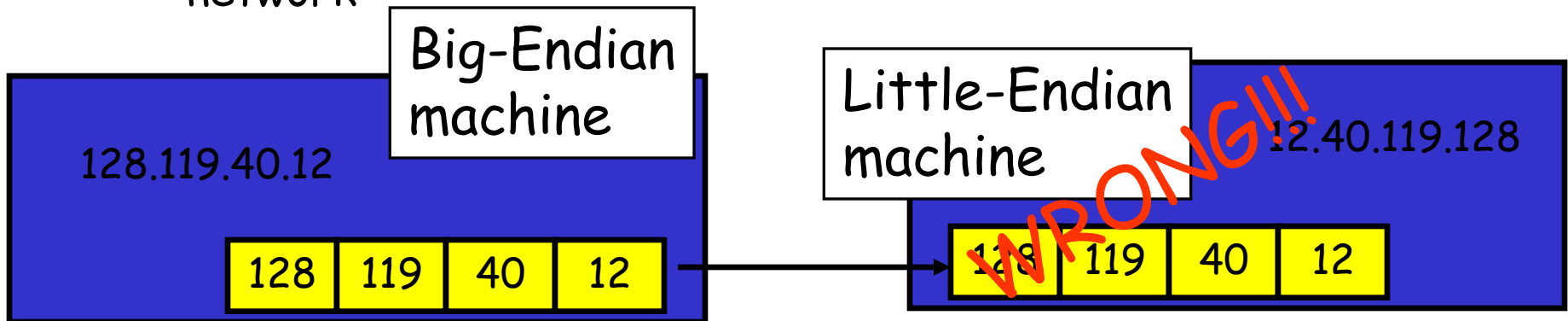
- Address and port are stored as integers

- 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

❑ Definitions:

- 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)

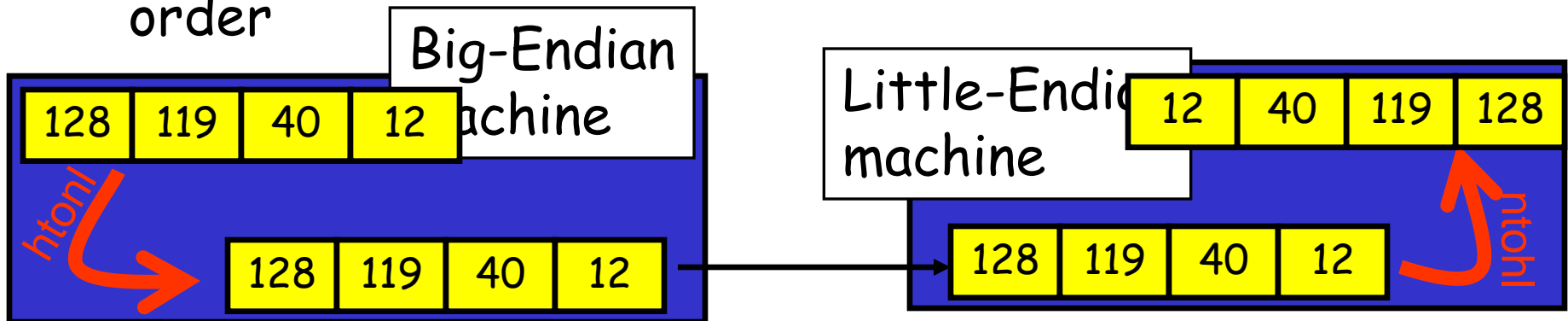
❑ Q: should the socket perform the conversion automatically?

❑ 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_long ntohl(u_long x);`
- ❑ `u_short htons(u_short 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 endianness of the two machines

Address Resolution

□ `struct hostent *gethostbyname(char *hostname);`

- `struct hostent {
 char* h_name; /* official name of host */
 char** h_aliases; /* alias list */
 int h_addrtype; /* host address type */
 int h_length; /* length of address */
 char** h_addr_list; /* list of addresses from name server */
 #define h_addr h_addr_list[0]
 /* address, for backward compatibility */
};`

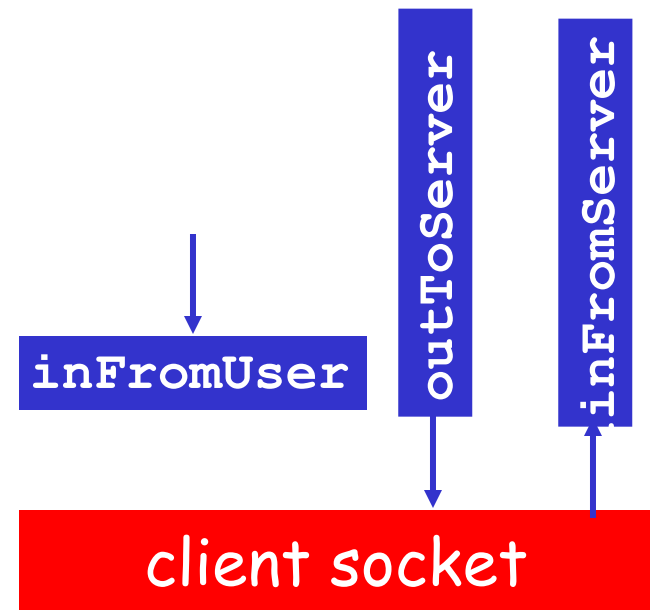
Socket programming with TCP

Example client-server app:

- ❑ client reads line from standard input, sends to server via socket; server reads line from socket
- ❑ server converts line to uppercase, sends back to client
- ❑ client reads, prints modified line from socket

Input stream: sequence of bytes into process

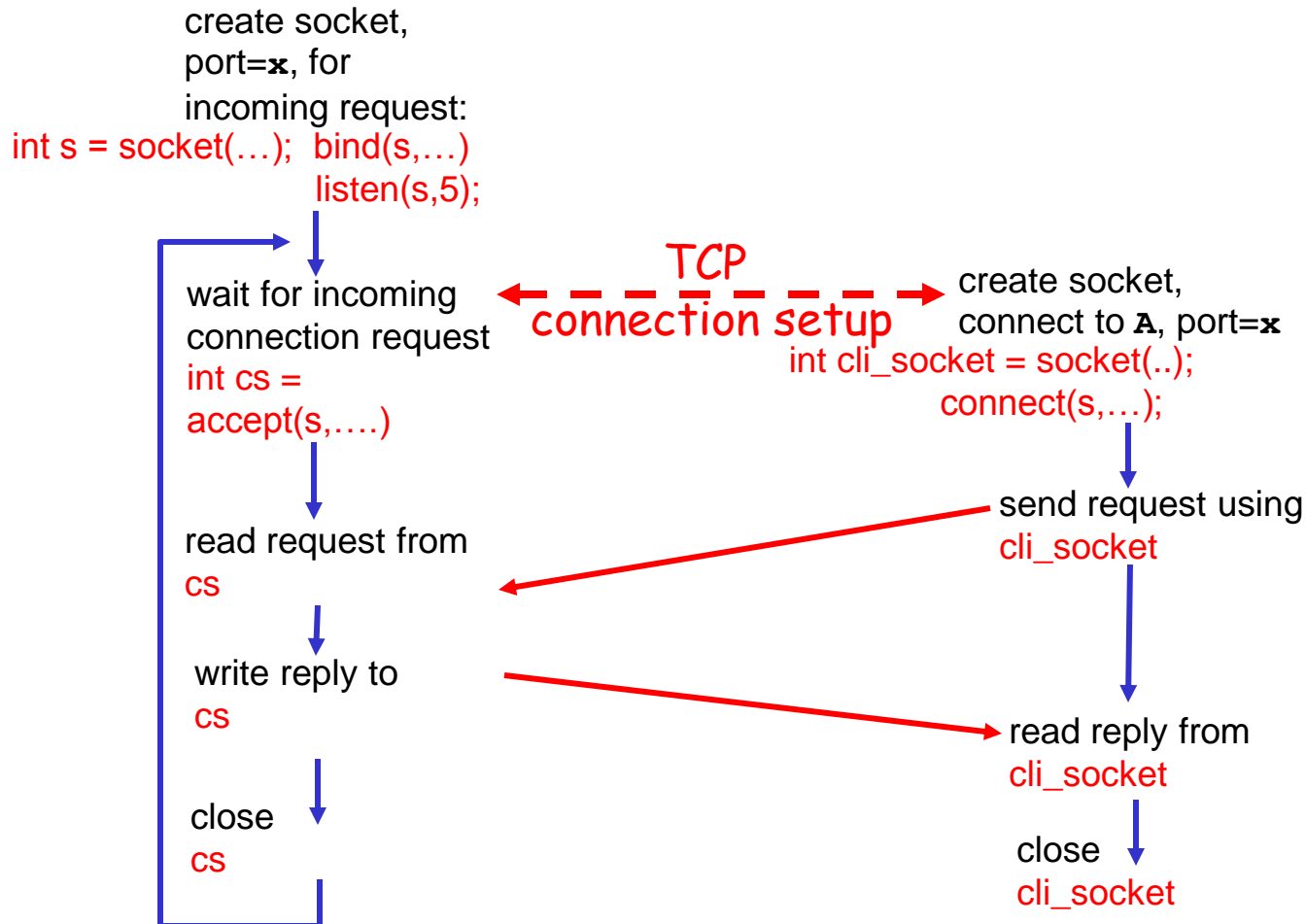
Output stream: sequence of bytes out of process



Client/server socket interaction: TCP

Server (running in A)

Client



Example: C++ client (TCP)

```
#include <stdio.h> /* Basic I/O routines */
#include <sys/types.h> /* standard system types */
#include <netinet/in.h> /* Internet address structures */
#include <sys/socket.h> /* socket interface functions */
#include <netdb.h> /* host to IP resolution */

int main(int argc, char *argv[]) {
    /* Address resolution stage */
    struct hostent* hen = gethostbyname(argv[1]);
    if (!hen) {
        perror("couldn't resolve host name");
    }
    struct sockaddr_in sa;
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(PORT); //server port number
    memcpy(&sa.sin_addr.s_addr, hen->h_addr_list[0], hen->h_length);

    int cli_socket = socket(AF_INET, SOCK_STREAM, 0);
    assert(cli_socket >= 0); //I am just lazy here!!
    connect(s, (struct sockaddr *)&sa, sizeof(sa));

    write(s, "hello", 5); //send it to server, better use while
    char buf[BUFLLEN];
    int rc;
    memset(buf, 0, BUFLLEN);
    char* pc = buf;
    while(rc = read(cli_socket, pc, BUFLLEN - (pc - buf)))
        pc += rc;
    write(1, buf, strlen(buf));
    close(cli_socket);
}
```

Create
client socket,
connect to server



Example: C++ server (TCP)

```
//include header files
#define PORT 6789
int main(int argc, char* argv[]) {
    struct sockaddr_in sa, csa;
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(PORT);
    sa.sin_addr.s_addr = INADDR_ANY; //any IP addr. Is accepted
    int s = socket(AF_INET, SOCK_STREAM, 0);
    assert( s >= 0);
    int rc = bind(s, (struct sockaddr *)& sa, sizeof(sa)); //hook s with port
    rc = listen(s, 5);
    int cs_socket = accept(s, (struct sockaddr *)& csa, sizeof(csa));
    char buf[BUFLEN];
    memset(buf, 0, BUFLen);
    char* pc = buf; int bcount = 0;
    while(bcount < 5) {
        if (rc = read(cs_socket, pc, BUFLen - (pc - buf)) > 0) {
            pc += rc; bcount += rc;
        } else return -1;
    }
    upper_case(buf); // covert it into upper case
    write(cs_socket, buf, strlen(buf));
    close(cs_socket);
    close(s);
}
```

Multi-Clients Servers

- ❑ Two main approaches to designing such servers.
- ❑ **Approach 1.**
- ❑ The first approach is using one process that awaits new connections, and one more process (or thread) for each Client already connected. This approach makes design quite easy, cause then the main process does not need to differ between servers, and the sub-processes are each a single-Client server process, hence, easier to implement.
- ❑ However, this approach wastes too many system resources (if child processes are used), and complicates inter-Client communication: If one Client wants to send a message to another through the server, this will require communication between two processes on the server, or locking mechanisms, if using multiple threads.
- ❑ Other approaches not included here

Socket programming with UDP

UDP: no "connection" between client and server

- ❑ no handshaking
- ❑ sender explicitly attaches IP address and port of destination
- ❑ server must extract IP address, port of sender from received datagram

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Summary

Our study of network apps now complete!

- application service requirements:
 - reliability, bandwidth, delay
- client-server paradigm
- Internet transport service model
 - connection-oriented, reliable: TCP
 - unreliable, datagrams: UDP
- specific protocols:
 - http
 - ftp
 - smtp, pop3
 - dns
- socket programming
 - client/server implementation
 - using tcp, udp sockets

Summary

Most importantly: learned about *protocols*

- ❑ typical request/reply message exchange:
 - client requests info or service
 - server responds with data, status code
- ❑ message formats:
 - headers: fields giving info about data
 - data: info being communicated
- ❑ control vs. data msgs
 - in-based, out-of-band
- ❑ centralized vs. decentralized
- ❑ stateless vs. stateful
- ❑ reliable vs. unreliable msg transfer
- ❑ "complexity at network edge"
- ❑ security: authentication

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

select function call

- ❑ `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"

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

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);
```


Final Thoughts

- ❑ Make sure to `#include` the header files that define used functions
- ❑ Check man-pages and course web-site for additional info