Ch12. Connections and Protocols: Writing a Web Server

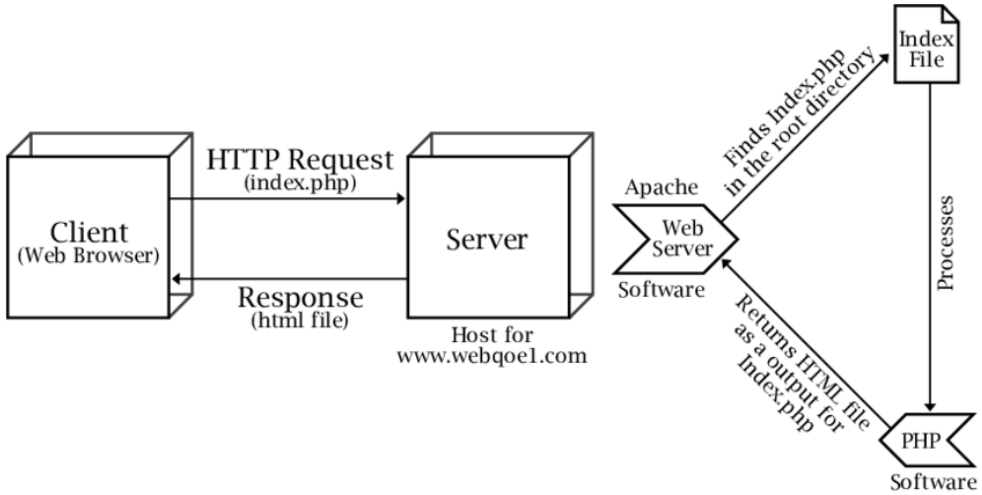
Prof. Seokin Hong

Kyungpook National University

Fall 2018

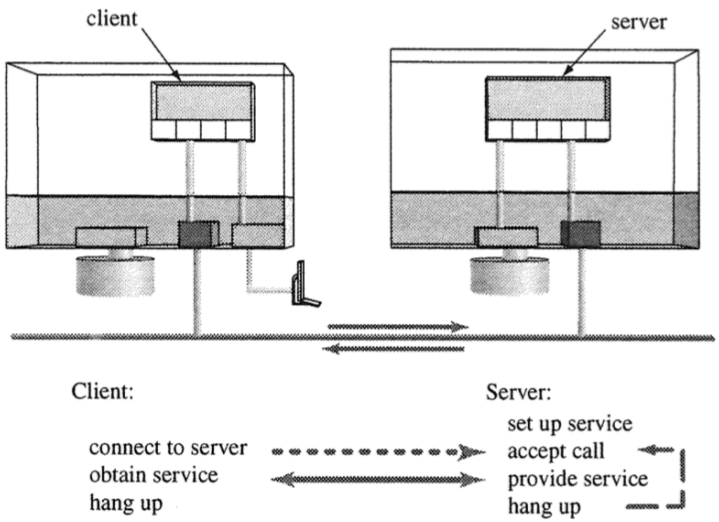
Web Server?

- Using the world wide web is easy
  - Type a web location into a browser or click on a link, and a web page is delivered from a **remote computer**



Three main operations of Server/Client System

- Three main operations
  - Server sets up a service
  - Client connects to a server
  - Server and client do business



- Most socket-based client/server systems are pretty similar
  - E-mail, file transfer, distributed databases,….
- Once you understand one socket-based client/server system, you can understand most of the other ones

Setting Up a Server Socket

- **Three steps**

  1. Create socket
     sock = **socket**(PF_INET, SOCK_STREAM, 0)
  2. Give the socket an address
     bind(sock, &addr, sizeof(addr))
  3. Arrange to take incoming calls
     listen(sock, queue_size)

- The three-step can be combined into a single function
  - sock = **make_server_socket** (int portnum)
    return -1 if error, or a server socket listening at port “portnum

Operation 2 : Connecting to a Server

1. Create a socket
   sock = **socket**(PF_INET, SOCK_STREAM, 0)
2. Use the socket to connect to a server
   connect(sock, &serv_addr, size(serv_addr))

- The parameters are “hostname” and “port number” of serv_addr.
- Thus we can be combined into a single function
  - fd = **connect_to_server**(hostname, portnum)
    returns -1 if error, or a fd open for reading and writing connected to the socket at port “portnum” on host “hostname”

socklib.c

```
// socklib.h
#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 <strings.h>

// register functions to be implemented
int make_server_socket(int);
int make_server_socket_q(int, int);
int connect_to_server(char*, int);
#define HOSTLEN 256
#define BACKLOG 1    // only one request at a time

int make_server_socket(int portnum){
    return make_server_socket_q(portnum, BACKLOG);
}
```

socklib.c

```
int make_server_socket_q(int portnum, int backlog)
{
    struct  sockaddr_in   saddr;          /* build our address here */
    structhostent*  hp;                   /* this is part of our          */
    char   hostname[HOSTLEN];             /* address                      */
    int sock_id;                          /* the socket                   */

    sock_id = socket(PF_INET, SOCK_STREAM, 0);   /* get a socket  */
    if ( sock_id == -1 )
        return -1;

    /** build address and bind it to socket **/
    bzero((void *)&saddr, sizeof(saddr));          /* clear out struct     */
    gethostname(hostname, HOSTLEN);                /* where am I ?         */
    hp = gethostbyname(hostname);                  /* get info about host  */
```

socklib.c

```
bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);   /* fill in host part    */
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 )
        return -1;

/** arrange for incoming calls **/
    if ( listen(sock_id, backlog) != 0 )
        return -1;
    return sock_id;
}
```

socklib.c

```
int connect_to_server(char *host, int portnum)
{
    int                 sock;
    struct sockaddr_in  servadd;    /* the number to call */
    struct hostent      *hp;        /* used to get number */

    /** Step 1: Get a socket **/
    sock = socket( AF_INET, SOCK_STREAM, 0 );   /* get a line   */
    if ( sock == -1 )
        return -1;

    /** Step 2: connect to server **/
    bzero( &servadd, sizeof(servadd) );         /* zero the address    */
    hp = gethostbyname( host );     /* lookup host's ip #   */
    if (hp == NULL)
        return -1;
```

socklib.c

```
bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr, hp->h_length);
    servadd.sin_port = htons(portnum);          /* fill in port number  */
    servadd.sin_family = AF_INET ;              /* fill in socket type  */

    if ( connect(sock,(struct sockaddr *)&servadd, sizeof(servadd)) !=0)
        return -1;

    return sock;
}
```

Operation 3: Client/Server Conversation

- The Generic Client

```
main()
{
    int fd;

    fd = connect_to_server(host, port);          /* call the server   */
    if ( fd == -1 )
        exit(1);                                 /* or die            */
    talk_with_server(fd);                        /* chat with server  */
    close(fd);                                   /* hang up when done */
}
```

- The Generic Server

```
main()
{
    int sock, fd;                /* socket and connection */

    sock = make_server_socket(port);
    if ( sock == -1 )
        exit(1);

    while(1){
       fd = accept(sock, NULL, NULL);            /* take next call    */
       if ( fd == -1 )
           break;                                /* or die            */
       process_request(fd);                      /* chat with client  */
       close(fd);                                /* hang up when done */
    }
}
```

timeserv/timeclnt using socklib.c

- What we need to implement
  - **talk_with_server()** for client
  - **process_request()** for server

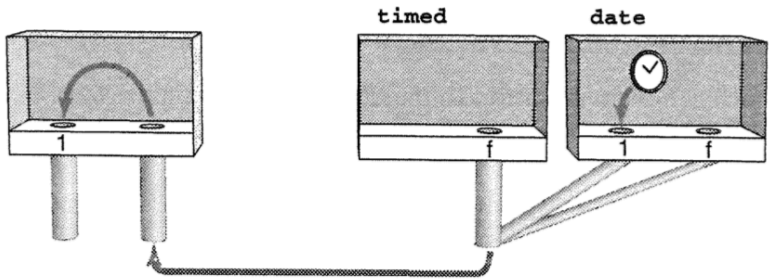
```
talk_with_server(fd)
{
   char buf[LEN];
   int  n;

   n=read(fd,buf,LEN);
   write(1,buf,n);
}
```

```
process_request(fd)
{
    time_t now;
    char *cp;

    time(&now);
    cp  = ctime(&now);
    write(fd, cp, strlen(cp));
}
```

A Second Version of the Server: Using fork




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

        switch(pid) {
                case -1: return;        /* cannot provide service */

                case 0: dup2(fd, 1);    /* child runs date        */
                        close(fd);      /* by redirecting stdout */
                        execl("/bin/date", "date", NULL);
                        oops("execlp");/* or quits */

                default: wait(NULL);    /* parent wait for child */
        }
}
```

What is advantage of the second version of time server?

Server Design Question: DIY or Delegate

- Two types of server design
  - **Do it yourself :** Serer does the work itself
  - **Delegate :** the server forks a process to do the work

- **Advantages and disadvantages of each design**
  - DIY for quick, simple tasks

  - **Delegate for slower, more complex tasks.**
    - **Can handle many requests simultaneously.**
    - In order to serve several requests at once, a **server should not "wait" for child to finish.**
    - How to prevent child processes from being zombies?

How to prevent child processes from being zombies?

- Rather than wait for child to die, a parent can arrange to receive a signal when a child dies
- When a child process exists or is killed, kernel sends SIGCHLD (SIGCHLD is ignored by default)
- Parent process may set a signal handler. That handler can call wait.

Using SIGCHLD for Zombie Prevention

```
main(){
        int sock, fd;

        signal(SIGCHLD, child_waiter);
        if((sock = make_server_socket(PORTNUM)) == -1) oops("make_server_socket");
        while(1) {
                fd = accept(sock, NULL, NULL);
                if(fd == -1) break;
                process_request(fd);
                close(fd);
        }
}

void child_waiter(int signum)
{
        wait(NULL);
}
```

Using SIGCHLD for Zombie Prevention

- Problem 1.
  - Jump to the signal handler interrupts the accept system call.
  - When interrupted by signal, accept returns -1, and sets errno to EINTR.
  - Our code treats the value of -1 from accept as an error and breaks from the main loop.
  - How to distinguish between real error and an interrupted system call?

Using SIGCHLD for Zombie Prevention

- Problem 2.
  - What happens if several child processes exit at almost exactly the same time?
  - Unix blocks signals, but does not queue signals. So newly-coming signals are lost.
  - If children exit while the parent is in the handler, signals from the children are lost.
  - How to prevent this?

Using SIGCHLD for Zombie Prevention

- Solution

  Call wait enough times to mop up all terminated processes.

```
void child_waiter (int signum)
{
        while (waitpid (-1, NULL, WNOHANG) > 0) ;
}
```

- waitpid()
  - First argument is the ID number of the process.
    - -1 tells waitpid to wait for all children.
  - Second argument is the pointer to the integer to receive the status.
  - The last argument specifies options.
    - WNOHANG option tells waitpid not to wait if there are no zombies.

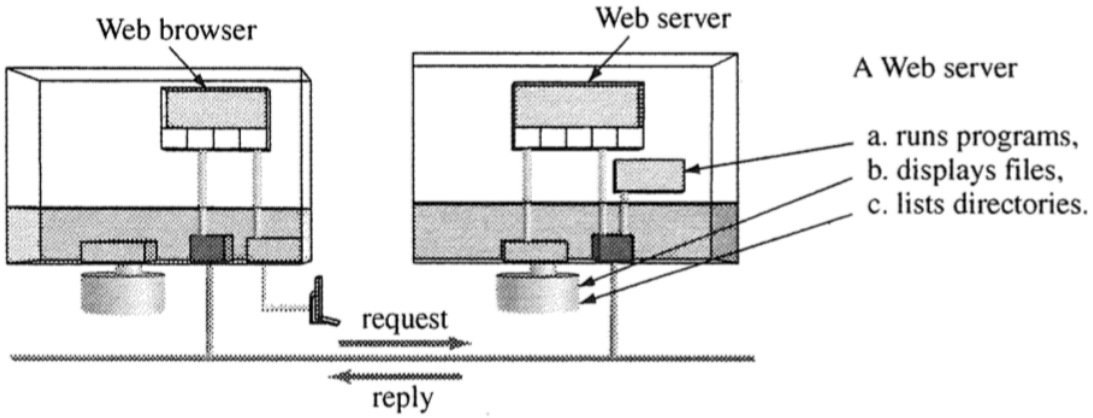
What a Web Server Does

- A Web server is a simple concept. A Web server is a program that performs the three most common user operations :

  (a) list directories

  (b) cat files

  (c) run programs



What a Web Server Does

- Logic of the Web Server and Client

client		server
user selects a link		
`connect` to server	——>	`accept` a call
`write` a request	——>	`read` a request
		handle request:
		directory: list it
		regular file: `cat` it
		`.cgi` file: run it
		not exist: error message
`read` the reply	<——	`write` a reply
hangup		
display the reply		
`html`: render it		
image: draw it		
sound: play it		
repeat		

Planning Our Web Server

- What operations do we need to code?

(a) Set up the server

We can use make_server_socket from socklib.c

(b) Accept a call

Use accept to get the file descriptor to the client. We can use fdopen to make that file descriptor into a buffered stream.

(c) Read a request

What does a request look like? How does the client ask for something? We need to study this one more.

(d) handle the request

We know how to list directories, cat files, and run programs. We can use opendir and readdir, open and read, dup2 and exec.

(e) Send a reply

What does a reply look like? What does the client expect to see? This one also requires more study.

The Protocol of a Web Server

$ telnet www.prenhall.com 80

Connect a web server (port number : 80)

Trying 165.193.123.253…

Connected to www.prenhall.com.

Escape character is '^]'.

GET /index.html HTTP/1.0

- HTTP request
  - 1st Arg: Command
  - 2nd Arg: Parameter
  - 3rd Arg: version of the protocol

HTTP Response

HTTP/1.1 200 OK

Date: Tue, 20 Nov 2018 21:27:27 GMT

Server: Apache

X-Powered-By: PHP/5.6.30

Content-Length: 717

Connection: close

Content-Type: text/html; charset=UTF-8

Header

- Status line
  - 1st Arg: version of the server
  - 2nd Arg: response code number
    - "200" means OK!
    - "404" means "Not found"

```
<HTML><HEAD>
<META HTTP-EQUIV="Refresh" CONTENT="0; URL=http://vig.prenhall.com/">
</HEAD><BODY></BODY></HTML>
<!-- ------------------------------------------------------------>
<!--          Caught you peeking!          -->
<!-- ------------------------------------------------------------>
```

Content

Connection closed by foreign host.

$

Writing a Web Server

- The main loop of our Web server looks like the following :

```
while(1) {
    fd = accept(sock, NULL, NULL);        /* take a call          */
    fpin = fdopen(fd, “r”);               /* make it a FILE *     */
    fgets(fpin, request, LEN);            /* read client request  */
    read_until_crnl(fpin);                /* skip over arguments */
    process_rq(request, fd);              /* reply to client      */
    fclose(fpin);                         /*  hang up connection  */
}
```

Webserv Source Code

```
/* webserv.c - a minimal web server (version 0.2)
 *      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
 *      build: cc webserv.c socklib.c -o webserv
 */
#include      <stdio.h>
#include      <sys/types.h>
#include      <sys/stat.h>
#include      <string.h>
```

```
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);
        }
}
```

Webserv Source Code

```
/* ------------------------------------------------------ *
   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 )
        ;
}
```

Webserv Source Code

```
/* ---------------------------------------- *
   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];
    /* create a new process and return if not the child */
    if ( fork() != 0 )
        return;
    strcpy(arg, "./");                  /* precede args with ./   */
    if ( sscanf(rq, "%s%s", cmd, arg+2) != 2 )
        return;
    if ( strcmp(cmd,"GET") != 0 )
        cannot_do(fd);
    else if ( not_exist( arg ) )       do_404(arg, fd );
    else if ( isadir( arg ) )           do_ls( arg, fd );
    else if ( ends_in_cgi( arg ) )  do_exec( arg, fd );
    else                                do_cat( arg, fd );
}
```

Writing a Web Server

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

    fp = fdopen(fd, “w”);                          /* make socket into a FILE*    */
    header(fp, “text/plain”);                      /* send HTTP reply header      */
    fprintf(fp, “\r\n”);                           /* and end of header mark      */
    fflush(fp);                                    /* force to socket             */

    dup2(fd, 1);                                   /* make socket stdout          */
    dup2(fd, 2);                                   /* make socket stderr          */
    close(fd);                                     /* close socket                */
    execl(“/bin/ls”, “ls”, “-l”, dir, NULL);       /* ls -l does the work         */
    perror(dir);                                   /* or it doesn’t               */
    exit(1);                                       /* child exits                 */
}
```

Webserv Source Code

```
/* ------------------------------------------------------ *
   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 );
}
```

Webserv Source Code

```
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);
}
```

Webserv Source Code

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

    fprintf(fp, "HTTP/1.0 404 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);
}
```

Webserv Source Code

```
/* ------------------------------------------------------ *
   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 ;

    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);
}
```

Webserv Source Code

```
/* ------------------------------------------------- *
   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, '.' )) != NULL )
        return cp+1;
    return "";
}

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

```
do_exec ( char *prog, int fd )
{
    FILE       *fp ;

    fp = fdopen(fd,"w");
    header(fp, NULL);
    fflush(fp);
    dup2(fd, 1);
    dup2(fd, 2);
    close(fd);
    execl(prog,prog,NULL);
    perror(prog);
}
```

Webserv Source Code

```
/* ---------------------------------------------- *
   do_cat(filename,fd)
   sends back contents after a header
   ---------------------------------------------- */
do_cat(char *f, int fd)
{
    char     *extension = file_type(f);
    char     *content = "text/plain";
    FILE     *fpsock, *fpfile;
    int      c;

    if ( strcmp(extension,"html") == 0 )
        content = "text/html";
    else if ( strcmp(extension, "gif") == 0 )
        content = "image/gif";
    else if ( strcmp(extension, "jpg") == 0 )
        content = "image/jpeg";
    else if ( strcmp(extension, "jpeg") == 0 )
        content = "image/jpeg";

    fpsock = fdopen(fd, "w");
    fpfile = fopen( f , "r");

    if ( fpsock != NULL && fpfile != NULL )
    {
        header( fpsock, content );
        fprintf(fpsock, "\r\n");

        while( (c = getc(fpfile) ) != EOF )
            putc(c, fpsock);

        fclose(fpfile);
        fclose(fpsock);
    }
    exit(0);
}
```

Running the Web Server

- Compile the code and then run the program at a port:
  - $ cc webserv.c socklib.c -o webserv
  - $ ./webserv 12345
- Create the following shell script (hello.cgi):

```
#!/bin/sh
# hello.cgi - a cheery cgi page
printf “Content-type: text/plain\n\nhello\n”;
```

- call this scirpt hello.cgi, chmod it to 755,
- and then your browser to invoke the script with
  - http://yourhostname:12345/
  - http://yourhostname:12345/hello.cgi
  - http://yourhostname:12345/webserv.c
  - http://yourhostname:12345/mypage