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

/*
 *      socklib.c
 *
 *      This file contains functions used lots when writing internet
 *      client/server programs.  The two main functions here are:
 *
 *      make_server_socket( portnum )   returns a server socket
 *                                      or -1 if error
 *
 *      connect_to_server(char *hostname, int portnum)
 *                                      returns a connected socket
 *                                      or -1 if error
 */

int
make_server_socket( int portnum )
{
        struct  sockaddr_in   saddr;   /* build our address here */
        struct  hostent         *hp;   /* this is part of our    */
        char    hostname[256];         /* address                */
        int     sock_id;               /* line id, file desc     */

       /*
        *      step 1: build our network address
        *               domain is internet, hostname is local host,
        *               port is some arbitrary number
        */

       gethostname( hostname , 256 );          /* where am I ?         */
       hp = gethostbyname( hostname );         /* get info about host  */
       if ( hp == NULL )
                return -1;

       bzero( &saddr, sizeof(saddr) );         /* zero struct          */
                                               /* fill in hostaddr     */
       bcopy( hp->h_addr, &saddr.sin_addr, hp->h_length);
       saddr.sin_family = AF_INET ;            /* fill in socket type  */
       saddr.sin_port = htons(portnum);        /* fill in socket port  */

       /*
        *      step 2: ask kernel for a socket, then bind address
        */

       sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a socket */
       if ( sock_id == -1 ) return -1;
                                                       /* give it      */
       if ( bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr)) != 0 )
                return -1;                              /* an address   */

       /*
        *      step 3: tell kernel we want to listen for calls
        */

       if ( listen(sock_id, 1) != 0 ) return -1;

        return sock_id;
}

int
connect_to_server( char *hostname, int portnum )
{
        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 */

       /*
        *      build the network address of where we want to call
        */
```

```
        hp = gethostbyname( hostname );
        if ( hp == NULL ) return -1;

        bzero( &servadd, sizeof( servadd ) );   /* zero the address     */
        servadd.sin_family = AF_INET ;          /* fill in socket type  */
                                                /* and machine address  */
        bcopy( hp->h_addr, &servadd.sin_addr, hp->h_length);
        servadd.sin_port = htons(portnum);      /* host to num short    */

        /*
         *        make the connection
         */

        sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a line   */
        if ( sock_id == -1 ) return -1;                  /* or fail      */
                                                        /* now dial     */
        if( connect(sock_id,(struct sockaddr *)&servadd,sizeof(servadd)) !=0 )
                return -1;

        /*
         *      we're connected to that number, return the socket
         */

         return sock_id ;
}

:::::::::::::: timec.c ::::::::::::::
#include       <stdio.h>

/*
 * timec.c
 *
 *      connects to time server and prints out report
 *      args: hostname portnum
 *      action: connect to server, read and print one line
 */

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

main(int ac, char *av[])
{
        int    fd;                          /* the connected socket     */
        char   *hostname;

        if ( ac != 3 ){
                fprintf(stderr,"usage: %s hostname portnum\n", av[0] );
                exit(1);
        }
        hostname = av[1];

        /*
         * make the connection
         */
        fd = connect_to_server( hostname, atoi(av[2]) );
        if ( fd == -1 ) oops( "connecting to server" );
        talk_with_server( fd );
        close(fd);
}

/*
 * print a header, then copy data from fd to stdout
 */
talk_with_server( int fd )
{
        char   buf[BUFSIZ];             /* to receive message */
        int    messlen;                 /* for message length */

        printf( "The time report from the server: ");
        fflush(stdout);
        while ((messlen = read( fd, buf, BUFSIZ )) > 0 )
                if ( write( 1, buf, messlen ) != messlen )
                        oops( "write" );

        if ( messlen == - 1 ) oops( "read" ) ;
}
```

```
:::::::::::::: timed1.c ::::::::::::::
#include        <stdio.h>
#include          <time.h>

/*
 * timed1.c
 *      a date-time server that replies with the output of ctime
 */

#define PORTNUM 9999
#define oops(s) { perror(s); exit(1) ; }

main()
{
        int     sock, fd;

        sock = make_server_socket( PORTNUM );
        if ( sock == -1 ) oops( "make_server_socket" );

        while( ( fd = accept(sock,NULL,NULL) ) != -1 )
        {
                process_request(fd);
                close(fd);
        }
}
process_request(fd)
/*
 * send the date out to the client via fd
 */
{
        time_t  now;
        char    *cp, *ctime();

        time( &now );                   /* get time from system */
        cp = ctime( &now );             /* convert to string    */

        if ( write( fd, cp, strlen(cp) ) != strlen(cp) )
                perror("write");
}

:::::::::::::: timed2.c ::::::::::::::
#include        <stdio.h>
#include        <time.h>

/*
 * timed2.c
 *      a date-time server that replies with the output of ctime
 *      this version uses fork() and exec() and dup2() to get
 *      the date command to generate the time and date and send
 *      it over the fd.
 */

#define PORTNUM 9998
#define oops(s) { perror(s); exit(1) ; }

main()
{
        int     sock, fd;

        sock = make_server_socket( PORTNUM );
        if ( sock == -1 ) oops( "make_server_socket" );

        while( ( fd = accept(sock,NULL,NULL) ) != -1 )
        {
                process_request(fd);
                close(fd);
        }
}
```

```
process_request(fd)
/*
 * send the date out to the client via fd
 */
{
        int     pid = fork();

        if ( pid == -1 ) return ;       /* error getting a new process  */

        if ( pid != 0 ){                /*  parent */
                wait(NULL);             /*         do we have to wait?  */
                return;                 /*         what about zombies?  */
        }

        /* child code here */
        dup2( fd, 1 );                  /* moves socket to fd 1 */
        close(fd);                      /* closes socket        */
        execlp("date","date",NULL);     /* exec date            */
        oops("execlp date");
}

:::::::::::::: ws_partial.c ::::::::::::::
#include       <stdio.h>
#include       <sys/types.h>
#include       <sys/stat.h>
#include       <strings.h>

/*
 * WebServ.c - a minimal web server
 *
 *    usage: ws portnumber
 * features: supports the GET command only
 *           runs in the current directory
 *           forks a new child to handle each request
 *           has MAJOR security holes, for demo purposes only
 *           has many other weaknesses, but is a good start
 */

main(int ac, char *av[])
{
        int     sock, fd;
        FILE    *fpin;
        char    request[BUFSIZ];

        if ( ac == 1 ){
                fprintf(stderr,"usage: ws portnum\n");
                exit(1);
        }
        sock = make_server_socket( atoi(av[1]) );
        if ( sock == -1 ) exit(2);

        /* main loop here */

        while(1){
                /* take a call and buffer it */
                fd = accept( sock, NULL, NULL );
                fpin = fdopen(fd, "r" );

                /* read request */
                fgets(request,BUFSIZ,fpin);
                printf("got a call: request = %s", request);
                read_til_crnl(fpin);

                /* do what client asks */
                process_rq(request, fd);

                fclose(fpin);
        }
}
```

```
/* ------------------------------------------------------ *
   read_til_crnl(FILE *)
   skip over all request info until a CRNL is seen
   ------------------------------------------------------ */

read_til_crnl(FILE *fp)
{
        char    buf[BUFSIZ];
        while( fgets(buf,BUFSIZ,fp) != NULL && strcmp(buf,"\r\n") != 0 )
                ;
}

/* ------------------------------------------------------ *
   process_rq( char *rq, int fd )
   do what the request asks for and write reply to fd
   handles request in a new process
   rq is HTTP command:  GET /foo/bar.html HTTP/1.0
   ------------------------------------------------------ */

process_rq( char *rq, int fd )
{
        char    cmd[BUFSIZ], arg[BUFSIZ];
        char    *item;

        /* create a new process and return if not the child */
        if ( fork() != 0 )
                return;

        if ( sscanf(rq, "%s%s", cmd, arg) != 2 )
                return;
        item = arg+1;                   /* skip leading / */
        if ( strcmp(cmd,"GET") != 0 )
                cannot_do(fd);
        else if ( not_exist( item ) )
                do_404(item, fd );
        else if ( isadir( item ) )
                do_ls( item, fd );
        else if ( ends_in_cgi( item ) )
                do_exec( item, fd );
        else
                do_cat( item, fd );
}

/* ------------------------------------------------------ *
   the reply header thing: all functions need one
   if content_type is NULL then don't send content type
   ------------------------------------------------------ */

header( FILE *fp, char *content_type )
{
        fprintf(fp, "HTTP/1.0 200 OK\r\n");
        if ( content_type )
                fprintf(fp, "Content-type: %s\r\n", content_type );
}

/* ------------------------------------------------------ *
   simple functions first:
        cannot_do(fd)       unimplemented HTTP command
    and do_404(item,fd)     no such object
   ------------------------------------------------------ */

cannot_do(int fd)
{
        FILE    *fp = fdopen(fd,"w");

        fprintf(fp, "HTTP/1.0 501 Not Implemented\r\n");
        fprintf(fp, "Content-type: text/plain\r\n");
        fprintf(fp, "\r\n");

        fprintf(fp, "That command is not yet implemented\r\n");
        fclose(fp);
}
```

```
do_404(char *item, int fd)
{
        FILE    *fp = fdopen(fd,"w");

        fprintf(fp, "HTTP/1.0 504 Not Found\r\n");
        fprintf(fp, "Content-type: text/plain\r\n");
        fprintf(fp, "\r\n");

        fprintf(fp, "The item you requested: %s\r\nis not found\r\n",
                        item);
        fclose(fp);
}

/* ------------------------------------------------------ *
   the directory listing section
   isadir() uses stat, not_exist() uses stat
   do_ls runs ls. It should not
   ------------------------------------------------------ */
isadir(char *f)
{
        struct stat info;
        return ( stat(f, &info) != -1 && S_ISDIR(info.st_mode) );
}
not_exist(char *f)
{
        struct stat info;
        return( stat(f,&info) == -1 );
}

do_ls(char *dir, int fd)
{
        FILE    *fp ;
        int     pid;

        fp = fdopen(fd,"w");
        header(fp, "text/plain");
        fprintf(fp,"\r\n");
        fflush(fp);

        dup2(fd,1);
        dup2(fd,2);
        close(fd);
        execlp("ls","ls","-l",dir,NULL);
        perror(dir);
        exit(1);
}

/* ------------------------------------------------------ *
   the cgi stuff.  function to check extension and
   one to run the program.
   ------------------------------------------------------ */
char *
file_type(char *f)
/* returns 'extension' of file */
{
        char    *cp;
        if ( cp = strrchr(f, '.' ) )
                return cp+1;
        return "";
}

ends_in_cgi(char *f)
{
        return ( strcmp( file_type(f), "cgi" ) == 0 );
}

/* executes a program with stdout sent to client via socket */
do_exec( char *prog, int fd )
{
        /* exercise for reader */
}
/* ------------------------------------------------------ *
   do_cat(filename,fd)
   sends back contents after a header
   e.g. .html gets  Content-type: text/html
        .gif  gets  Content-type: image/gif
   ------------------------------------------------------ */
do_cat(char *f, int fd)
{
        /* exercise for reader */
}
```