

ECE 463

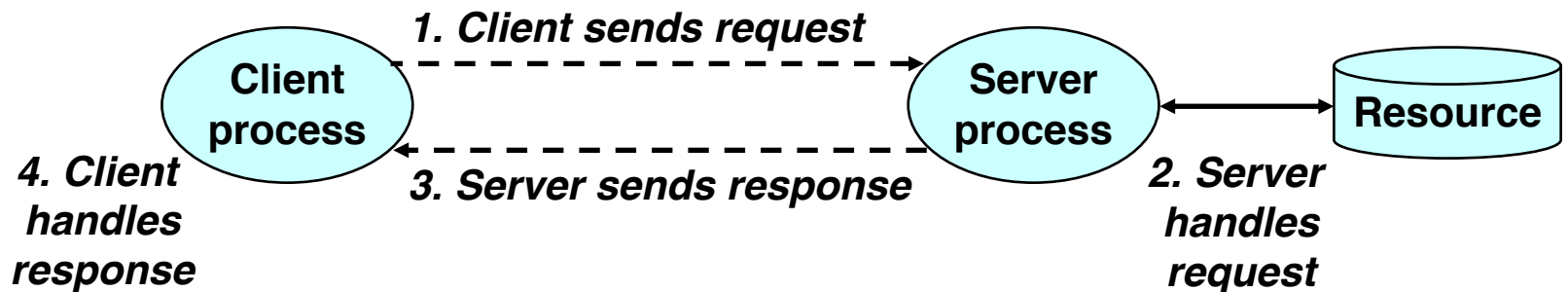
Introduction to Computer Networks

Lecture: Socket Programming

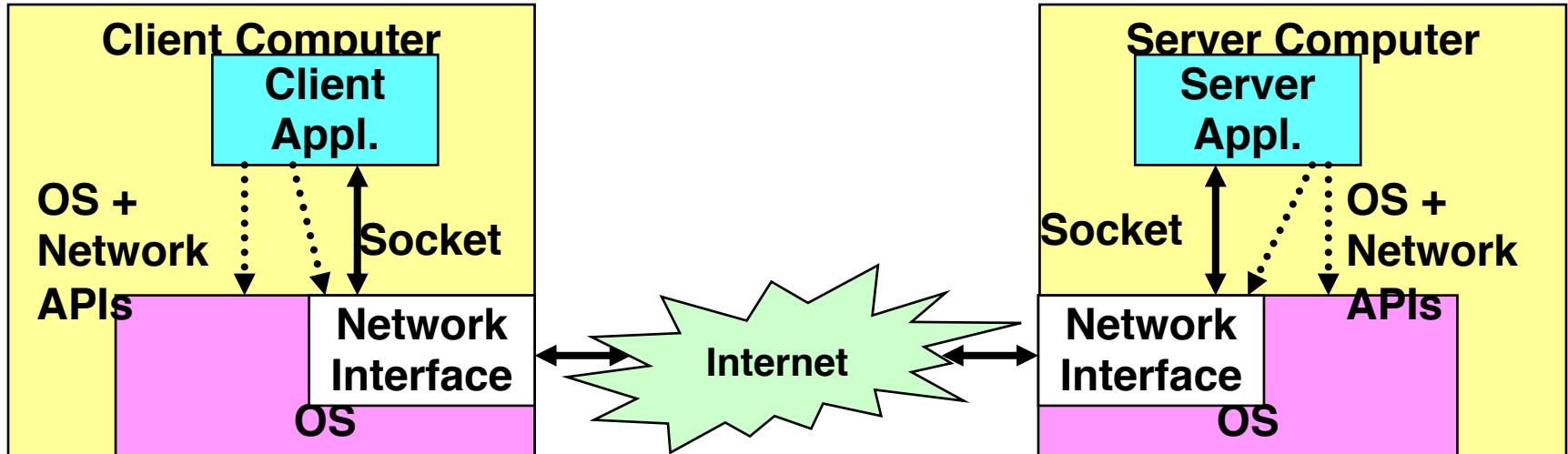
Sanjay Rao

A Client-Server Transaction

- Every network application is based on the client-server model:
 - A *server* process and one or more *client* processes
 - Server manages some *resource*.
 - Server provides *service* by manipulating resource for clients.



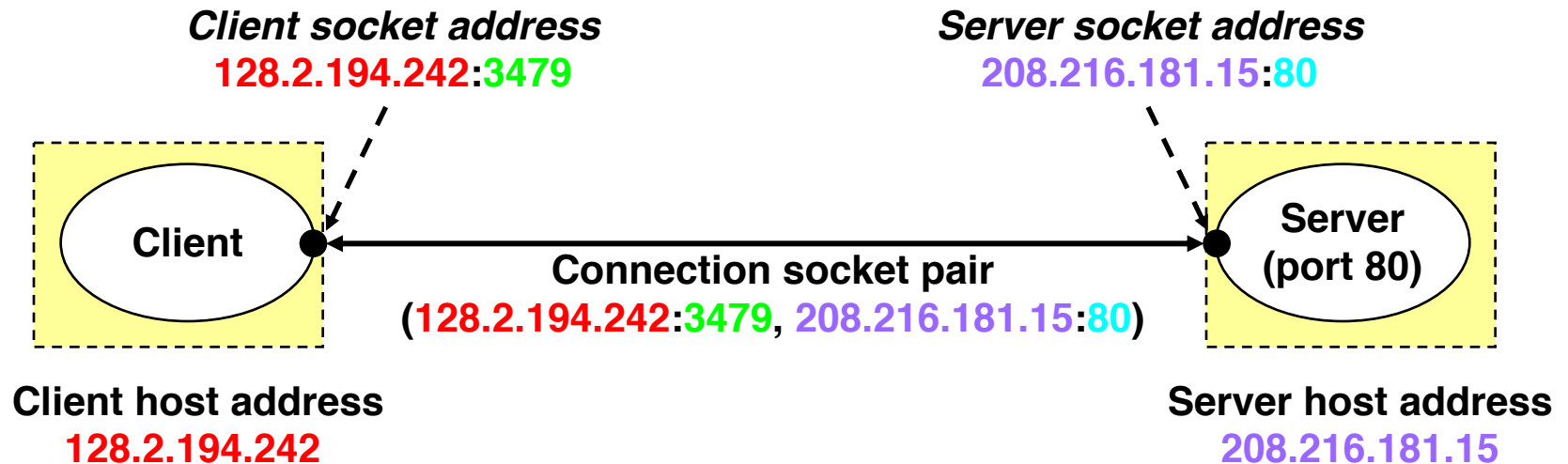
Network Applications



- Access to Network via Program Interface
 - Sockets make network I/O look like files
 - Call system functions to control and communicate
 - Network code handles issues of routing, reliability, ordering, etc.

Internet Connections (TCP/IP)

- Clients and servers communicate by sending streams of bytes over *connections*.
- Connections are point-to-point, full-duplex (2-way communication), and reliable.



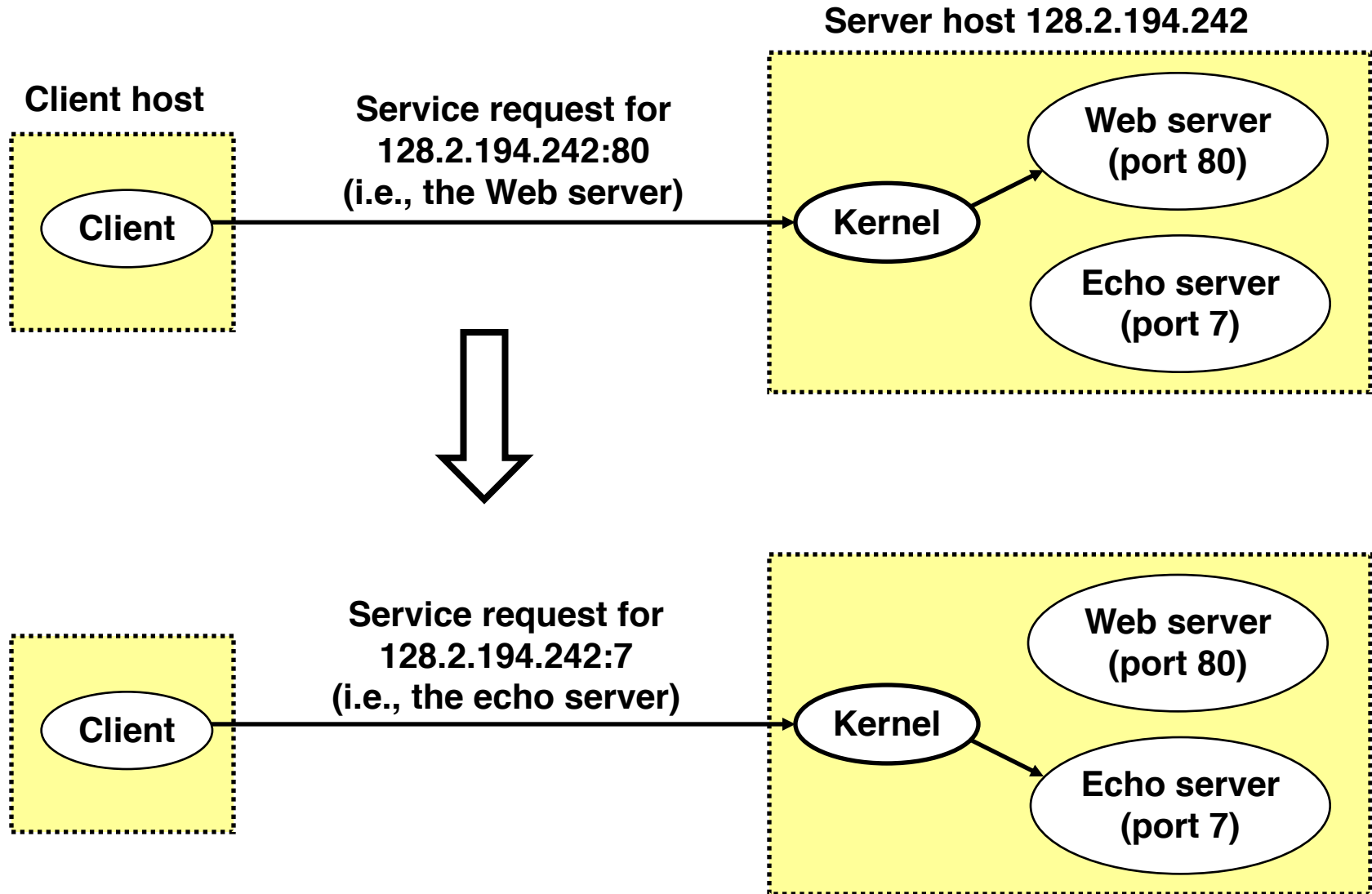
Note: 3479 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers

Clients

- Examples of client programs
 - Web browsers, `ftp`, `telnet`, `ssh`
- How does a client find the server?
 - The IP address in the server socket address identifies the host
 - The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
 - Examples of well known ports
 - Port 7: Echo server
 - Port 23: Telnet server
 - Port 25: Mail server
 - Port 80: Web server

Using Ports to Identify Services

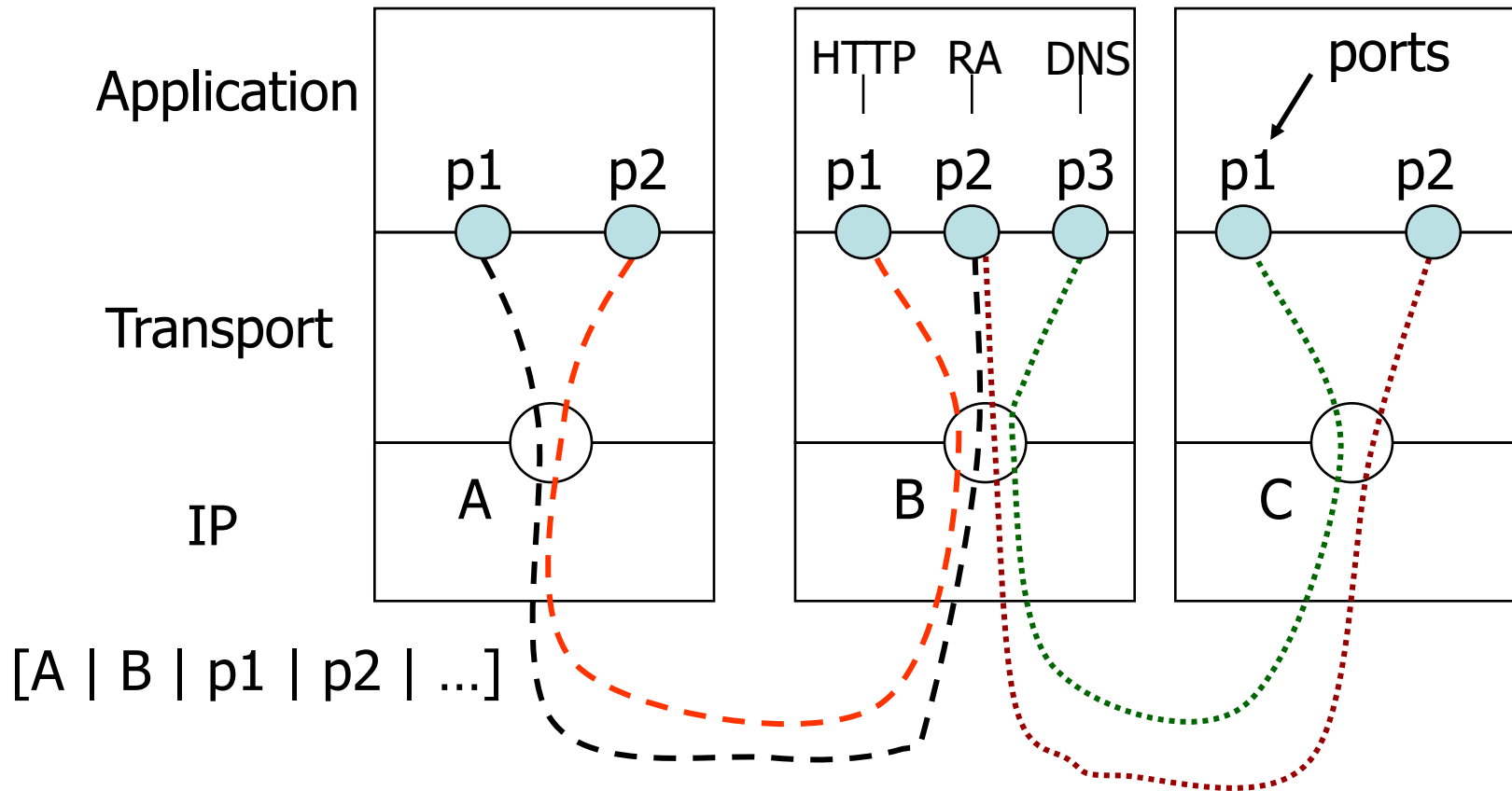


Servers

- Servers are long-running processes (daemons).
 - Created at boot-time (typically) by the init process
 - Run continuously until the machine is turned off.
- Each server waits for requests to arrive on a well-known port associated with a particular service.
 - Port 7: echo server
 - Port 23: telnet server
 - Port 25: mail server
 - Port 80: HTTP server
- A machine that runs a server process is also often referred to as a “server.”

See `/etc/services` for a comprehensive list of the services available on a Linux machine.

Transport Layer



Multiplexing using Ports

- Well-known Vs. Ephemeral Ports
 - 0-1023 “well-known” port numbers
 - Typically used by servers of well-known apps
 - Higher numbers: “ephemeral”

Socket Programming using TCP

- TCP: “Byte Stream” Abstraction
 - Sender sends a stream of bytes
 - Receiver gets a stream of bytes
- Guarantees:
 - Reliability
 - In-Order Delivery

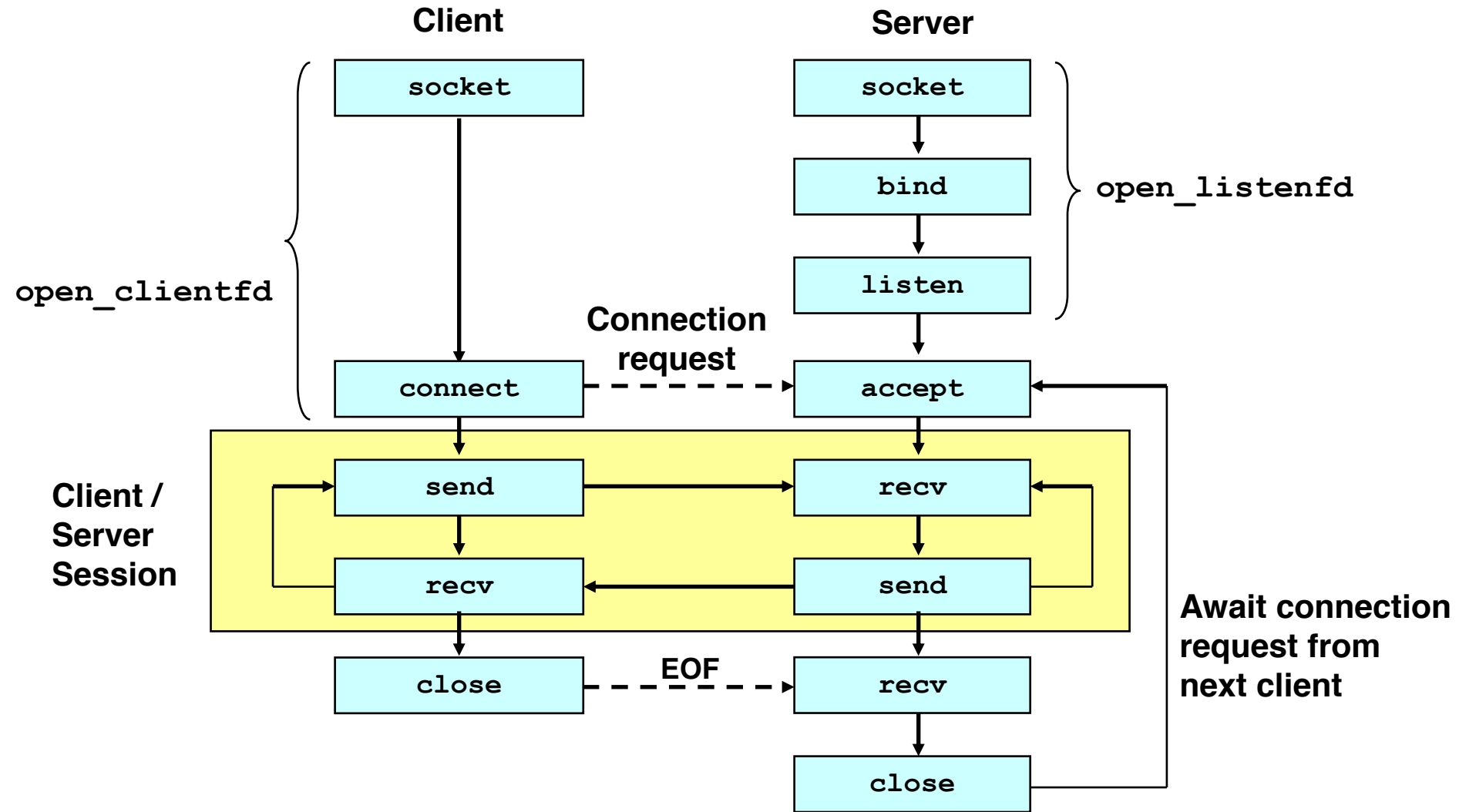
Sockets Interface

- Created in the early 80's
 - Part of the original Berkeley distribution of Unix
 - Contained an early version of the Internet protocols.
- Provides a user-level interface to the network.
- Underlying basis for all Internet applications.
- Based on client/server programming model.

Sockets

- What is a socket?
 - To the kernel: endpoint of communication.
 - To an application: a file descriptor that lets the application read/write from/to the network.
 - All Unix I/O devices are modeled as files.
- Clients and servers communicate with each by reading from and writing to socket descriptors.
- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.

Overview of the Sockets Interface



Echo Client Main Routine

```
/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];

    host = argv[1];
    port = atoi(argv[2]);

    clientfd = Open_clientfd(host, port);

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        write(clientfd, buf, strlen(buf));
        read(clientfd, buf, MAXLINE,);
        fputs(buf, stdout);
    }
    close(clientfd);
    exit(0);
}
```

Send line to
server

Receive line
from server

Echo Client: open_clientfd

```
int open_clientfd(char *hostname, int port)
{
    int clientfd;
    struct hostent *hp;
    struct sockaddr_in serveraddr;

    if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1; /* check errno for cause of error */

    /* Fill in the server's IP address and port */
    if ((hp = gethostbyname(hostname)) == NULL)
        return -2; /* check h_errno for cause of error */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)hp->h_addr,
          (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
    serveraddr.sin_port = htons(port);

    /* Establish a connection with the server */
    if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
        return -1;
    return clientfd;
}
```

This function opens a connection from the client to the server at hostname:port

Echo Client: `open_clientfd` (`socket`)

- `socket` creates a socket descriptor on the client.
 - `AF_INET`: indicates that the socket is associated with Internet protocols.
 - `SOCK_STREAM`: selects a reliable byte stream connection.

```
int clientfd; /* socket descriptor */  
  
if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)  
    return -1; /* check errno for cause of error */  
  
... (more)
```


Echo Client: `open_clientfd` (`gethostbyname`)

- Convert server name to an Internet address (IP address)

```
int clientfd;                /* socket descriptor */
struct hostent *hp;          /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */

...

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
....
```

Socket Address Structures

- Generic socket address:
 - Address arguments to `connect`, `bind`, and `accept`.

```
struct sockaddr {  
    unsigned short  sa_family;    /* protocol family */  
    char            sa_data[14]; /* address data.  */  
};
```

- Internet-specific socket address:

```
struct sockaddr_in {  
    unsigned short  sin_family; /* address family (always AF_INET) */  
    unsigned short  sin_port;   /* port num in network byte order */  
    struct in_addr  sin_addr;   /* IP addr in network byte order */  
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */  
};
```

Echo Client: `open_clientfd`

- The client then builds the server's Internet address.

```
int clientfd;                /* socket descriptor */
struct hostent *hp;          /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */

...

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
bcopy((char *) hp->h_addr,
      (char *) &serveraddr.sin_addr.s_addr, hp->h_length);
serveraddr.sin_port = htons(port);
```

Endian Format

- Different architectures have different endian formats
- IP addr and port stored in network (big-endian) byte order
 - `htonl()` converts longs from host byte order to network byte order.
 - `htons()` converts shorts from host byte order to network byte order.
 - `ntohl()`, `ntohs()` respectively convert from network byte order to host byte order

Echo Client: `open_clientfd` (`connect`)

- The client creates a connection with the server.
 - Client process blocks until connection is created.
 - After resuming, the client is ready to begin exchanging messages with the server on descriptor `sockfd`.

```
int clientfd;                /* socket descriptor */
struct sockaddr_in serveraddr; /* server address */
typedef struct sockaddr SA;   /* generic sockaddr */
...
/* Establish a connection with the server */
if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
return clientfd;
}
```

Echo Server: Main Loop

- The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```
main() {  
  
    /* create and configure the listening socket */  
  
    while(1) {  
        /* accept(): wait for a connection request */  
        /* echo(): read and echo input lines from client til EOF */  
        /* Close(): close the connection */  
    }  
}
```

Echo Server: Main Routine

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
                           on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("Fd %d connected to %s (%s:%s)\n",
               connfd, hp->h_name, haddrp, ntohs(clientaddr.sin_port));
        echo(connfd);
        close(connfd);
    }
}
```

Echo Server: `open_listenfd`

```
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                    (const void *)&optval , sizeof(int)) < 0)
        return -1;

    ... (more)
```


Echo Server: open_listenfd (cont)

```
...

/* Listenfd will be an endpoint for all requests to port
   on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;

/* Make it a listening socket ready to accept
   connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;

return listenfd;
}
```

Echo Server: `open_listenfd` `(socket)`

- `socket` creates a socket descriptor on the server.
 - `AF_INET`: indicates that the socket is associated with Internet protocols.
 - `SOCK_STREAM`: selects a reliable byte stream connection.

```
int listenfd; /* listening socket descriptor */

/* Create a socket descriptor */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
```

Echo Server: `open_listenfd` (initialize socket address)

- Next, we initialize the socket with the server's Internet address (IP address and port)

```
struct sockaddr_in serveraddr; /* server's socket addr */
...
/* listenfd will be an endpoint for all requests to port
   on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
```

Echo Server: `open_listenfd` `(bind)`

- `bind` associates the socket with the socket address we just created.

```
int listenfd;                /* listening socket */
struct sockaddr_in serveraddr; /* server's socket addr */

...
/* listenfd will be an endpoint for all requests to port
   on any IP address for this host */
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
```

Echo Server: `open_listenfd` `(listen)`

- `listen` indicates that this socket will accept connection (`connect`) requests from clients.

```
int listenfd; /* listening socket */

...
/* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;
return listenfd;
}
```

Echo Server: Main Routine

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
                           on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("Fd %d connected to %s (%s:%s)\n",
               connfd, hp->h_name, haddrp, ntohs(clientaddr.sin_port));
        echo(connfd);
        close(connfd);
    }
}
```

Echo Server: `accept`

- `accept()` blocks waiting for a connection.

```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

- `accept` returns a *connected descriptor* (`connfd`) with the same properties as the *listening descriptor* (`listenfd`)
 - Returns when the connection between client and server is created and ready for I/O transfers.
 - All I/O with the client will be done via the connected socket.
- `accept` also fills in client's IP address.

Echo Server: `accept` Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.



2. Client makes connection request by calling and blocking in `connect`.



3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.

Connected vs. Listening

- Listening descriptor
 - End point for client connection requests.
 - Created once and exists for lifetime of the server.
- Connected descriptor
 - End point of connection between client and server.
 - A new descriptor is created each time the server accepts a connection request from a client.
 - Exists only as long as it takes to service client.
- Why the distinction?
 - Allows for concurrent servers that can communicate over many client connections simultaneously.

Echo Server: Identifying Client

- The server can determine the domain name, IP address, and port of the client.

```
struct hostent *hp; /* pointer to DNS host entry */
char *haddrp;      /* pointer to dotted decimal string */

hp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                  sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
printf("Fd %d connected to %s (%s:%s)\n",
       connfd, hp->h_name, haddrp, ntohs(clientaddr.sin_port));
```

Echo Server: echo

- The server reads and echoes text lines until EOF (end-of-file) is encountered.
 - EOF notification caused by client calling `close(clientfd)`.
 - IMPORTANT: EOF is a condition, not a particular data byte.

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];

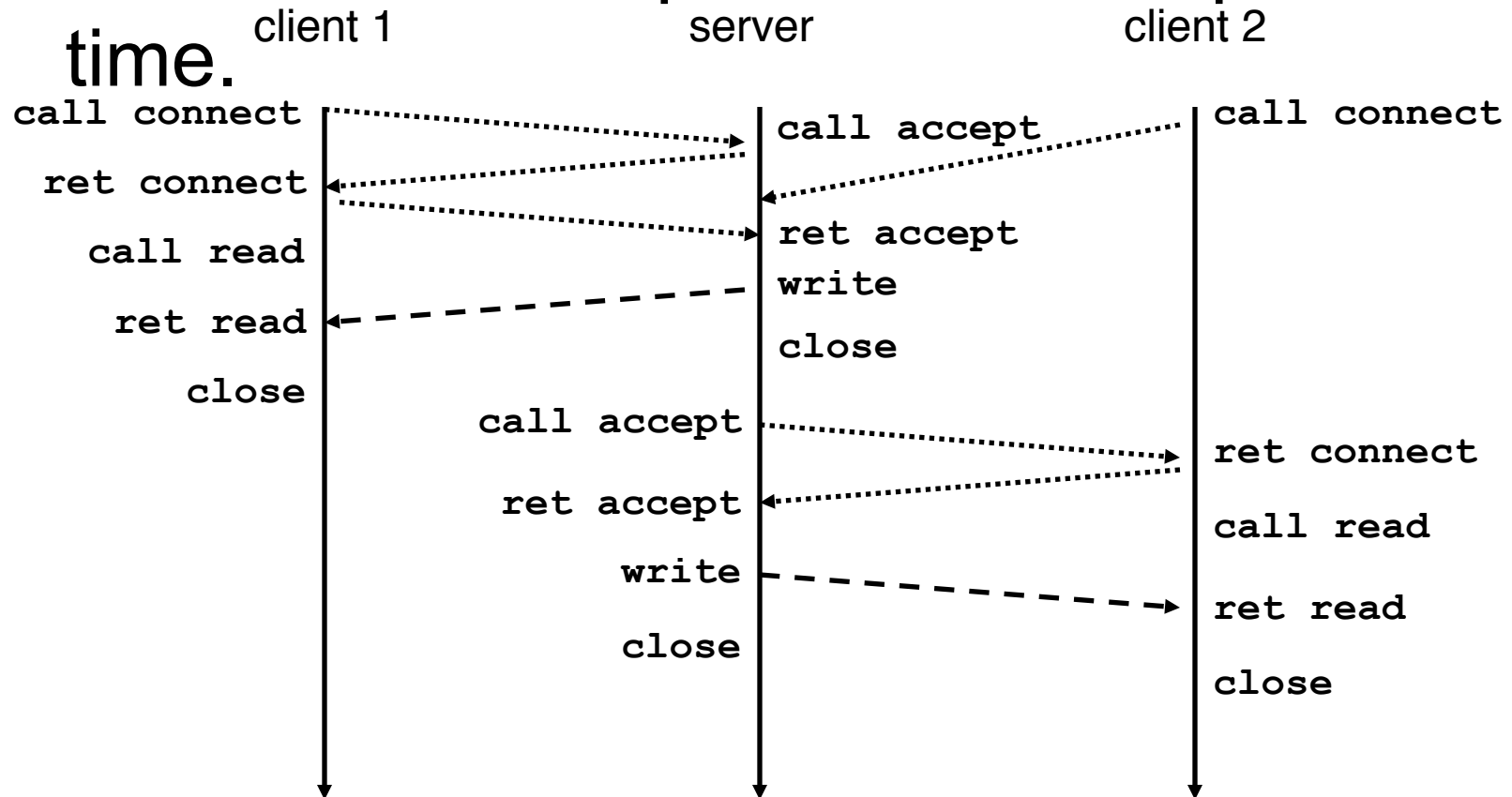
    while((n = read(connfd, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        write(connfd, buf, n);
    }
}
```

Receive line
from client

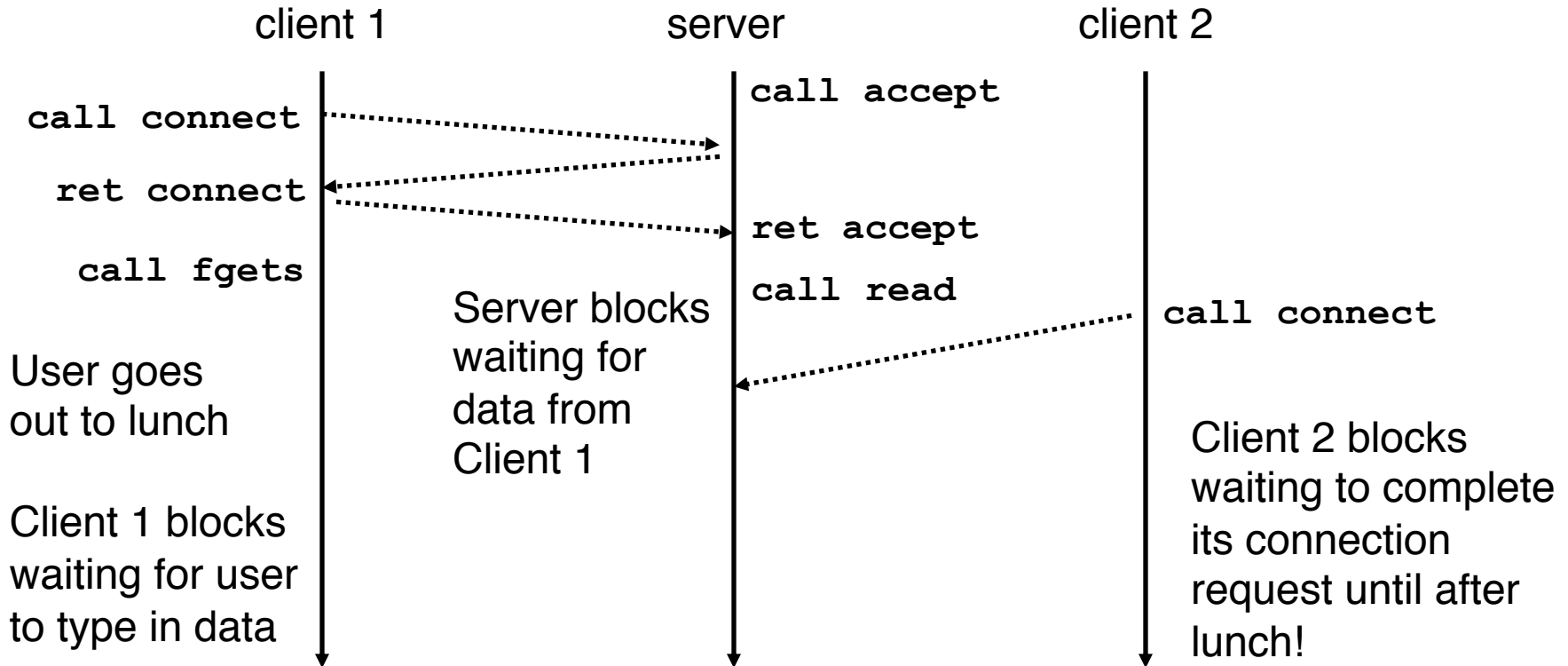
Send line to
client

Iterative Servers

- Iterative servers process one request at a time.



Iterative Servers: “Blocking”

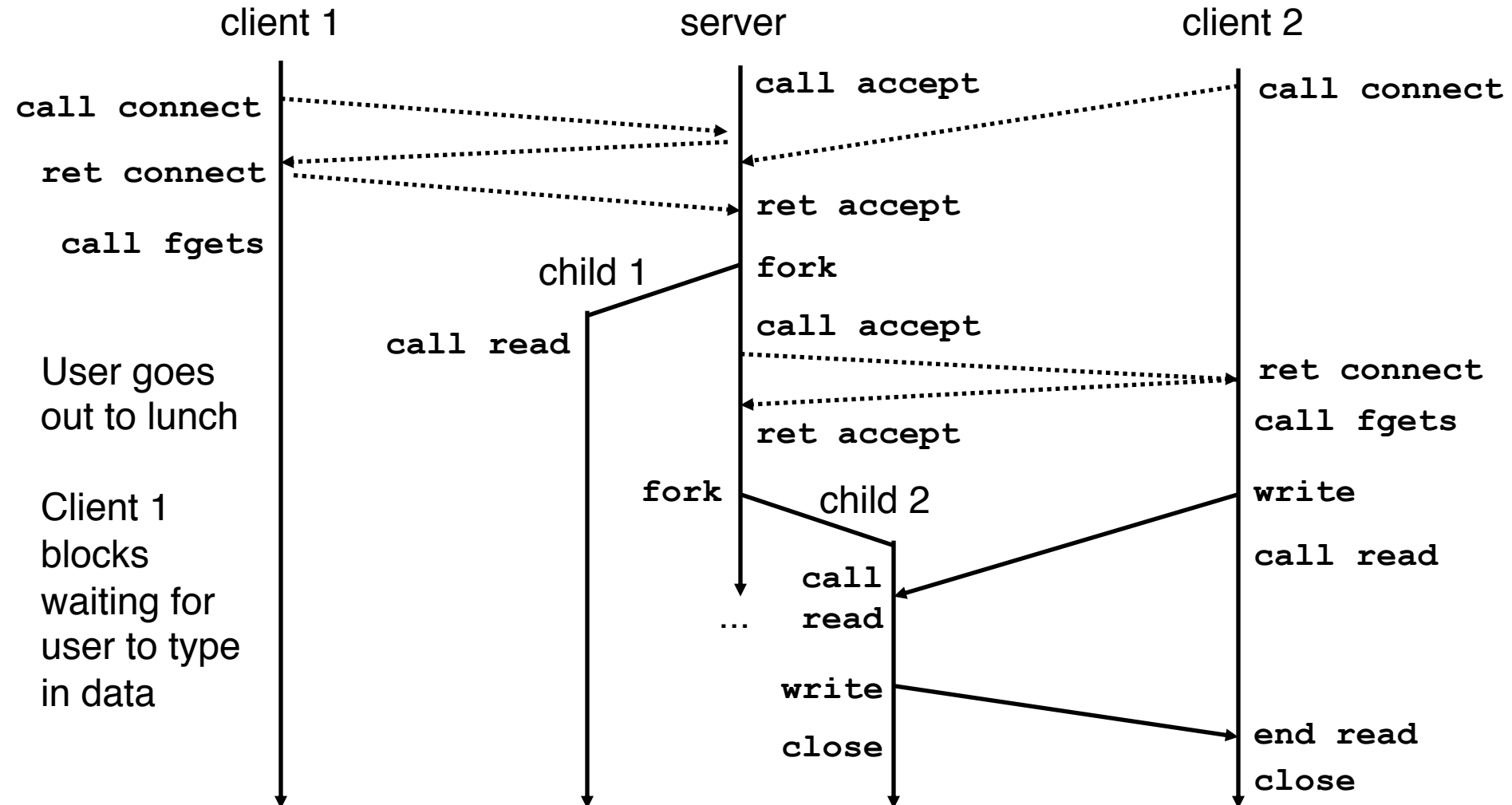


- Solution: use *concurrent servers* instead.
 - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.

Mechanisms for Concurrency

- 1. Processes
 - Each process has private address space.
- 2. I/O multiplexing with `select()`
 - User manually interleaves multiple flows.
 - Each flow shares the same address space.
- 3. Threads
 - Kernel interleaves multiple logical flows.
 - Each flow shares the same address space.

Concurrent Servers



Processes and Fork

- A **process** is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently.
- Fork:
 - Enables a process to make a copy of itself, so that one copy can handle one operation, while the other copy handles another task.

Fork() mechanics

- Calling fork()
 - Returns twice!
 - Once in parent process: ret value is childpid
 - Once in child process: ret value is 0
- All descriptors open in parent before the call to fork are shared with the child after fork returns.
 - Child and parent close descriptors not applicable to them.

Iterative Server

.....

```
while (1) {
    clientlen = sizeof(clientaddr);
    connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
    hp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                      sizeof(clientaddr.sin_addr.s_addr), AF_INET);
    haddrp = inet_ntoa(clientaddr.sin_addr);
    printf("Fd %d connected to %s (%s:%s)\n",
          connfd, hp->h_name, haddrp, ntohs(clientaddr.sin_port));
    echo(connfd);
    close(connfd);
}
}
```

Concurrent Server

.....

```
while (1) {
    clientlen = sizeof(clientaddr);
    connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);

    if( (childpid = Fork()) == 0){
        /* child process */

        close(listenfd);
        processRequest(connfd);
        exit(0);
    }
    close(connfd);          /* parent process */
}
```

Mechanisms for Concurrency

- 1. Processes
 - Each process has private address space.
- 2. I/O multiplexing with `select()`
 - User manually interleaves multiple flows.
 - Each flow shares the same address space.
- 3. Threads
 - Kernel interleaves multiple logical flows.
 - Each flow shares the same address space.

I/O Multiplexing with Select

- Select allows the process to instruct the kernel to wait for one of multiple events to occur.
- Process to be waken up when one of the events occurs.
- Example events:
 - New data arrived on a connection
 - A timeout has expired.

Example: Quiz Program

- Design a “quiz” program
 - Ask user a question,
 - If user answers question within the time, process question
 - Else if user doesn't respond within the time, he loses,
- How would we design this?

Example: Quiz program using select

```
#include <stdio.h>
#include <sys/select.h>
```

```
int main(void) {
    fd_set rfd;
    struct timeval tv;
    int retval;
    char ch1,ch2;

    while(1){
        tv.tv_sec = 5; tv.tv_usec = 0; /* Wait up to five seconds. */
        FD_ZERO(&rfd);
        FD_SET(0, &rfd);

        retval = select(1, &rfd, NULL, NULL, &tv);

        if (FD_ISSET(0, &rfd)){
            scanf("%c",&ch1); scanf("%c",&ch2);
            printf("Data is available now: %c %c",ch1,ch2);
        }
        else{
            printf("No data within five seconds.\n");
        }
    }
}
```

The select Function

```
int select(int maxfdp1,  
          fd_set *readset,  
          fd_set *writefds,  
          fd_set *errorfds,  
          struct timeval *timeout);
```

- readset**: Specifies file descriptors to check for being ready to read
- writefds**: Specifies the file descriptors to be checked for being ready to write
- errorfds**: Specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.
- timeout**: Specifies maximum time for which select must block.

The `select` Function

- Entries can be set to `NULL` if they are not relevant.

```
int select(int maxfdp1, fd_set *readset, NULL, NULL, NULL);
```

`readset`

- Opaque bit vector (max `FD_SETSIZE` bits) that indicates membership in a *descriptor set*.
 - On Linux machines, `FD_SETSIZE` = 1024
- If bit `k` is 1, then descriptor `k` is a member of the descriptor set.
- When call `select`, should have `readset` indicate which descriptors to test

`maxfdp1`

- Maximum descriptor in descriptor set plus 1.
 - Tests descriptors 0, 1, 2, ..., `maxfdp1` - 1 for set membership.
- `select()` returns the number of ready descriptors and keeps “on” each bit of `readset` for which corresponding descriptor is ready

Macros for Manipulating Set Descriptors

- `void FD_ZERO(fd_set *fdset);`
 - Turn off all bits in `fdset`.
- `void FD_SET(int fd, fd_set *fdset);`
 - Turn on bit `fd` in `fdset`.
- `void FD_CLR(int fd, fd_set *fdset);`
 - Turn off bit `fd` in `fdset`.
- `int FD_ISSET(int fd, *fdset);`
 - Is bit `fd` in `fdset` turned on?

Example: Waiting on 2 descriptors

```
.....
while (1){
    FD_ZERO(&rfd);
    FD_SET(socketfd1, &rfd);
    FD_SET(socketfd2, &rfd);
    maxfd= max(socketfd1,socketfd2)+1 // max is a function that returns
                                     max of 2 numbers.
    retval = select(maxfd, &rfd, NULL, NULL, NULL);

    if (FD_ISSET(socketfd1, &rfd)){
        // read data from socketfd1
    }
    else if (FD_ISSET(socketfd2, &rfd)){
        // read data from socketfd2
    }
}
```

Example: Concurrent Server

- Maintain a pool of connected descriptors.
 - 1 listening descriptor, 1 connected descriptor for each connection
- Repeat the following forever:
 - Use the Unix `select` function to block until:
 - (a) New connection request arrives on the listening descriptor.
 - (b) New data arrives on an existing connected descriptor.
 - If (a), add the new connection to the pool of connections.
 - If (b), read any available data from the connection
 - Close connection on EOF and remove it from the pool.

Pro and Cons of Event-Based Designs

- Design of choice for high-performance Web servers and search engines.
 - One logical control flow.
 - Can single-step with a debugger.
 - No process or thread control overhead.
- Key issues with select:
 - More complex to code than process- or thread-based designs.

---UDP Sockets---

- Discussion so far focused on client/server programming with TCP.
- Programming with UDP similar, but a few key differences.

TCP Vs. UDP

- TCP: Reliable, in-order delivery
- UDP: Unreliable, possibly out-of-order delivery
- TCP:
 - Connection-Oriented, Stream Semantics
 - TCP: “connections”: state that captures a given pair of end-points are communicating.
 - Stream of bytes sent across end-points
- UDP:
 - Connectionless, datagram semantics
 - No “state” across a sequence of packets

UDP Vs. TCP Sockets

- Call to `socket()`
 - Use `SOCK_DGRAM` instead of `SOCK_STREAM`

UDP:

```
clientfd = socket(AF_INET, SOCK_DGRAM, 0)
```

TCP:

```
clientfd = socket(AF_INET, SOCK_STREAM, 0)
```


UDP Vs. TCP: Functions called

- TCP Client
 - Socket
 - Connect
- TCP Server
 - Socket
 - Bind
 - Listen
 - Accept
- UDP Client
 - Socket
 - sendto/recvfrom
- UDP Server
 - Socket
 - Bind
 - sendto/recvfrom

write/read Vs. sendto/recvfrom

- TCP uses connect()
 - Establishes communication with given server.
 - All further reads/writes refer to this server.
- UDP: typically don't use connect()
 - All further sendto must specify who the intended destination is.

```
write(int fd, const void *buf, size_t count);
```

```
sendto(int fd, const void *buf, size_t count, int flags,  
        const struct sockaddr *dest_addr, socklen_t dest_len);
```

- Similar for read/recvfrom.

Stream Vs. Datagram Semantics

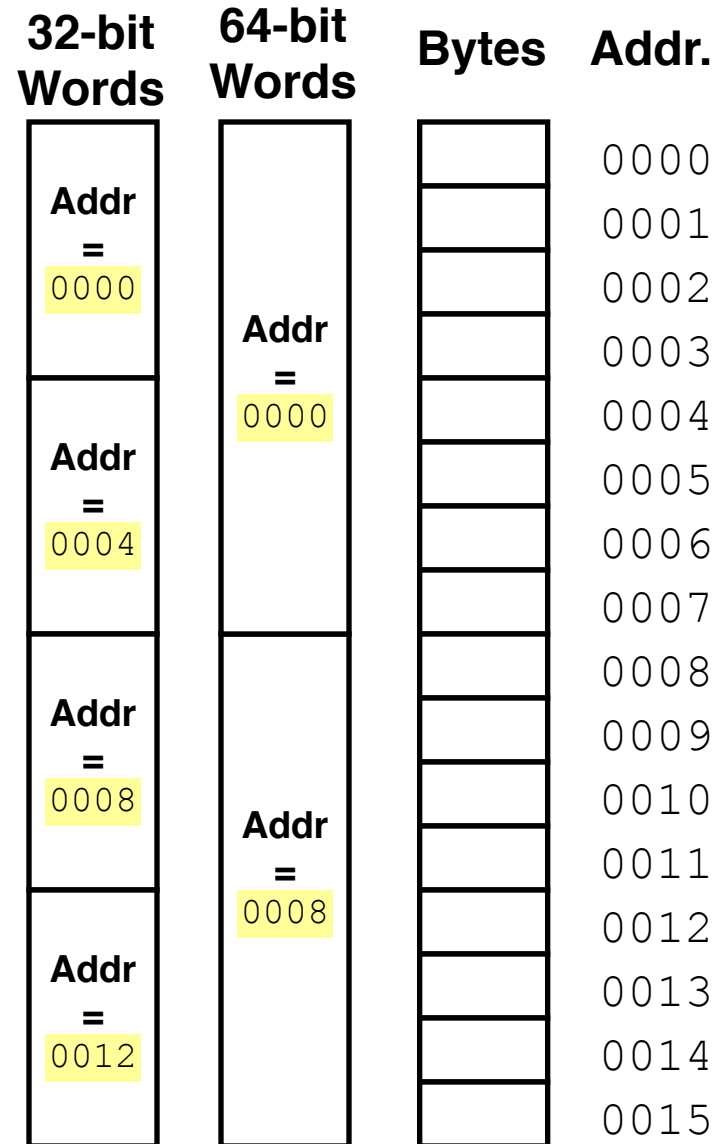
- TCP:
 - Assume application does:
`write(int fd, const void *buf, size_t count);`
 - No relationship between buffer size and actual TCP packets sent.
 - In fact, must put “write” (or “read”) inside a loop.
- UDP:
 - Assume application does:
`sendto(int fd, const void *buf, size_t count,...);`
 - `buf` corresponds to an actual UDP data packet sent
 - Each call to `sendto()` corresponds to one UDP packet.
 - Each call to `recvfrom()` reads entire UDP packet.

For More Information

- W. Richard Stevens, *Unix Network Programming: Networking APIs: Sockets and XTI*, Volume 1, Second Edition, Prentice Hall, 1998.
 - THE network programming bible.

C-programming Hints

- **Addresses Specify Byte Locations**
 - Address of first byte in word
 - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Simple Data Types

datatype	size	values
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)

Byte Ordering

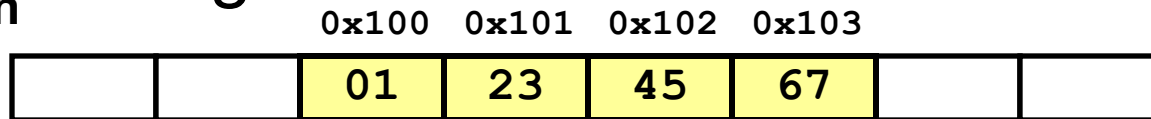
- How should bytes within multi-byte word be ordered in memory?
- Conventions
 - Sun's, Mac's are "Big Endian" machines
 - Least significant byte has highest address
 - Alphas, PC's are "Little Endian" machines
 - Least significant byte has lowest address

Byte Ordering Example

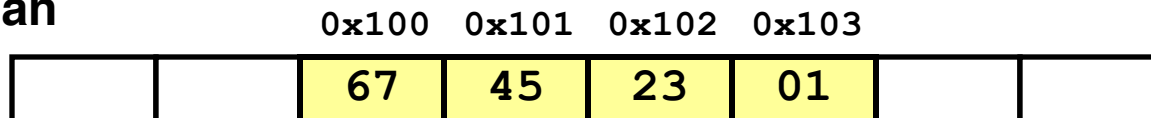
- Big Endian: LSB has highest address
- Little Endian: LSB has lowest address
- Example
 - Variable `x` has 4-byte representation

0x01234567

Big Endian – Address given by `&x` is 0x100



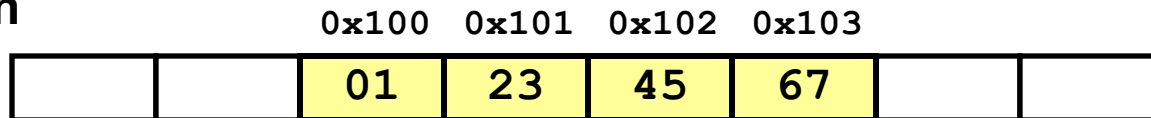
Little Endian



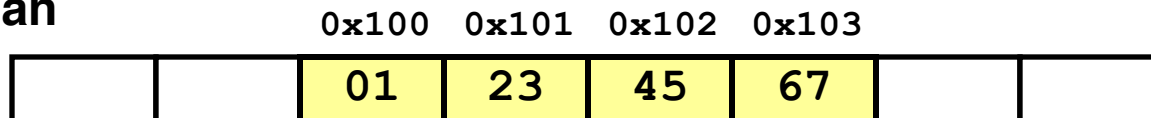
Byte Ordering Example

- What if character array?
 - 4 bytes 0x01 0x23 0x45 0x67

Big Endian



Little Endian



C Test

```
void main(){  
    int a= 0x01020304;  
  
    int *ptr= &a;  
    char *c = (char *)ptr;  
  
    char x = (*c);  
    printf ("%d \n",x);  
}
```

What happens if we type cast a (char *) to a (long *)?

More C-stuff

- ```
struct Pkt{
 int pktCode;
 char data[10];
};
```

```
Pkt pkt; ... fill up stuff ...
```

```
PktPtr *pktPtr=&pkt;
char *dataPtr= (char *)pktPtr;
```

```
char dataArr[10000];
memcpy(dataArr, pktPtr, sizeof(Pkt))
```

# Network Programming Tips

- Transfer data between machines
- Sender: struct -> char arr
- Receiver: char arr -> struct
- Need to be wary of big/little endian
- “Network” byte order:
  - ntohl, ntohs, htonl, htons

# Arrays

- **char** foo[80];
  - An array of 80 characters
  - **sizeof(foo)**  
= 80 × **sizeof(char)**  
= 80 × 1 = 80 bytes
- **int** bar[40];
  - An array of 40 integers
  - **sizeof(bar)**  
= 40 × **sizeof(int)**  
= 40 × 4 = 160 bytes

# Debuggers

- Gdb
  - gcc -g -o bin a.c
  - gdb bin <core>
  - gdb bin <processID>
  - Breakpoints etc.
- Valgrind/purify for memory errors