Ch11. Connecting to Processes Near and FAR - Servers and Sockets

Prof. Seokin Hong

Kyungpook National University

Fall 2019

Chapter Summary

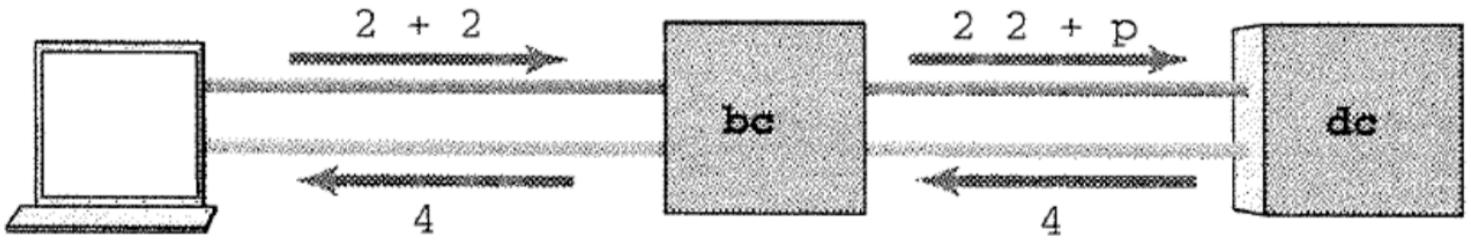
- Client/Server programming using pipes and sockets
- Interprocess communication and client/server design
- The idea and techniques of socket programming

bc: UNIX calculator

- bc

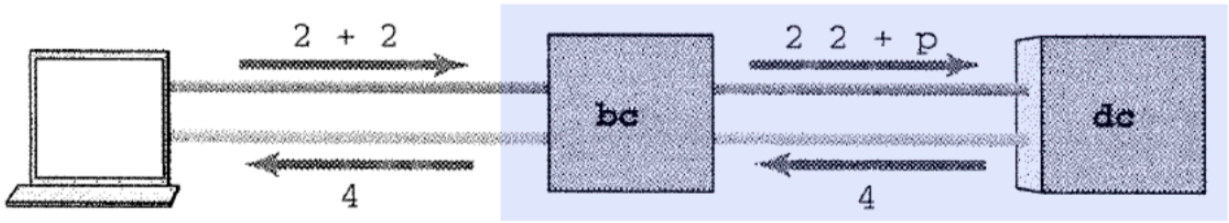
```
seokin@compasslab1:~$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
12^123
5486473221892422150934082166721730811348667928100671624512517021843 4\
56541709523359082780720277398867836972367369456704108169294512128
1000+1000
2000
20000000+200000
20200000
```

- bc is not a calculator

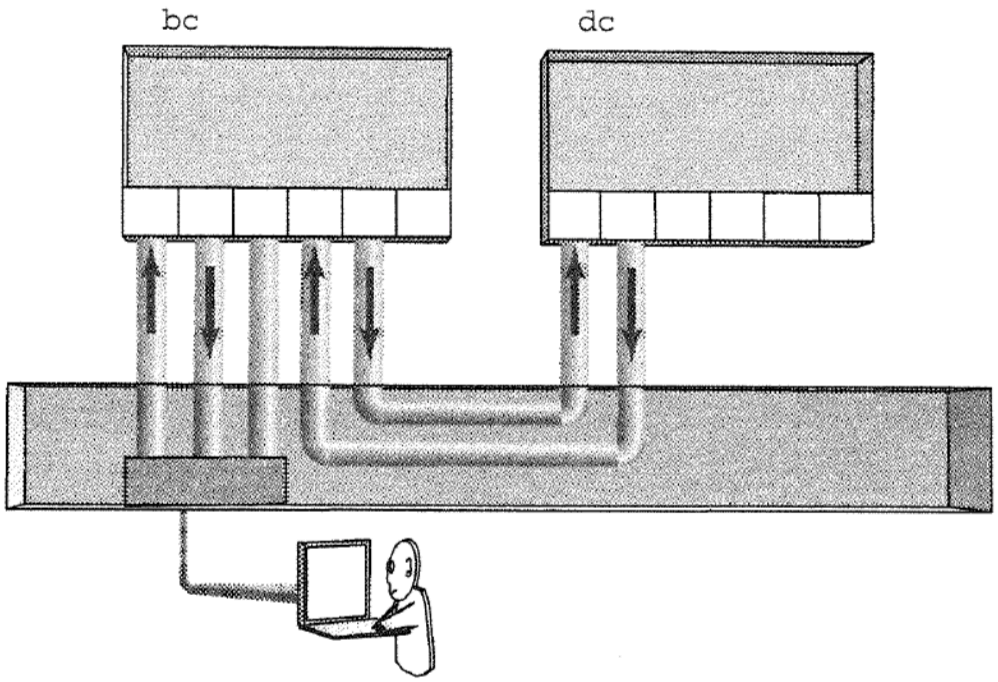


bc: UNIX calculator

- Ideas from bc
  - **Client/Server Model**
    - **dc** provides a service (calculation)
    - **bc** provides a user interface and uses the service
    - **bc** is called a client of dc
  - **Bidirectional Communication**
    - Using pipes, you need two pipes
  - **Persistent Service (coroutines)**
    - Both processes continue to run
    - Control passes from one to the other as each completed its part of the jobs
    - **bc** has the job of parsing input and printing the computation results
    - **dc** has the job of computing

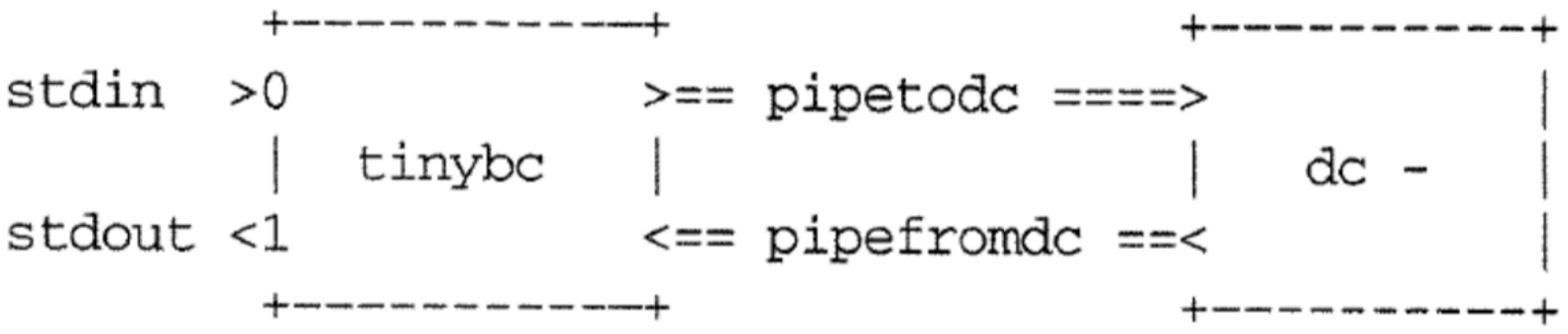


Coding bc: pipe, fork, dup, exec



- Create two pipes
- Create a process to run **dc**
- In the new process, redirect stdin and stdout to the pipes, then exec **dc**
- In the parent, read and parse user input, write commands to **dc**, read response from **dc**, and send response to user

Coding bc: pipe, fork, dup, exec



- **Program outline**
  - Get two pipes
  - fork (get another process)
  - In the dc-to-be process,
    - Connect stdin and out to pipes
    - Then exec **dc**
  - In the tinybc-process,
    - Talk to human via normal I/O and send stuff via pipe
  - Close pipe and dc dies

Ex1. tinybc.c

```c
#include    <stdio.h>
#include    <unistd.h>
#include    <stdlib.h>
#include    <fcntl.h>
#include <sys/wait.h>
#define oops(m,x)   { perror(m); exit(x); }

void be_dc(int in[2], int out[2]);
void be_bc(int todc[2], int fromdc[2]);
void fatal(char mess[]);

int main()
{
    int pid, todc[2], fromdc[2];    /*equipment */

    /* make two pipes */
    if( pipe(todc) == -1 || pipe(fromdc) == -1 )
        oops("pipe failed", 1);

    // test
    printf("todc[0]  : %d, todc[1]  : %d \n", todc[0], todc[1]);
    printf("fromdc[0]: %d, fromdc[1]: %d\n", fromdc[0], fromdc[1]);

    /* get a process for user interface */
    if(( pid = fork()) == -1 )
        oops("cannot fork", 2);
    if( pid == 0 )              /* child is dc  */
        be_dc(todc, fromdc);
    else {
        be_bc(todc, fromdc);        /* parent is ui */
        wait(NULL);         /* wait for child */
    }
}
```

Ex1. tinybc.c

```c
void be_dc(int in[2], int out[2])
/*
 *  set up stdin and stdout, then execl dc
 */
{
    /* setuup stdin from pipein */
    if( dup2(in[0], 0) == -1 )  /* copy read end to 0   */
        oops("dc: cannot redirect stdin", 3);
    close(in[0]);           /* moved to fd 0    */
    close(in[1]);           /* won't write here */

    /* setup stdout to pipeout */
    if( dup2(out[1], 1) == -1)  /* dupe write end to 1  */
        oops("dc: cannot redirect stdout", 4);
    close(out[1]);          /* moved to fd 1    */
    close(out[0]);          /* won't read from here */

    /* now execl dc with the - option */
    execlp("dc", "dc", "-", NULL);
    oops("Cannot run dc", 5);
}
```

Ex1. tinybc.c

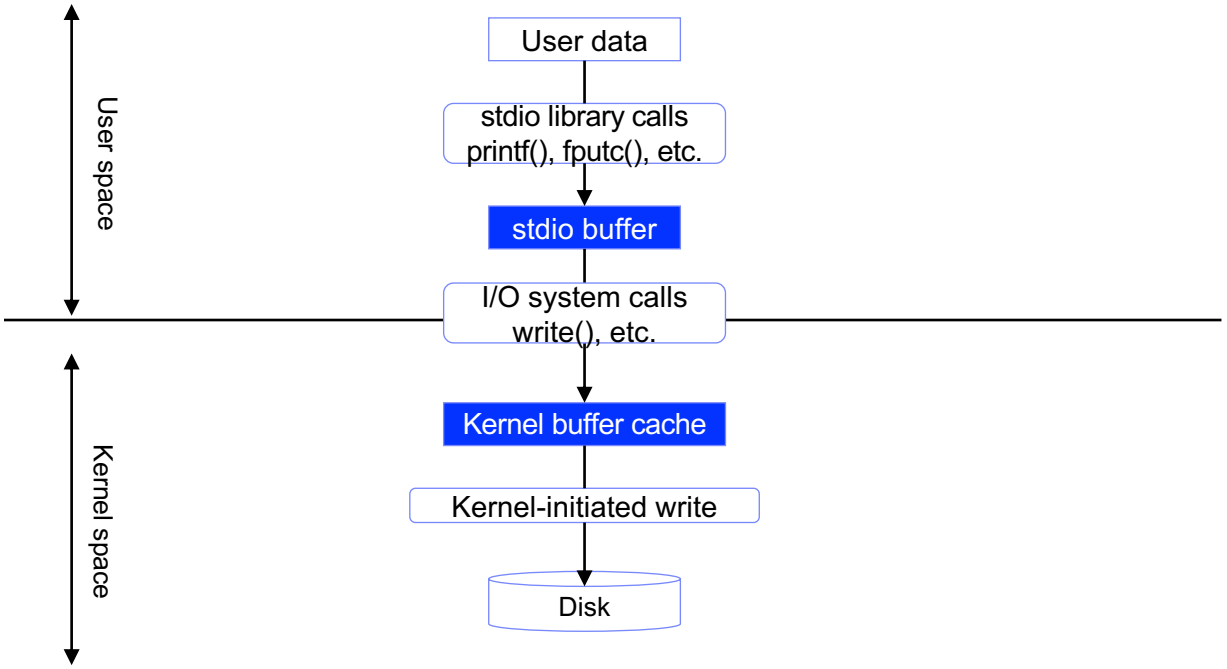
```c
void be_bc( int todc[2], int fromdc[2])
{
    int num1, num2;
    char    operation[BUFSIZ], message[BUFSIZ], *fgets();
    FILE    *fpout, *fpin, *fdopen();

    /* setup */
    close(todc[0]);         /* won't read from pipe to dc   */
    close(fromdc[1]);       /* won't write to pipe from dc  */

    fpout = fdopen( todc[1], "w" ); /* convert file descriptors to stream */
    fpin  = fdopen( fromdc[0], "r" );


    if( fpout == NULL || fpin == NULL )
        fatal("Error converting pipes to streams");

    /* main loop */
    while( printf("tinybc: "), fgets(message, BUFSIZ, stdin) != NULL )
    {
        /* parse input */
        if( sscanf(message, "%d%[-+*/^]%d", &num1, operation, &num2) != 3)
        {
            printf("syntax error\n");
            continue;
        }

        if( fprintf( fpout, "%d\n%d\n%c\np\n", num1, num2, *operation ) == EOF )
            fatal("Error writing");
        fflush( fpout );

        if( fgets( message, BUFSIZ, fpin ) == NULL )
            break;
        printf("%d %c %d = %s", num1, *operation, num2, message ); // stdout
    }
    fclose(fpout);  /* close pipe       */
    fclose(fpin);   /* dc will see EOF  */
}


void fatal( char mess[] )
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

Ex1. tinybc.c

```
    /* main loop */
    while( printf("tinybc: "), fgets(message, BUFSIZ, stdin) != NULL )
    {
        /* parse input */
        if( sscanf(message, "%d%[-+*/^]%d", &num1, operation, &num2) != 3)
        {
            printf("syntax error\n");
            continue;
        }

        if( fprintf( fpout, "%d\n%d\n%c\np\n", num1, num2, *operation ) == EOF )
            fatal("Error writing");
        fflush( fpout );

        if( fgets( message, BUFSIZ, fpin ) == NULL )
            break;
        printf("%d %c %d = %s", num1, *operation, num2, message ); // stdout
    }
    fclose(fpout);  /* close pipe       */
    fclose(fpin);   /* dc will see EOF  */
}


void fatal( char mess[] )
{
    fprintf(stderr, "Error: %s\n", mess);
    exit(1);
}
```

fdopen: Making file descriptors look

- ***File stream***
  - A pointer to a FILE structure that contains information about a file.
  - permits user-controllable buffering and formatted input and output.
- **fopen**: file name → FILE *
- **fdopen** : file descriptor → FILE *



Contents

- bc: a Unix Calculator
- **popen: Making Process Look like Files**
- Sockets: Connecting to Remote Processes

What popen does

- **fopen() opens a buffered connection to a file**

```
FILE *fp;
fp = fopen("file1", "r");
c = getc(fp);
fgets(buf, len, fp);
fscanf(fp, "%d%d%s", &x, &y, x);
```

- **popen() opens a buffered connection to a process**
  - **Implemented using fork(), pipe(), exec()**

```
File *fp;
fp = popen("ls", "r");
fgets(buf, len, fp);
pclose(fp);
```

What popen does

- Similarities between popen and fopen

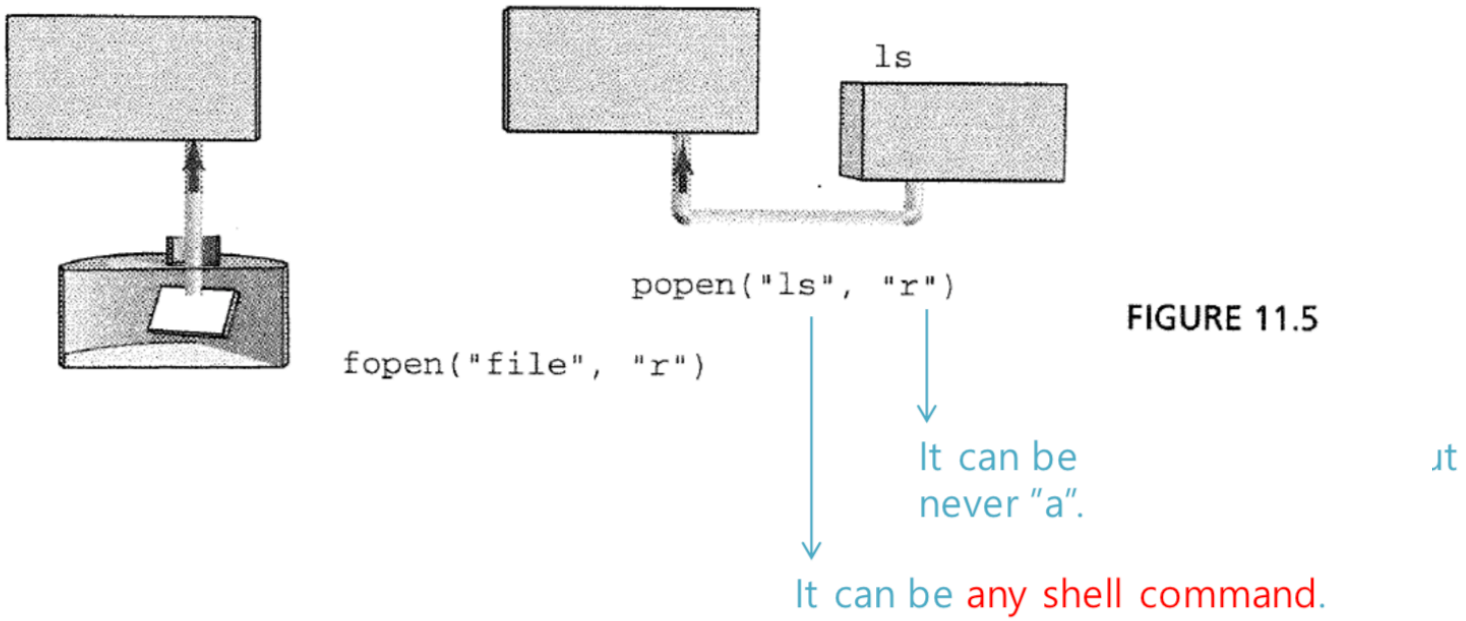


FIGURE 11.5

- **pclose()** is required
  - A process needs to wait while the child process executes
  - pclose() calls wait()

Ex2: popendemo.c

```c
#include    <stdio.h>
#include    <stdlib.h>

int main(void)
{
    FILE    *fp;
    char    buf[100];
    int i = 0;

    fp = popen( "who|sort", "r" );      /* open the command  */

    while ( fgets( buf, 100, fp ) != NULL ) /* read from command */
        printf("%3d %s", i++, buf );    /* print data        */

    pclose( fp );                   /* IMPORTANT!        */
    return 0;
}
```

Writing popen()

- How does popen work?
  - popen runs a program and returns a connection to the standard input or standard output of that program.

popen(cmd, “r”)

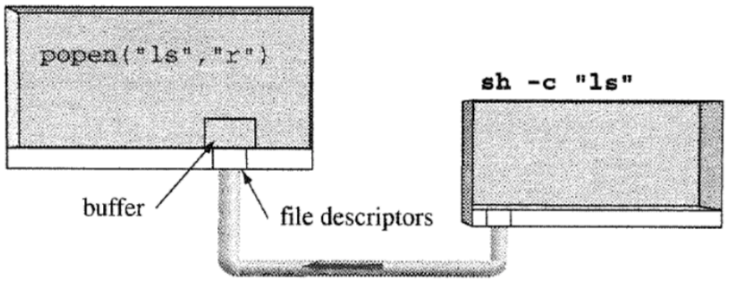
pipe(p)
fork()

Parent

Child

close(p[1])
fp = fdopen(p[0],”r”);
return fp;

close(p[0])
dup2(p[1],1);
close(p[1]);
execl(“/bin/sh”, “sh”, “-c”, cmd, NULL);



Ex3. popen.c

```c
#include    <stdio.h>
#include    <signal.h>
#include    <stdlib.h>
#include    <unistd.h>

#define READ    0
#define WRITE   1

FILE *popen(const char *command, const char *mode)
{
    int pfp[2], pid;        /* the pipe and the process */
    FILE    *fdopen(), *fp;     /* fdopen makes a fd a stream   */
    int parent_end, child_end;  /* of pipe          */

    if ( *mode == 'r' ){        /* figure out direction     */
        parent_end = READ;
        child_end = WRITE ;
    } else if ( *mode == 'w' ){
        parent_end = WRITE;
            child_end = READ ;
    } else return NULL ;


    if ( pipe(pfp) == -1 )          /* get a pipe       */
        return NULL;
    if ( (pid = fork()) == -1 ){        /* and a process    */
        close(pfp[0]);          /* or dispose of pipe   */
        close(pfp[1]);
        return NULL;
    }

    /* --------------- parent code here ------------------- */
    /*   need to close one end and fdopen other end     */

    if ( pid > 0 ){
        if (close( pfp[child_end] ) == -1 )
            return NULL;
        return fdopen( pfp[parent_end] , mode); /* same mode */
    }
```

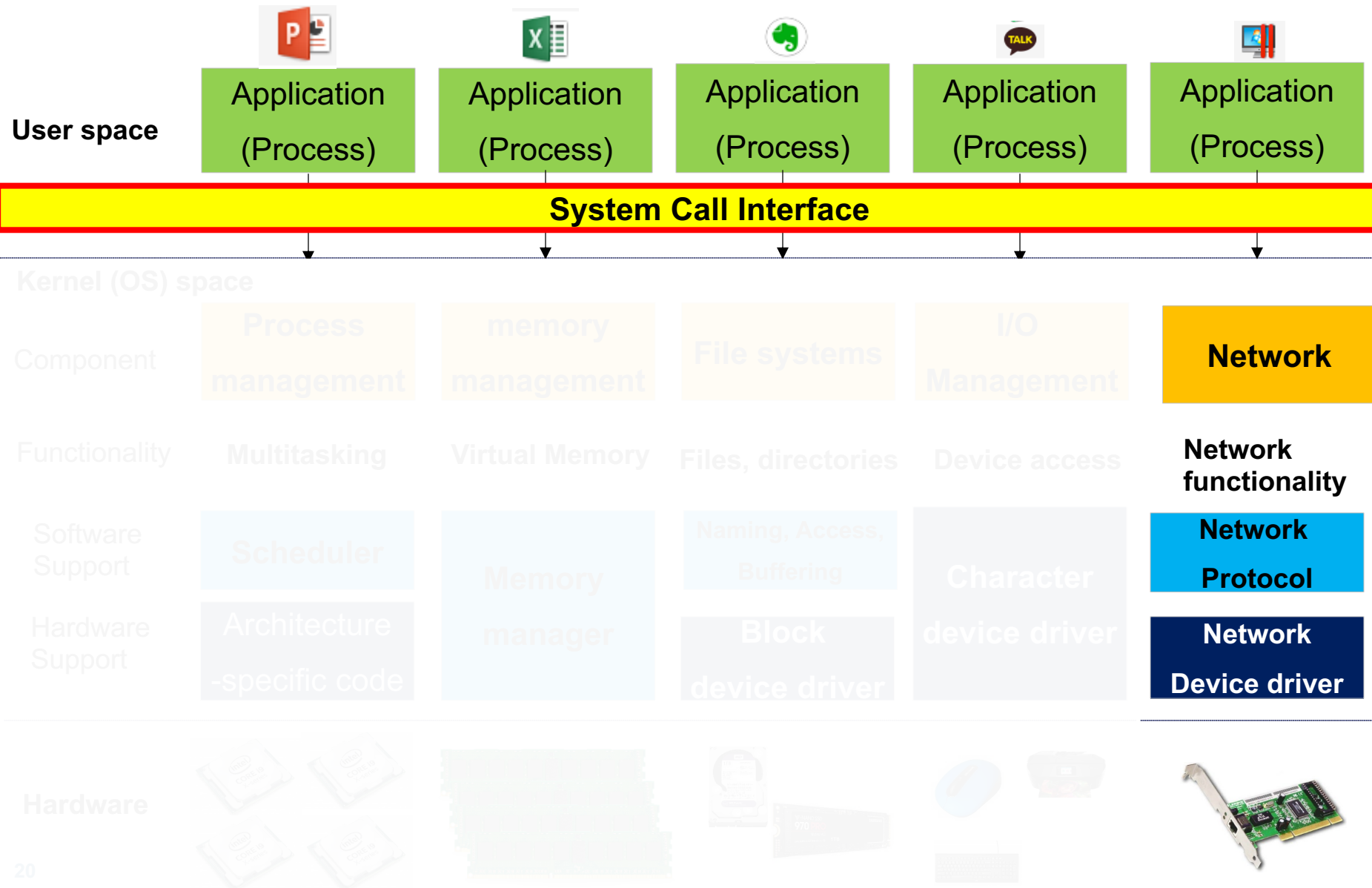
Ex3. popen.c

```
    /* --------------- child code here -------------------- */
    /*   need to redirect stdin or stdout then exec the cmd  */

    if ( close(pfp[parent_end]) == -1 ) /* close the other end  */
        exit(1);            /* do NOT return    */

    if ( dup2(pfp[child_end], child_end) == -1 )
        exit(1);

    if ( close(pfp[child_end]) == -1 )  /* done with this one   */
        exit(1);
                        /* all set to run cmd   */
    execl( "/bin/sh", "sh", "-c", command, NULL );
    exit(1);
}


int main()
{
    FILE    *fp;
    char    buf[BUFSIZ];

    fp = popen("ls","r");
    while( fgets(buf, BUFSIZ,fp) != NULL)
        fputs(buf, stdout);

    return 0;
}
```

Contents

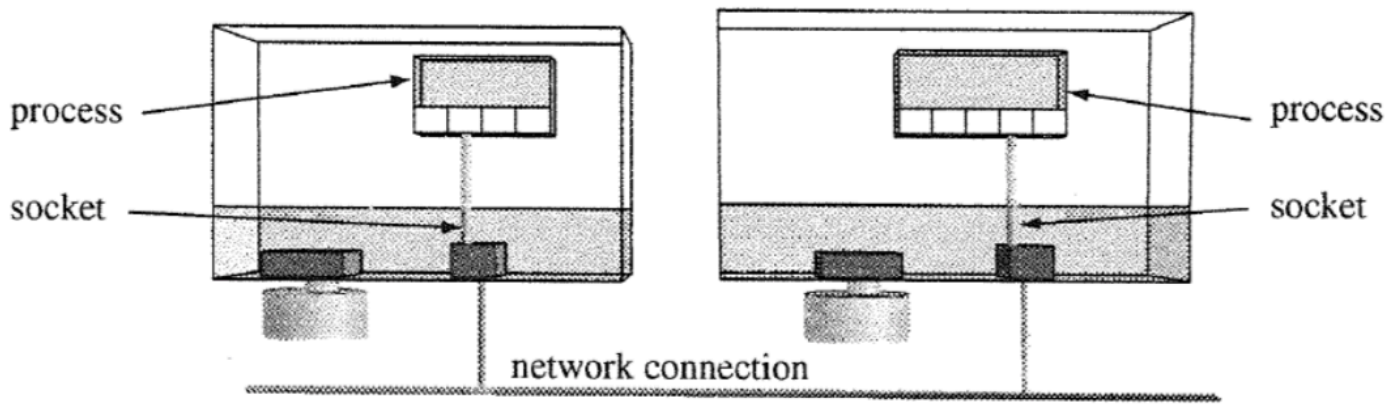
- bc: a Unix Calculator
- popen: Making Process Look like Files
- **Sockets: Connecting to Remote Processes**

Sockets: Connecting to Remote Processes



Sockets

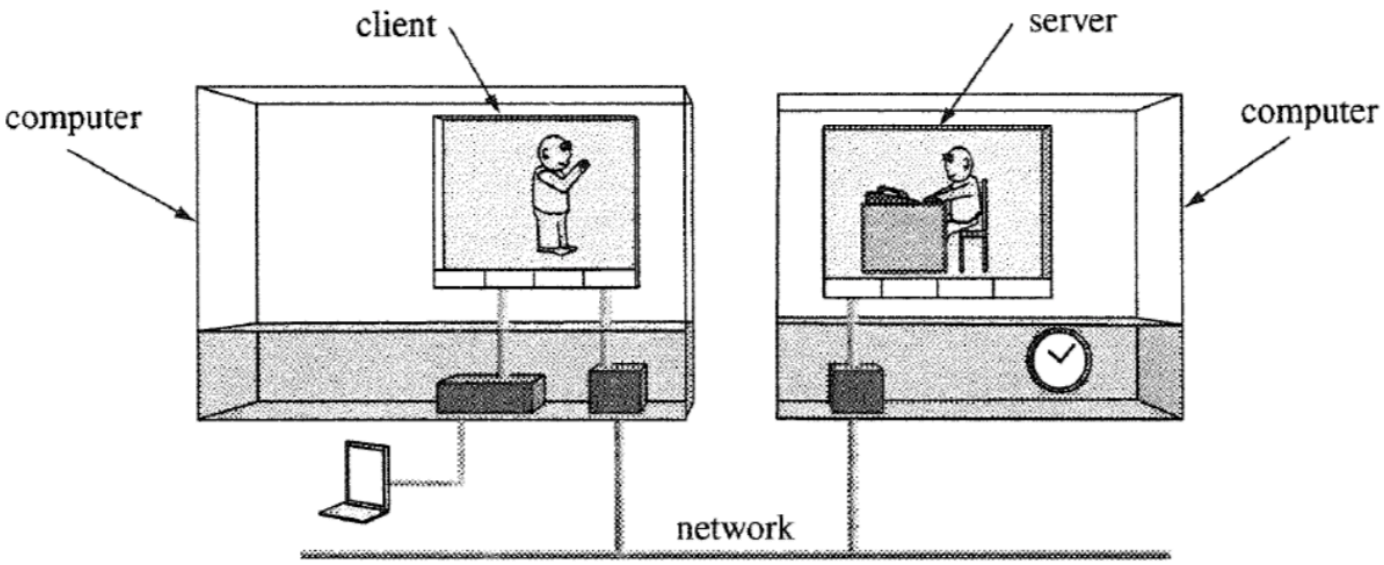
- Limitations of pipes
  - It can only connect related processes: ex) parent/child
  - It can only connect processes on the same computer
- Unix/Linux provides another method of interprocess communication: **sockets**
- Sockets allow processes to create pipe-like connections to unrelated processes and even to process on other computers



Important Concepts in Socket Programming

- **Client and server**
  - In Unix terms, server is a program, not a computer
  - A server process wait for a request, processes it and loops back to take the next request
- **Hostname and port:**
  - A server on the internet is a process running on a computer
  - The computer is called the host, and it has a hostname.
  - A port number is allocated to each server program.
- **Address**
  - Like phone-number or zip code, a host has an address
- **Protocol**
  - A set of rules that governs interaction between client and server

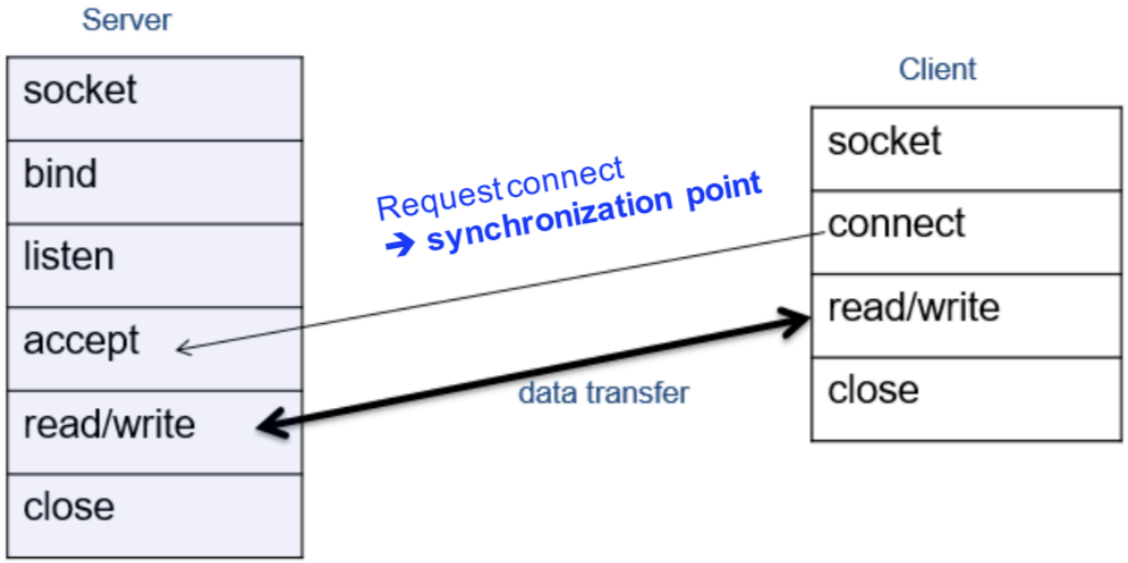
Example: Time Server and Client




```
[seokin@compasslab2 ch11]$ ./timeserv &
[1] 28592
[seokin@compasslab2 ch11]$ ./timeclnt compasslab2 13000
Wow! got a call!
The time here is ..Mon Nov 18 10:39:30 2019
```

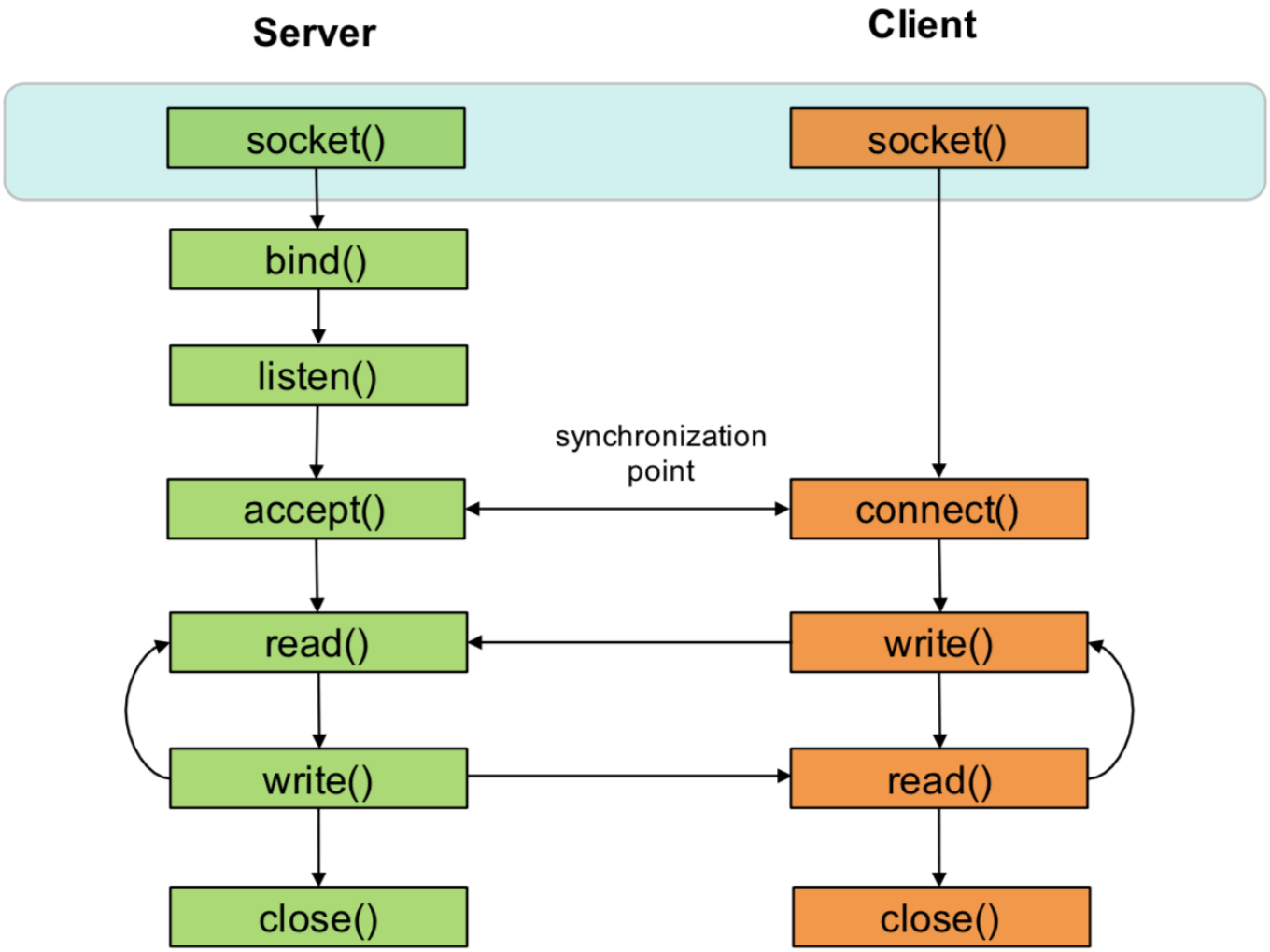
Writing timeserv.c : A Timer Server

Action	Syscall
Get a phone line	socket()
Assign a number	bind()
Allow incoming calls	listen()
Wait for a call	accept()
Transfer data	read()/write()
Hang up	close()



Writing timeserv.c : A Timer Server

- Step 1: Create a socket



Writing timeserv.c : A Timer Server

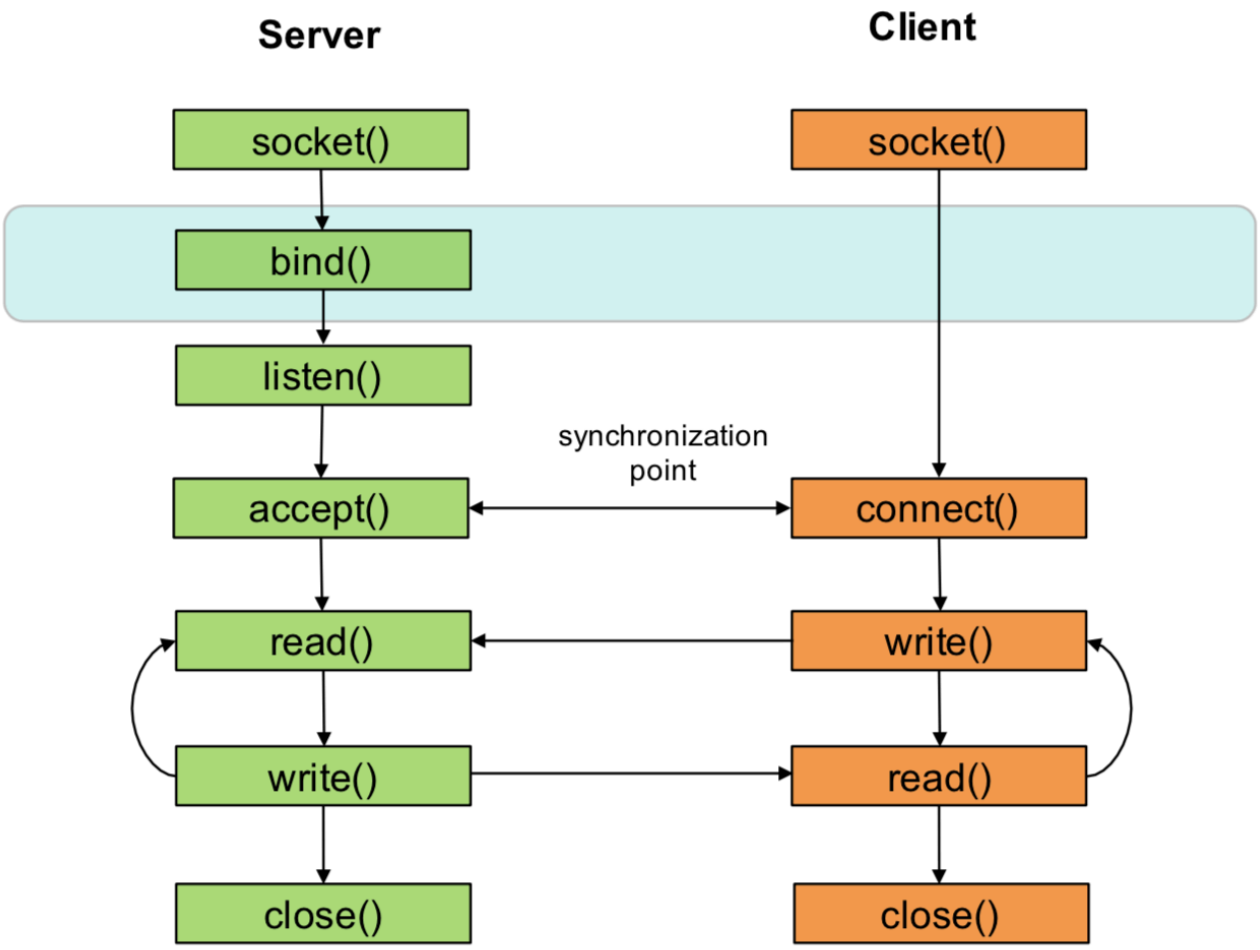
- Step 1: Create a socket → socket()

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
                                    Returns file descriptor on success, or −1 on error
```

  - **domain :** communication domain (PF_INET or PF_UNIX)
  - **type :** socket type (SOCK_STREAM or SOCK_DGRAM)
  - **protocol :**
    - Specify 0 (for SOCK_STREAM and SOCK_DGRAM)
    - Nonzero for some socket types (e.g., SOCK_RAW)
  - **Return value :** file descriptor

Writing timeserv.c : A Timer Server

- Step 2: Bind a socket to an address



Writing timeserv.c : A Timer Server

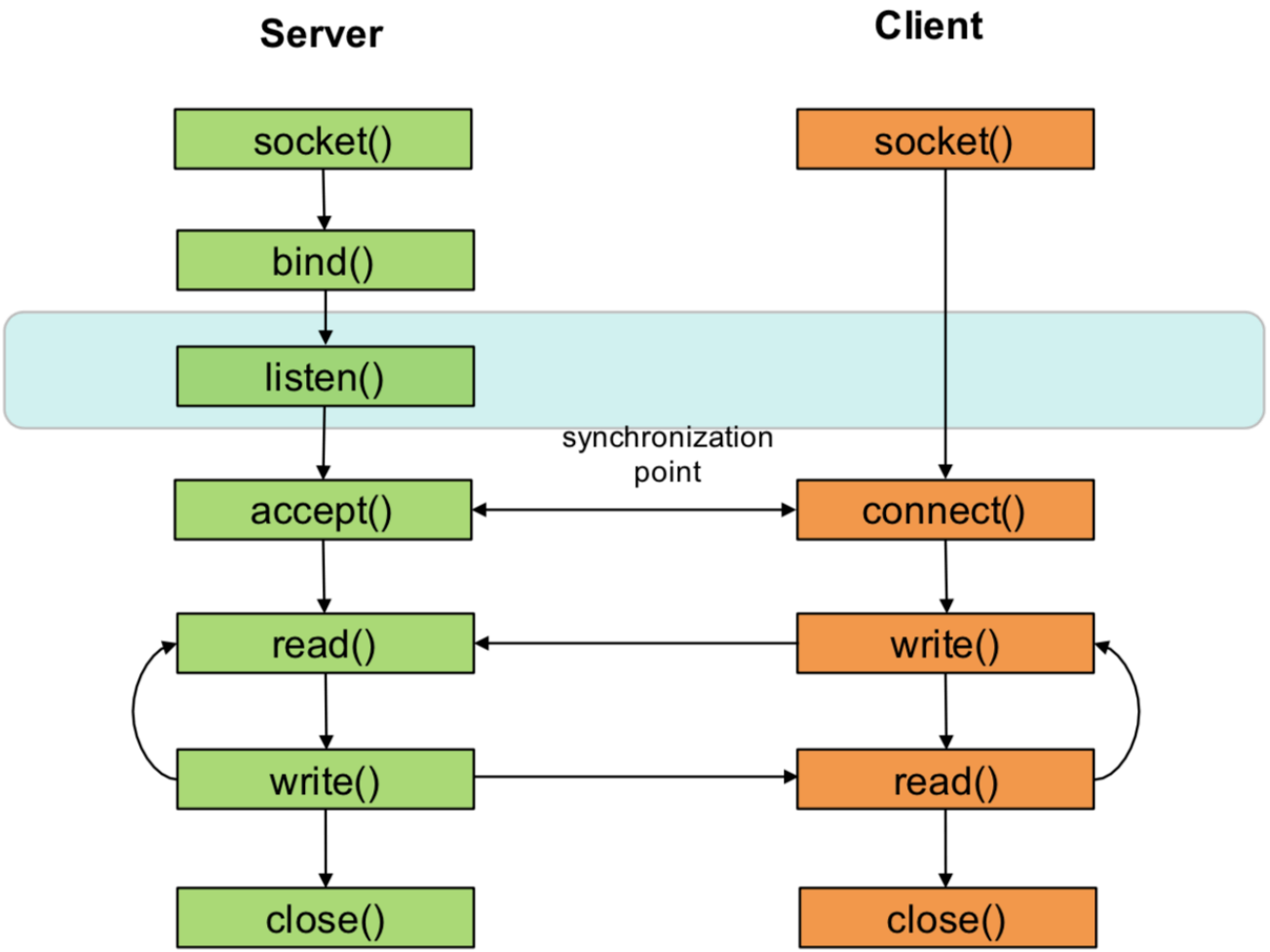
- Step 2: Bind a socket to an address → **bind()**

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                                    Returns 0 on success, or −1 on error
```

  - **sockfd :** file descriptor obtained from socket()
  - **addr :** a pointer to a structure specifying the address to which this socket is to be bound
  - **addrlen :** the size of the address structure

Writing timeserv.c : A Timer Server

- Step 3: Allow incoming calls on socket

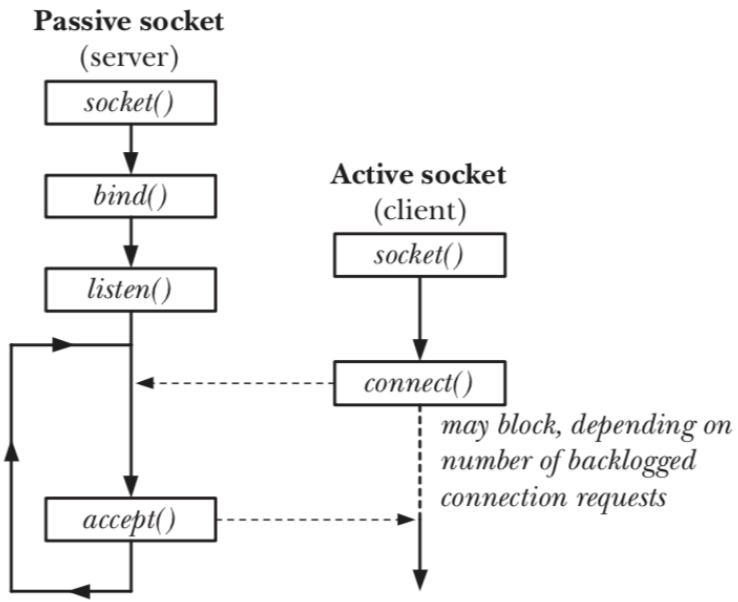


Writing timeserv.c : A Timer Server

- Step 3: Allow incoming calls on socket → **listen()**
  - Marks the socket referred to by the file descriptor as passive

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);          //Returns 0 on success, or –1 on error
```

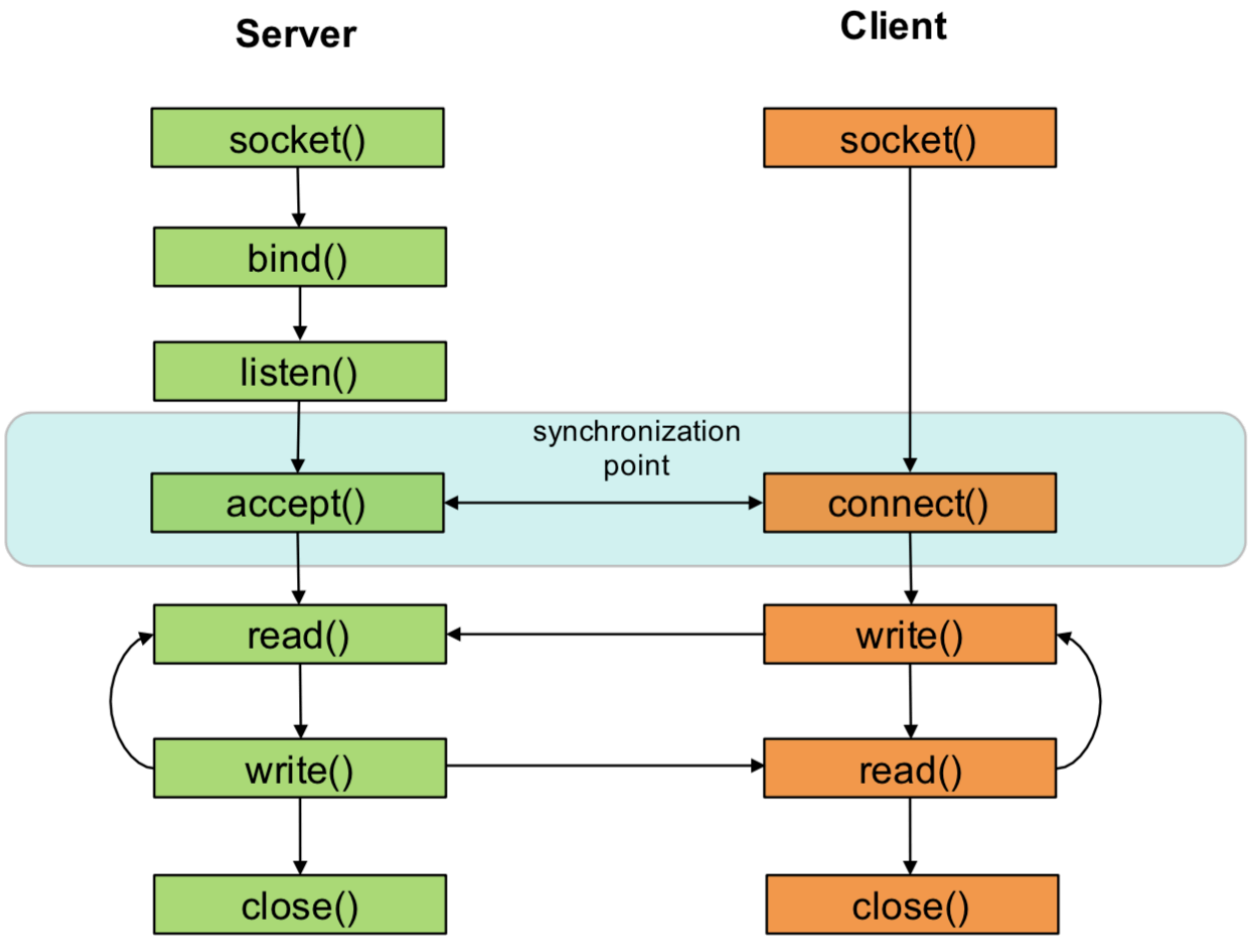
  - **sockfd :** file descriptor obtained from socket()
  - **backlog :** used to limit the number of pending request



Client may call connect() before server calls accept()
→ the connection is pended
→ kernel must record some information about each pending connection

Writing timeserv.c : A Timer Server

- Step 4: Wait for and accept a connection



Writing timeserv.c : A Timer Server

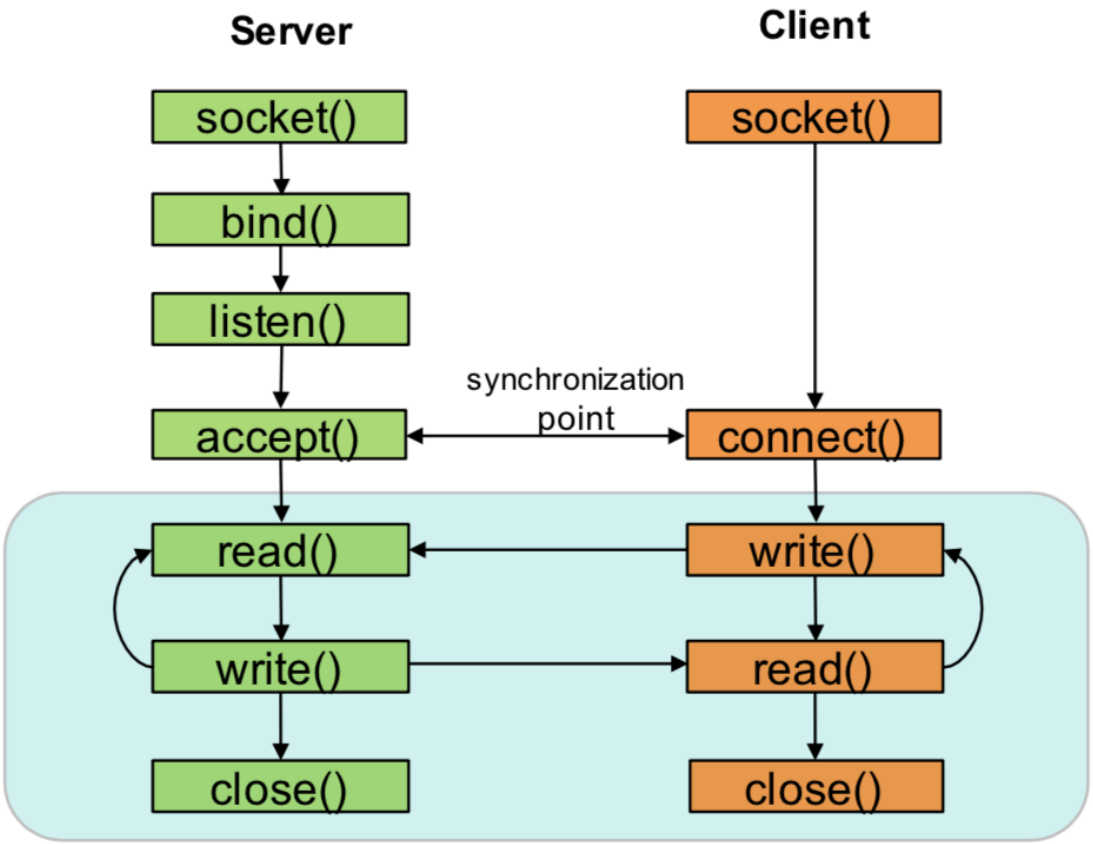
- Step 4: Wait for and accept a connection → **accept()**
  - accept an incoming connection on the listening socket
  - If no pending connections, the accept() is **blocked**

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
                                        Returns file descriptor on success, or −1 on error
```

  - **sockfd** : file descriptor obtained from socket()
  - **addr** : pointer to structure that specify address of the peer address
  - **addrlen** : pointer to the size of the address structure
  - If we are not interested in the address of the peer socket, addr and addrlen are specified as NULL and 0, respectively.
  - **Return value** : **a new socket** that is connected to the peer socket
    - The listening socket remain open and can be used to accept further connection

Writing timeserv.c : A Timer Server

- Step 5: Transfer data → read(), write()
- Step 6: Close connection → close()



Ex4: timeserv.c

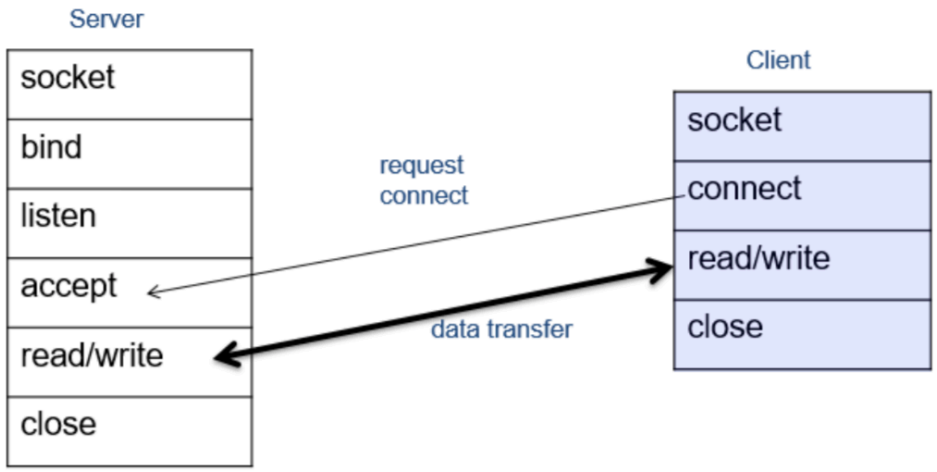
```c
/* timeserv.c - a socket-based time of day server
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>

#define PORTNUM 13000   /* our time service phone number */
#define HOSTLEN 256
#define oops(msg)   { perror(msg); exit(1); }

int main(int ac, char *av[])
{
    struct sockaddr_in saddr;   /* build our address here   */
    struct hostent *hp;     /* this part of our     */
    char hostname[HOSTLEN];     /* address          */
    int sock_id, sock_fd;       /* line id, file descriptor */
    FILE *sock_fp;          /* use socket as steam      */
    char *ctime();          /* convert secs to string   */
    time_t thetime;         /* the time we report       */

    /*
     * Step 1: ask kernel for a socket
     */
    sock_id = socket( PF_INET, SOCK_STREAM, 0);
    if( sock_id == -1)
        oops( "socket" );
```

Ex4: timeserv.c

```
/*
 * Step 2: bind address to socket. Address is host, port
 */
bzero ( (void *) &saddr, sizeof( saddr) ); /* clear out struct  */

gethostname( hostname, HOSTLEN );      /* where am I ?  */
hp = gethostbyname( hostname );        /* get info about host */
                        /* fill in host part */
bcopy( (void *) hp->h_addr, (void *) &saddr.sin_addr, hp->h_length);
saddr.sin_port = htons(PORTNUM);       /* fill in socket port */
saddr.sin_family = AF_INET;        /* fill in addr family */

if( bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr)) != 0 )
    oops( "bind" );

/*
 * Step 3: allow incoming calls with Qsize=1 on socket
 */

if ( listen (sock_id, 1) != 0 )
    oops( "listen" );

/*
 * main loop: accept(), write(), close()
 */

while (1) {
    sock_fd = accept (sock_id, NULL, NULL); /* wait for call */
        printf("Wow! got a call!\n");
    if( sock_fd == -1 )
        oops( "accept" );   /* error getting calls */

    sock_fp = fdopen( sock_fd, "w");/* we'll write to the   */
    if( sock_fp == NULL )       /* socket as a sream    */
        oops ("fdopen" );   /* unless we can't  */

    thetime = time(NULL);       /* get time     */
                    /* and convert to string */
    fprintf( sock_fp, "The time here is .." );
    fprintf( sock_fp, "%s", ctime(&thetime) );
    fclose( sock_fp );      /* release connection   */
}
}
```

Writing timecInt.c: a Time Client

action	syscall
1. Get a phone line	socket()
2. Call the server	connect()
3. Transfer data	read()/write()
4. Hang up	close()



Writing timecInt.c: a Time Client

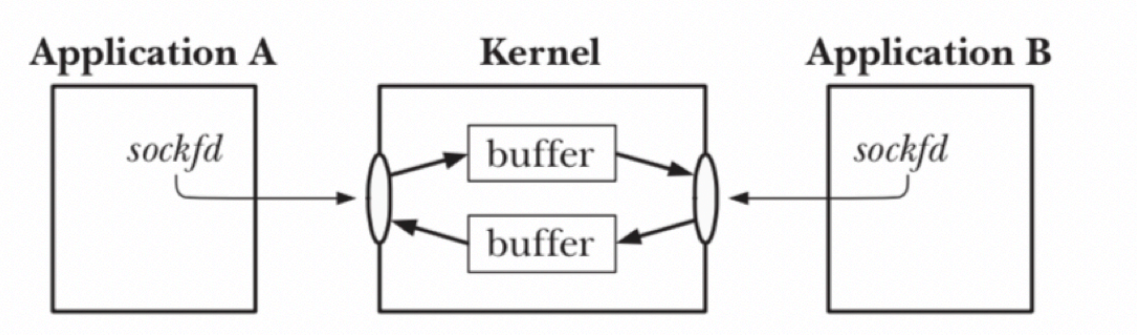
- Establish a connection : **connect()**
  - connect the active socket to the listening socket

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                                    Returns 0 on success, or −1 on error
```

  - **sockfd** : file descriptor of the active socket obtained form socket()
  - **addr** : pointer to structure that specify address of the listening socket
  - **addrlen :** size of the address structure

Writing timecInt.c: a Time Client

- Use read() and write() system call to perform I/O
  - Both calls may be used on each end of the connection since the sockets are bidirectional



- After a socket is closed with close()
  - read() returns zero (the end-of-file)
  - write() fails with the error EPIPE and receives a SIGPIPE signal

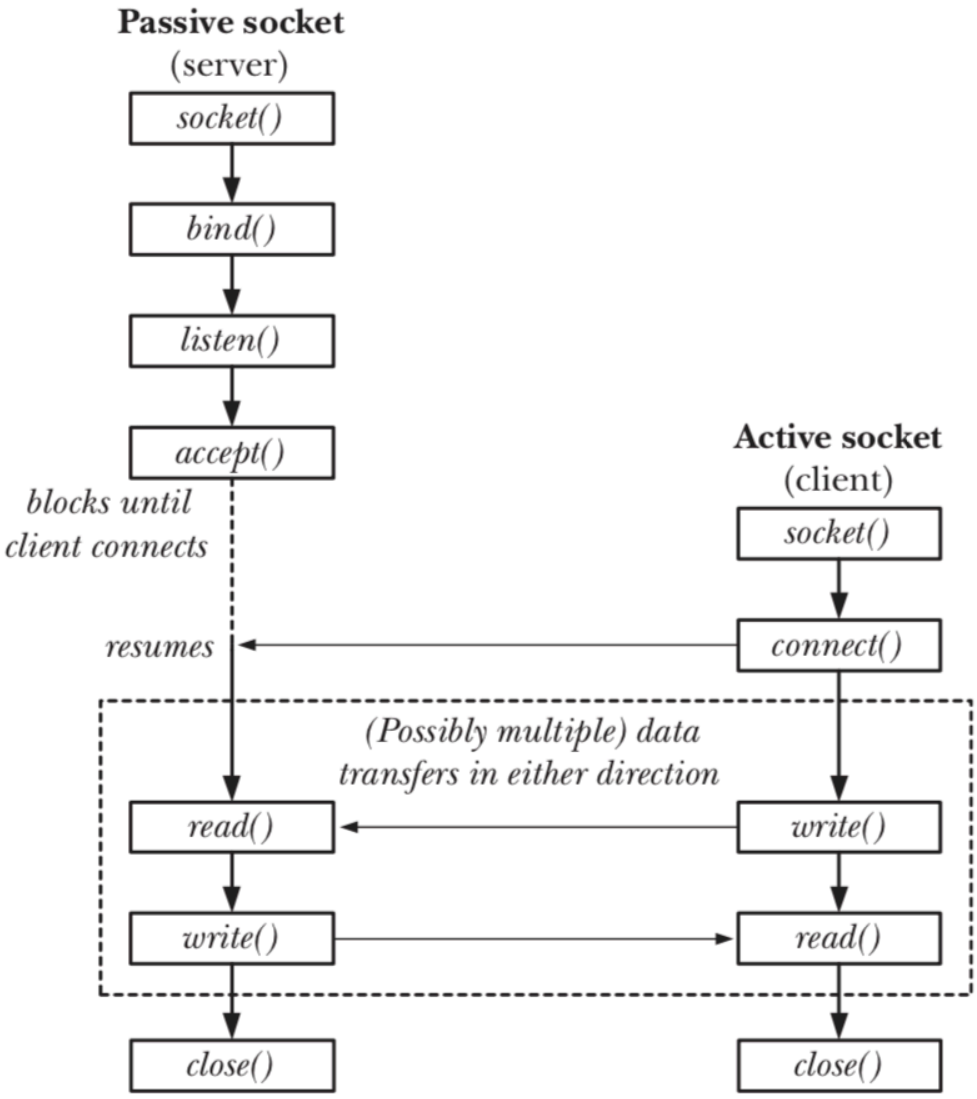
Ex5: timecInt.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <stdlib.h>
#include <strings.h>

#define oops(msg)   { perror(msg); exit(1); }

int main(int ac, char *av[])
{
    struct  sockaddr_in servadd;    /* the number to call   */
    struct  hostent *hp;        /* used to get number   */
    int sock_id/*, sock_fd*/;   /* the socket and fd    */
    char    message[BUFSIZ];    /* to receive message   */
    int messlen;        /* for message length   */


    /*
     * Step 1: Get a socket
     */
    sock_id = socket( AF_INET, SOCK_STREAM, 0); /* get a line */
    if( sock_id == -1)
        oops( "socket" );           /* or fail */
```

Ex5: timeclnt.c

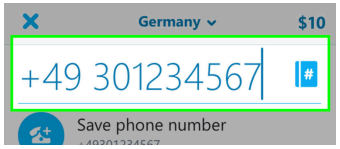
```
    /*
     * Step 2: connect to server
     *     need to build address ( host, port ) of server first
     */
    bzero ( &servadd, sizeof(servadd) );    /* zero the address */
    hp = gethostbyname(av[1]);      /* lookup host's ip # */
    if ( hp == NULL)
        oops(av[1]);
    bcopy( hp->h_addr, (struct sockaddr *)&servadd.sin_addr, hp->h_length);
    servadd.sin_port = htons(atoi(av[2]));  /* fill in port number */
    servadd.sin_family = AF_INET;       /* fill in socket type */

                        /* now dial */
    if(connect(sock_id, (struct sockaddr *)&servadd, sizeof(servadd)) != 0 )
        oops( "bind" );

    /*
     * Step 3: transfer data from server, then hangup
     */
    messlen = read(sock_id, message, BUFSIZ);   /* read stuff   */
    if ( messlen == -1 )
        oops( "read" );
    if( write(1, message, messlen ) != messlen )    /* and write to */
        oops( "write" );            /* stdout   */
    close( sock_id );
}
```

Summary of system calls used with Socket

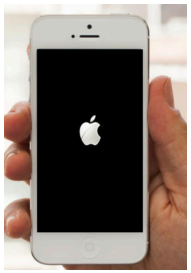


socket() : create a socket =

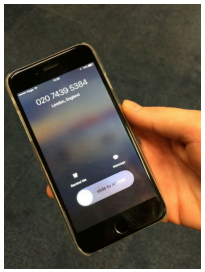
bind() : bind a socket to address=



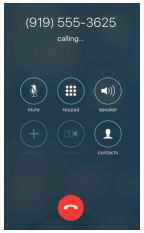
listen() : notify the kernel of its willingness to accept incoming connections =



accept() : notify the kernel of its willingness to accept incoming connections =



connect() : establish a connection



Appendix

```
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 */
};
#define h_addr h_addr_list[0] /* for backward compatibility */
```

```
struct sockaddr {
  unsigned short sa_family; /* Address family (e.g. AF_INET) */
  char sa_data[14]; /* Family-specific address information */
}
```

```
struct in_addr {
  unsigned long s_addr; /* Internet address (32 bits) */
}
```

```
struct sockaddr_in {
  unsigned short sin_family; /* Internet protocol (AF_INET) */
  unsigned short sin_port; /* Address port (16 bits) */
  struct in_addr sin_addr; /* Internet address (32 bits) */
  char sin_zero[8]; /* Not used */
}
```